5101

Burroughs

B 6000/B 7000

APL/700

USER REFERENCE MANUAL

(RELATIVE TO MARK II.8 RELEASE)



Printed in U.S. America

3-77

5000813

COPYRIGHT © 1974, 1975, 1977 BURROUGHS CORPORATION

Burroughs Corporation believes the information described in this manual to be accurate and reliable, and much care has been taken in its preparation. However, the Corporation cannot accept any responsibility, financial or otherwise, 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
	Properties and Features	1-2
	Use Requirements	1-2
	APL/700 Interactive Environment	1-3
	Usage Modes	1-4
	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 and Usercode	1-7
	Workspaces, Library and Files	1-8
	Self Protection	1-8
	Security and Sharing	1-9
2	INTERACTING WITH APL/700	2-1
	APL Terminal Keyboard Configurations	2-2
	APL Character Set and Special Keys	2-3
	Typing Conventions	2-5
	Connection with the APL/700 System	2-6
	Sign-On	2-6
	Transaction Entries	2-8
	Entry Editing	2-8
	Correcting Typing Errors within an Entry	2-9
	Editing a Prior Entry	2-9
	Recall of Prior Entries	2-11
	Sign-Off	2-13
	Recovery Operations	2-14

Table of Contents (Cont)

Section	Title	Page
3	SYSTEM COMMANDS	3-1
	Command Format	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
	Messages	3-13
	Other Message Control Systems	3-14
	Keywords and ASCII Characters	3-15
4	THE APL/700 LANGUAGE	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	4-11
5	PRIMITIVE FUNCTIONS AND OPERATORS	5-1
	Selection and Assignment	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

Table of Contents (Cont)

Section	Title	Page
5	Primitive Operators	5-19
	Outer Product Operator	5-20
	Reduction Operator	5-22
	Scan Operator	5-24
	Inner Product	5-26
	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
	Implicit Format Function	5-58
	Numeric Format Function	5-60
	Character Format Function	5-62
6	SYSTEM VARIABLES, SYSTEM FUNCTIONS AND SHARED VARIABLES	6-1
	System Variables	6-1
	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 FUNCTIONS	7-1
	File System Characteristics	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 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, Detach	7-9
	File Component Existence	7-10
	File System Interrogate, Status, Query	7-11
8	FUNCTION DEFINITION, EDITING AND EXECUTION	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
	Execution of Defined Functions	8-18
	Scope of Names	8-18
	Execution Control Sequence	8-18
	Multiple Instances	8-19
	Discarding Output During Execution	8-19
	Suspending Function Execution	8-19
	Interrupting Function Execution	8-20
	Terminating Function Execution	8-20
	Error During Execution	8-21
	Defined Function Editing Using APL Functions	8-21
	Defined Function Documentation	8-21
	Example of Function Execution, Trace, and Editing	8-22
9	ERROR REPORTS AND THEIR INTERPRETATION	9-1
	Error Reports	9-1
	Unimplemented Constructs	9-8

Table of Contents (Cont)

Section	Title	Page
APPENDIX		
A	Glossary	A-1
B	Workspace Content Space Considerations	B-1
C	Reference Charts	C-1
D	Aids in Puzzling Circumstances	D-1
INDEX		Index-1

List of Illustrations

Figure	Title	Page
1-1	Transaction Cycle	1-3
2-1	APL Terminal, Typical 88-Character Keyboard	2-2
2-2	APL Terminal, Typical 94-Character Keyboard	2-2
5-1	Format Functions Syntax Diagrams	5-57

List of Tables

Table	Title	Page
2-1	APL Character Set Names, Keywords, and Overstrikes	2-4
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
- Workspace library
- 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:

- Additional primitive 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 (see pages C-12 to C-15).

The APL/700 Utilities Manual, Form 5001910, describes the supported shared variable utilities that provide the APL user access to the environment outside of APL. A "ZIP" utility allows entry and execution of Work Flow Language jobs. A "System File Access" utility provides an APL interface to system files.

The 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. Operations information extracted from it appears on the APL/700 Operations Reference Card, Form 5001928.

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, 1976 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, 4 appendices and an index. The sections cover generally independent aspects of APL/700.

Section 1	User viewpoint of APL and its environment
Section 2	User interaction through a suitable terminal
Section 3	User control over the APL environment by system commands
Section 4	General properties of the APL language: its array data objects, names, and expressions
Section 5	Primitive APL functions and operators
Section 6	System variables and functions to affect the account environment; and shared variables to communicate with other processes
Section 7	File functions for workspace extension and multi-user applications with common data
Section 8	Application function definition, editing, and execution
Section 9	Error reports, repair, and resumption
Appendix A	Glossary
Appendix B	Control of space in workspace
Appendix C	Summary reference charts
Appendix D	Puzzling events and possible explanations
Index	Terms and concepts from this manual and other APL publications

The reader is encouraged to become an APL user from the start, to learn by doing. The interactive environment and concise language allow problem formulation and solution at the user's pace, rapid application development, and effective application use.

SECTION 1

APL/700 SYSTEM DESCRIPTION

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	Text processing and documentation
Inventory control	Report generation
Manufacturing scheduling	Message processing and distribution
Forecasting	Statistical analysis
Manpower management	Mathematical analysis
Resource control	Simulation and optimization
PERT	Computer aided instruction
Reservation control	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.

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 (or ACSII characters using keywords)
- an account on an APL/700 system

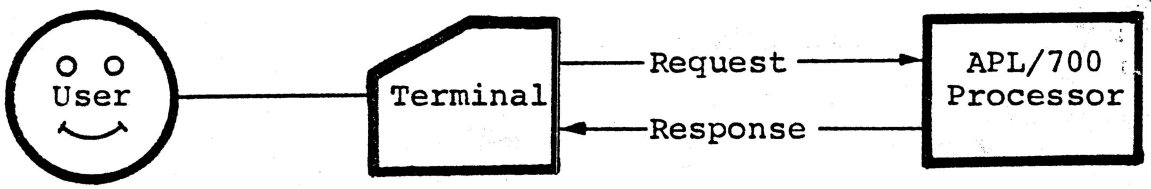
Note that typing skill is not on the above list. APL is so concise that lack of typing skill is not a significant barrier. Since the reader is encouraged to learn APL on a terminal, keyboard familiarity develops with use.

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 user striking the return key. Then the processor phase starts by receiving the entry, providing the service required, possibly generating output, and finally making the terminal ready for next user entry.



	User Phase	Processor Phase
Action	Type Entry	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 "simple" requests (that take 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 simple requests from other users. Most requests are simple.

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

immediate response for simple 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
shared access to data files

USAGE MODES

The user of APL may select one of three modes for use at any time. Each mode is recognizable by a distinguishing prompt. A prompt is the response from APL on the display before the keyboard is unlocked. Prompts include visible display and cursor position. The cursor is the indicator of next entry position.

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 □:
edit, in response to the system function □*ED*
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 shape and value. A data object has zero or more elements. All elements have the same type.

The type of an element specifies its corresponding value domain:

Type	Domain for Values
character	APL characters
numeric	values directly representable as numbers
Boolean	truth values: 1 if TRUE, 0 if FALSE

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 rightmost 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. If any element of the shape is 0, the array is empty. The rank of an object is the number of dimensions.

The value of a data element is a single, scalar member of the domain for that type. The value of an array data object is the arrayed collection of values of its elements.

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. 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.

FUNCTION

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; for example:

```
      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 set of file functions provides convenient access to extensive data.

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 provide 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 in expressions like 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 for functions is right to left within an expression. Parentheses may be used to alter this order.

If a 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.

Lines within defined functions are normally executed in sequence. Non-sequential execution may be achieved by explicit transfer to a line number, which may be computed.

There are no formal conditional or iterative control structures for defined functions. When required, these control structures are synthesized by explicit control transfers. The need for these may be generally avoided by concurrent processing on all or selected elements of structured data objects.

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 AND USERCODE

Each user must be assigned a valid usercode by the installation. "Usercode" and "account" are informally used synonymously, although strictly a usercode is the name for the corresponding account. The attributes of an account include:

- account name (usercode) and optional user-supplied password
- workspace quota
- file number quota
- file space quota
- computer use quota
- computer use rate limit
- shared variable quota
- utilities permitted
- default workspace parameters

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. The maximum active workspace size is specified by the installation.

A named copy of the active workspace can be saved in the account library of inactive workspaces. Such a workspace may be subsequently reactivated. The number of workspaces in the user library is limited to the quota established by the installation for that account.

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 either is 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.

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 in which it was locked.

A usercode (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 user can add a distinct password to the usercode, and to any of the workspaces or files of the account entered thereby. 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. The entry over the blot may be deciphered in some cases. Of course, no protection is provided against failure to blot display of the password. Password protection is the user's responsibility.

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

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 certain prior entries 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 or screen-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 are given for using an acoustically-coupled telephone interface with the Data Communication Processor; procedures for other connection means are generally simpler.

APL TERMINAL KEYBOARD CONFIGURATIONS

Figures 2-1 and 2-2 show the configurations of the most commonly available APL terminal keyboards. The terminals have 44 or 47 keys, each containing two characters (shifted and unshifted), and some number of special keys/bars.



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



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

APL CHARACTER SET AND SPECIAL KEYS

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 order of overstriking is arbitrary. The 6 extra characters on the 94 character terminals are displayed as single characters; they may be entered directly, or as overstrikes. The character set available in APL/700 is shown in Table 2-1. Names of the graphic symbols and keyword surrogates are also shown.

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

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

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

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

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

- correction of transaction entry error
- display and adjustment of a prior entry
- output termination
- execution interruption

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

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z <A>...<Z>
0 1 2 3 4 5 6 7 8 9

¨ dieresis		⍱ nor	<NOR>	v	~
¯ overbar, negative	<NEG>	⍲ nand	<NAND>	∧	~
< less than	<LT>	⍒ del stile, grade down	<GRDN>	∇	|
≤ less or equal	<LE>	⍋ delta stile, grade up	<GRUP>	Δ	|
= equal		⌽ circle stile, rotate	<ROT>	o	|
≥ greater or equal	<GE>	⍉ circle slope, transpose	<TRAN>	o	\
> greater than	<GT>	⊖ circle bar, rotate 1st	<ROTF>	o	-
≠ not equal	<NE>	⍟ log	<LOG>	o	*
v or	<OR>	⌶ Ibar	<IBAR>	⊥	T
∧ and	<AND>	⍫ del tilde	<LOCK>	∇	~
- bar		⍎ base jot, evaluate	<EVAL>	⊥	∘
÷ divide	<DIV>	⍕ top jot, format	<FMT>	T	∘
+ plus		⍀ slope bar, expand 1st	<EXPF>	\	-
× times	<MUL>	⌿ slash bar, compress 1st	<COMF>	/	-
? query		⍝ cap jot	<NOTE>	∩	∘
ω omega	<OMEGA>	⍞ quote quad	<QQ>	'	□
∈ epsilon	<EPS>	! quote dot		'	.
ρ rho	<RHO>	⍙ underbar delta	<UDELTA>	_	Δ
~ tilde	<NOT>	⌸ domino, matrix divide	<MDIV>	□	÷
↑ up arrow	<TAKE>	⍁ quad less than	<FFIRST>	□	<
↓ down arrow	<DROP>	⍂ quad equal	<FVALUE>	□	=
⍳ iota	<IOTA>	⍃ quad greater than	<FLAST>	□	>
○ circle	<CIRCLE>	⍄ quad not equal	<FEMPTY>	□	≠
* star		⍅ quad or	<FFREE>	□	v
→ right arrow, go to	<GO>	⍆ quad and	<FHOLD>	□	∧
← left arrow, assign	<IS>	⍐ quad up arrow	<FTAKE>	□	↑
α alpha	<ALPHA>	⍗ quad down arrow	<FDROP>	□	↓
⌈ upstile, maximum	<MAX>	⍈ quad tilde	<FGRAB>	□	~
⌊ downstile, minimum	<MIN>	⍜ quad circle	<FROT>	□	○
_ underbar	< >	⍇ quad right arrow	<FWRITE>	□	→
∇ del, definition	<DEF>	⍈ quad left arrow	<FREAD>	□	←
Δ delta	<DELTA>	⍌ quad del	<FKILL>	□	∇
∘ jot	<JOT>	⍏ quad delta	<FMAKE>	□	Δ
' quote, accent aigu		⍤ quad jot	<FQUERY>	□	∘
□ quad	<QD>	⍁ quad slope	<FEXP>	□	\
(left parenthesis		⍂ quad slash	<FCOM>	□	/
) right parenthesis		⊣ left tack	<LTACK>	(	-
[left bracket		⊢ right tack	<RTACK>	)	-
] right bracket		⋄ diamond	<DMD>	<	>
⊂ subset	<SUBSET>	⍷ left brace	<LBRACE>	[	∘
⊃ superset	<SUPSET>	⍸ right brace	<RBRACE>	]	∘
∩ cap, intersection	<INTER>	$ dollar		S	|
∪ cup, union	<UNION>	¢ cent	<CENT>	⊂	|
⊥ base	<BASE>	Å national 1	<NAT1>	A	'
⊤ top, represent	<REP>	Ä national 2	<NAT2>	A	¨
| stile, residue	<RES>	Ö national 3	<NAT3>	O	¨
; semicolon		Å national 4	<NAT4>	A	_ '
: colon		Ä national 5	<NAT5>	A	_ ¨
, comma		Ö national 6	<NAT6>	O	_ ¨
. dot		& ampersand	<AMPSND>	7	\
\ slope, backslash		^ circumflex	<CMFLEX>	¨	∧
/ slash		¶ paragraph	<PARAGR>	⊂	∩
space		% percent	<PERCNT>	:	/
		` accent, grave	<GRAVE>	¨	'
		# hash	<HASH>	H	=
		@ at sign	<ATSIGN>	0	α

Table 2-1 APL Character Set Names, Keywords, and Overstrikes

Other convenience keys available on some terminals include:

Linefeed (LF) — The linefeed (index on some terminals) key provides line advance, and in-line edit correction like an embedded ATTN without the displayed caret.

Repeat (REPT) — The repeat key provides repeated, automatic typing for any character.

Tabs (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 — These keys set/clear 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 conventions are:

User Entry — A user can type only when the keyboard is ready for entry (on some terminals the APL system locks the keyboard, preventing further entry when processing a user input or displaying response, on others the cursor remains at the right end of the displayed line). 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 — A user entry may exceed the length of a single display line. If so, the continuing portions are indicated with a terminal ATTN, rather than the RETN that finally completes the entry.

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 if necessary).

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:

   APL/700 VIA 1234567 (6).

   Where 1234567 and (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. Wait for the APL system prompt (cursor indents five spaces).

6. Enter the system command:

)ON Usercode [Password]

   Where Usercode is the user account identification.

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

   Example

)ON TERRY[HAPPY]

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

```
A P L / 7 0 0  VERSION 28.000
FRIDAY  28 JANUARY 1977  8:00 AM
```

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

```
SYSTEM OPERATION TODAY 0800 TO 2400
```

9. Observe the system prompt (five-space indention) and possibly keyboard unlock indicating completion of connection and readiness for transaction entry in execution mode. A light or bell may be used on some terminals to indicate the entry-ready condition, rather than a physical unlock of the keyboard.

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

```
APL/700 VIA 1234567 (6).
     )ON TERRY[HAPPY]
A P L / 7 0 0  VERSION 28.000
FRIDAY  28 JANUARY 1977  8:00 AM

SYSTEM OPERATION TODAY 0800 TO 2400
```

Three other system functions may be entered before)*ON* in step 6. The first two may also be entered after the)*ON*.

6a. If the terminal has tabs and the constant tab interval (n) needs to be reestablished at the terminal, enter:

```
)TABS n
```

6b. If a terminal without APL characters is being used, the system command

```
)KEYWORDS ON
```

may be entered to cause APL to accept and display keyword surrogates in place of special APL characters as indicated in Table 2-1.

6c. If a different Message Control System than APL is desired (for example CANDE) enter:

```
)USE MCSname
```

This command relinquishes the terminal from APL to the other MCS.

6d. If the connection response indicates some other MCS than APL, transfer to the APL MCS by entering the typical message from the left margin:

```
?MCS SYSTEM/APL ON SYSTEM
```

TRANSACTION ENTRIES

When the system sign-on process is completed, transactions (cycles consisting of user entry and system response) may be initiated. To start, make certain that the APL system has initiated the transaction cycle (by unlocking the keyboard). "Locked" is the time during which input will not be accepted: on some terminals the keyboard physically constrains entry; on other terminals, entries are ignored.

1. Using the character keys (and TAB, BKSP, and SPACE for positioning), type the desired entry. When the text of the entry is all typed, press RETN to complete the entry and initiate processing on it.

2. Await any system-provided display response, generally starting at the left margin. The response can be a transaction result, an error report, or a special prompt. The next transaction starts when the 5 space indent is received and the keyboard is unlocked.

```
      ⎕WI          keyboard unlocked, entry concluded by RETN
(DSM)DIVYA         response
                   5 space indent prompt, ready for next entry
```

An entry may be made that is longer than a single physical line of the display. After typing a portion on one line, a terminal ATTN (an ATTN at the rightmost position that the cursor has reached) causes a continuation indicator "<" under that position. Then the cursor returns to the left margin on the next line to allow another text portion to be typed. Several continuation lines may be entered, up to the installation-determined maximum number of entered characters. Each continuation portion of the entry is separate in that no backspacing into a prior portion is permitted for correction using an embedded ATTN as described below.

At any time the entire partial entry can be discarded: if the first key pressed after a final ATTN is another ATTN (an initial ATTN), a discard warning indicator ">" is displayed in the initial position. Another initial ATTN discards the partial entry. Any other keyin is treated as continuation of the prior portion(s).

If an error message is received, make the appropriate correction by editing and re-enter with a RETN. (Refer to Section 9 for error-message descriptions.)

ENTRY EDITING

A number of variations exist to facilitate editing of prior entries 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, embedded, or final; and on the kind of editing required.

CORRECTING TYPING ERRORS WITHIN AN ENTRY

A typing error in a portion of an entry may be corrected if it is noticed any time before that portion is completed:

1. Position the cursor using backspace (BKSP) under the left-most character that is in error.

2. Press ATTN. (This is an embedded ATTN; to the left of the right-most entered character.) The system displays the correction indicator (v) under the character backspaced to in step 1 and then advances the display one line. This action eliminates from the entry the characters above and to the right of the correction indicator. (On some terminals, a linefeed may be substituted for this embedded ATTN; the line containing the "v" is omitted.)

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

```
)ON MYACCT [KEYLOAD]          backspace to A, press ATTN
          v                   response is correction mark
          CK]                 enter remainder, press RETN
```

The corrected entry)*ON MYACCT* [*KEYLOCK*] is now accepted by APL.

EDITING A PRIOR ENTRY

Some prior user entries may be retrieved and modified. This capability may be used to correct an erroneous entry, to repeat a prior expression, or to develop a computational expression. The procedure for applying entry editing has four distinct steps:

1. Invoke editing on a prior entry by an initial ATTN (before anything else is keyed) in execution mode. The entry that is edited depends on the state of the system at the moment of invocation. Note that invocation can result also from specific commands in definition mode, and from the system function "⎕*LC*".

2. If necessary, the text of the selected entry is displayed.

3. Type a line of characters specifying the desired editing below the displayed characters (spacing the cursor accordingly):

   ",T" The first comma causes text T thereafter to be inserted before the character above the "," and subsequent " ", "/", "." and "," to be treated as part of that text.

   Prior to the first comma:

   " " Each space preserves the character above it.

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

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

If the original text is longer than one display width, a terminal ATTN will allow continued specification on the next portion. Complete the specification by a RETN.

4. Revise the text. The system and the user take turns cooperatively creating a new entry. The first phrase is displayed excluding the specified deletions. The user enters the addition, concluding with a terminal ATTN. The next phrase is then displayed and the user makes more additions, etc.

   During an addition to a phrase, if the addition nearly completes the current display line, then a terminal ATTN may cause a continuation indicator to appear, allowing the user to continue the addition. In this case, an initial ATTN will bring the next phrase (if any), rather than give the destroy warning indicator.

Any character entered but not recognized by APL/700 results in a "CHARACTER ERROR" report. The positions of invalid characters are indicated by both a line with "*" markers and the prior entry redisplayed with good parts preserved and invalid characters replaced by the squish-quad "⌷" display character. Then the cursor exdents to accept editing specifications. An example of this and subsequent editing that also includes revision is:

```
     'THIS IS A BAD LINE.'          invalid characters
*** CHARACTER ERROR ***             error message
         **    *                    position markers
     'THIS ⌷⌷ A B⌷D LINE.'          display with invalids marked ⌷
          //.  ///,GOOD             enter edit specification
     'THIS IS NOW A GOOD LINE.'     ATTN after typing "IS NOW"
                        v           backspace to "L", ATTN
                        ENTRY.'     completed entry - RETN
THIS IS NOW A GOOD ENTRY.           display of entry (no quotes)
```

If RETN is pressed during the above sequence while one or more of the phrases delimited by the "."s in the immediate edit line have not been displayed, those phrases (and possibly including ",T" insertion) are lost. RETN completes the entry.

Entering any character other than a space, slash or dot before a comma in the specification(step 3) results in the following error message:

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

Pressing ATTN reinitiates the transaction editing sequence.

The use of ATTN for making typing corrections within an entry does not conflict with the ATTN uses described above. That is, for embedded ATTN, the cursor has been backed-up at least one position from the right-most (uncorrected) extent, and so is embedded, not at the right-most entry position when ATTN is pressed.

An ATTN can be used to interrupt the display of a text line (step 2). The result is a cursor return to the left margin, ready for step 3. An abbreviated display may conclude with ellipsis "...".

Ellipsis also appear when a window of an entry is being displayed, indicating that more text exists at the start and/or end of the window, for example:

... IS THE MIDDLE OF SOME TEXT WITH ...

The last portion of a long entry containing a character error is the only one preceded by a line with error positions noted by "*". All erroneous positions are shown with the squish-quad.

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 line with "*", then the entered response only (excluding the prompt) and including the squish-quads, and exdent to the left margin ready for edit specification. Entry of RETN there abandons that response and repeats the prompt.

An ATTN can also be used to discard unwanted output from or interrupt execution of an executing function (see Section 8).

RECALL OF PRIOR ENTRIES

Up to four prior entries may be recalled for editing or reentry. One (or possibly two) of these apply for each environment established by a suspended function. A return to a prior environment restores the recall entries of that environment.

Editing is automatically invoked starting at step 3 upon any of:

keyword or character error in any entry (or portion thereof)
syntax errors while in definition mode
any errors while in evaluated input mode

Otherwise, the specific prior entry in response to an initial ATTN is the highest precedence prior retained entry. When a particular entry is completed, its image is injected at a precedence level, and all entries of higher precedence are cleared. At initiation of a particular key-in environment (for example, execution mode) all precedence levels are cleared. Then, possibly one level is injected. This also applies to re-initiation of a prior key-in environment, except that precedence levels lower than the injected level are restored to their former state.

The rules for injecting after terminating an execution mode entry are:

Precedence Level	Kind of Entry Just Processed
2	system command with erroneous or exception response, or unsuccessful attempt to enter definition mode
1	incompletely processed APL expression
0	completely processed APL expression

An empty specification entry in response to an invocation of an edit aborts that edit. An empty entry itself clears the highest precedence level unless 0 is the highest level.

Examples

```
     )CLEAR                    start off clean
CLEAR WS
     3×4                       enter an expression
12                             result
                               initial ATTN to edit expression
     3×4                       expression back for edit
     /.                        editing specifications
     5×4                       new expression
20                             new result
     )SYSTEM COMMAND ERROR     entry by the user
*** SYNTAX ERROR ***
                               initial ATTN by the user
     )SYSTEM COMMAND ERROR     recalled system command
      ////////////////////     editing specifications
     )KEYWORDS                 good system command
ARE OFF                        answer
                               initial ATTN by the user
     5×4                       recalled expression
     /,7                       editing specifications
     7×4                       new expression
28                             new result
     ∇A B C D                  attempt to define function
*** DEFINITION ERROR ***
                               initial ATTN by the user
     ∇A B C D                  recalled entry
       /,←                     editing specifications
     ∇A←B C D                  revised entry
[1]   A←B+E∇                   define the function
                               initial ATTN by the user
     7×4                       recalled expression
                               RETN by the user abandons edit
     X+2 C 3                   another entry by the user
*** VALUE ERROR ***
C[1]* A←B+E                    this line was not completed
          ^                    note suspended function
                               initial ATTN by the user
     A←B+E                     recalled incomplete expression
     .   /                     editing specifications
     ∇C[1]A←B+D∇               insert in function and correct
     2+4                       another entry by the user
6                              and its result
                               initial ATTN by the user
     2+4                       recalled expression
                               RETN by the user abandons edit
     →                         pop the state indicator
                               initial ATTN by the user
     X+2 C 3                   recalled previous expression
           v                   initial ATTN no editing specifications
                               RETN re-enters line
*** VALUE ERROR ***            different error this time
     v
     X+2 C 3
     /.                        initial ATTN then editing specifications
     5+2 C 3                   RETN re-executes line
10                             answer
```

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
FRIDAY   28 JANUARY 1977  8:15 AM
TIMES:    SESSION      TO DATE
CONNECT   0.14.48     17.50.25
CPU       0.01.07      0.09.15
IO        0.00.02      0.01.51
```

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

   Another MCS can be entered. Start with step 6c of the sign-on actions.

   Otherwise, for a dialup connection, remove the telephone handset from the acoustic coupler and return it to the telephone cradle.

5. Turn off terminal and coupler power as required.

RECOVERY OPERATIONS

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

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

```
      )ON DIANE[JOY]
A P L / 7 0 0   VERSION 28.000
FRIDAY  28 JANUARY 1977  8:20 AM


WS MYWORK CONTINUED FROM 77/01/25 16.28.13.
```

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 next suspension point; 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 displays an input prompt and waits for an execution mode entry. If an input request is encountered before the suspension, processing continued normally as if there had been no disconnection.

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 occured while an entry was being composed but before the RETN results in ignoring that non-entry.

If the keyboard was unlocked awaiting input in response to an input communicator prompt, the word *INPUT* appears before the *CONTINUED*.

If the continued active workspace had not been named, the *WS* Name is omitted. If the active workspace was loaded from another account, the owning account name in parentheses precedes the workspace name.

Recovery does not reestablish shared variable offers.

If a system malfunction occurred in the middle of some file action, warning is given.

SECTION 3

SYSTEM COMMANDS

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 parts; and the 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 Usercode [Password] Chargecode	signs on account
)COFF [Oldpassword/Newpassword]	signs off to continue
)OFF [Oldpassword/Newpassword]	signs off

)ON logs the account named by Usercode onto 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 continuation at the next)ON for that account.

)OFF logs the account off and discards the active workspace, so that at the 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, IO (input output) time, and elapsed time used. These amounts are given both for the session and for the installation accounting period. Units are hours, minutes, and seconds.

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

The optional Password allows protection of a usercode from unauthorized use. The Password can be initially set by the installation, or by the user at any sign-off. Once set, the proper (non-empty) Password must be used for any successful sign-on. The forms for adjusting the Password at sign-off are:

[/Newpassword]	establishes Password
[Oldpassword/Newpassword]	changes Password

The Usercode and Password are common to APL and the installation. A Password set in APL applies for that Usercode when used outside of APL. Likewise a Password set or changed outside of APL applies in APL.

Usercode and Chargecode may have 1 to 17 characters. A Password is empty, or once set has 1 to 17 characters. These characters are alphanumeric (excluding the APL underscore alphabet).

)ON DOREEN	initially no Password
)COFF[/SESAME]	add Password
)ON DOREEN[SESAME]	now must use Password
)OFF [SESAME/NEWKEY3]	change Password

The optional Chargecode may be used for installation accounting.

)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 Usercode. 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 250. 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. This command may be entered before)ON.

```
     )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
DEFAULT IS 256
      )SYMS 400
DEFAULT 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
DEFAULT IS 1
      )ORIGIN 0
DEFAULT 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
DEFAULT IS 10
     )DIGITS 4
DEFAULT WAS 10
```

)SEED n default random number seed

The starting value for 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
DEFAULT IS 0
     )SEED 37752963
DEFAULT 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 $0 \leq n < 1$. The default fuzz may be overridden by the system variable ⎕*CT*, comparison tolerance. See that description for details.

```
     )FUZZ
DEFAULT IS 1E¯10
     )FUZZ 0.1
DEFAULT 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 the primitive APL file 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
```

)LIB Usercode

The identifiers of workspaces in the indicated account library that have been public saved are displayed (but not their Passwords).

```
      )LIB ACTG
ACCOUNTING
REPORTER
```

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

Workspacename is (Usercode) Wi [Password]

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

The Usercode 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 formed from 1 to 12 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 Wi of the loaded workspace, including (Usercode) but not the Password, becomes the name of the active workspace. If Workspacename is absent, then the library copy of the active workspace is reloaded.

```
)LOAD TEXTEDIT                          own account
)LOAD (JOANNE)TENDS[KOLOHE]             from account JOANNE
```

)COPY Workspacename Namelist — replace copy

Copy into the active workspace from the library workspace identified by Workspacename. If Namelist is absent, then copy all functions, variables and groups (not the state indicator, environment, character prompt, or latent expression). If Namelist is present, then copy only the items attached to names in it that are present in that workspace. Monitors are reset on copied functions. Copied items replace like-named prior items in the active workspace.

```
)COPY OLDWORK                           entire workspace
)COPY (ACTG)REPORTER[KEY17]TOTALS FORECASTGRP 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 (ACTG)ACCOUNTING SCHEDULE PERIODS
NOT SCHEDULE
```

Locked functions obtained from another account are sealed. The user cannot unlock them.

)LOAD should be used where adequate.)COPY and)PCOPY are more complex commands requiring more resources than)LOAD.

```
)SAVE Wi [Password]                     save as new workspace
)SAVE [Currentpassword]                 replace library version
)SAVE [Oldpassword/Newpassword]         change existing password
```

A copy of the active workspace can be saved in the account library of the user. If Wi (not currently in the user's library) is present, that name is the one used for subsequent library reference. If Wi is absent, the prior active workspace identifier is used. A workspace loaded with a Password must include it as part of the)SAVE. Newpassword may be empty, removing the Password.

```
)SAVE WORK
)SAVE MY[LOCK]
)SAVE [/VERSION]
```

)DROP
)ACCESS
)CLEARACCESS

)DROP Wi [Password] drop workspace from account library

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.

)DROP NEW[VERSION]

The normal response from the)LOAD,)COPY,)PCOPY,)SAVE and)DROP commands is the time stamp when the workspace was most recently saved. The account and workspace names are appended for)SAVE.

A set of permissions for access to a workspace by any user are contained in the workspace and are controlled by the workspace owner.

)ACCESS Accesses query/alter active workspace accesses

)CLEARACCESS Accesses query/alter clear workspace access defaults

If no Accesses are specified, the current access condition is displayed. Otherwise, the specified Accesses (separated by spaces) are changed. Other specified Accesses are retained. Accesses may include at most one member from each of the following 5 subsets.

1. *NORMAL* any user knowing the workspace name may use it, subject to its internal access set
   PUBLIC same as NORMAL, plus the workspace name appears in the response to)LIB Usercode
   OWN only the owner may access the workspace

2. *LOAD* the workspace may (not) be loaded by any account
   NOLOAD

3. *COPY* the workspace or names therein may (not) be copied
   NOCOPY by any account

4. *SAVE* the workspace may (not) be saved by any account
   NOSAVE after being loaded

5. *REPLACE* the account library version of the workspace may
   NOREPLACE (not) be replaced by the owner with a)SAVE

The first member of each subset is the "normal" default value. The set of permissions existing when a workspace is saved apply to control subsequent accesses to that workspace. The Accesses apply to all users including the owner. Only the owner can override them (assuming that either LOAD or COPY is permitted). For example, if NOSAVE is set in the library version of the active workspace, the owner must repeat)ACCESS NOSAVE or)ACCESS SAVE before a)SAVE.

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 (with collating sequence A ... Z 0 ... 9 A ... Z) 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
W  Z
      )GRPS
ANALYZE  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 discussions in Group Commands following, and the system function ⎕*EX*, expunge.

```
      )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

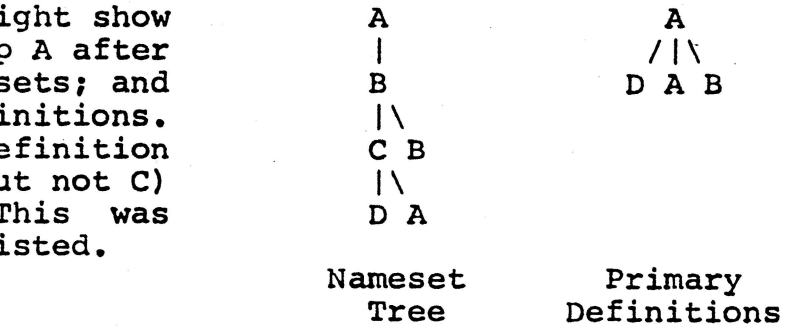
retaining the Groupname on its second occurrence

ignoring any additional occurrences

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
      A THIS WAS FOR THE SECOND OCCURRENCE
      )GRPS
A  B  C
      )ERASE A
NOT D
      A 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.



```
)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 pending functions (awaiting completion of functions above) and earlier suspended functions.

The state indicator may also include suspensions with evaluate functions or evaluated input requests that are incompleted. In each such case, the appropriate symbol "⍎" or "⍞" appears prior to (below) the function line causing suspension. The Load Expression "⍞*LX*" may also appear as a "⍎".

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

```
      )SI
RUN[1]*                    most recent suspension
⍎                          evaluate included "RUN"
MAIN[5]                    pending evaluate result
RUN[4]*                    prior suspension
```

A function can appear more than once in the state indicator (the function may be restarted, or may call itself recursively). Usually the state indicator should be emptied of unnecessary entries, as space is consumed, and global names may be shielded by local names.

The suspended function at the top of the state indicator may be restarted by entering "→L" where L is a line number. The suspended function and any pending its completion may be aborted by entering "→". Response is a line showing the next suspended function if any.

```
      →                    remove top 3 levels
RUN[4]*                    next prior suspension
      )SI
RUN[4]*                    only prior suspension
```

)RESET — state indicator reset

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

```
)RESET                     removes all levels
)SI                        empty
```

MESSAGES

A message system allows single-line messages to be sent by a user.

)TO Usercode Text	send Text to Usercode
)OPR Text	send Text to operator position

The intended action is to display the Text on the terminal of the addressee. The normal reception condition for a terminal allows a received message to be displayed before any execution mode prompt (5 space indent). A user may restrict message reception.

)RECEPTION ABSENT	no messages are accepted
)RECEPTION NORMAL	messages are accepted only before the execution mode prompt; not before character- or evaluated-input requests, nor in function definition or execution.
)RECEPTION URGENT	messages are accepted any time except within any single logical output.

A message sender is notified by RECEPTION ABSENT, NOT SIGNED ON, SIGNED OFF, or DISCONNECTED if a message cannot be delivered. Otherwise after initiatiating a message that can not be immediately delivered, a "jiggle" is given and the sender's terminal waits for the addressee's terminal to reach an acceptable condition for message receipt. This wait is just before the execution mode prompt. While waiting the sender may regain use of the keyboard by entering ATTN, at the expense of aborting the message before delivery.

When the message delivery has started at the addressee's terminal, a RECEIVED message is displayed and the sender's terminal receives the execution mode prompt, ready for another entry. An ATTN may interrupt a message being displayed; the rest of it is lost.

At each sign-on the default status is)RECEPTION NORMAL. This prevents messages from intruding during function definition mode or function evaluation.

The System operator (OPR) may direct a message to one user, or may make a public announcement to all APL users presently signed-on. In the first case, the message is treated as a user message. In the second case, that message is retained for those users currently signed-on but not receiving messages. This message will generally be delivered just before each user next enters execution mode. A user with)RECEPTION ABSENT will only see the message after doing a)RECEPTION NORMAL or)RECEPTION URGENT. Only one such OPR message is retained. Thus a second message from OPR may replace the first (without the user ever receiving it).

)LOCK
)USE
?MCS

)LOCK ON — Pause before user phase of transaction cycle to await message reception

)LOCK OFF — Normal transaction cycle, message reception when allowed by reception condition

Unless)RECEPTION URGENT is set, a message from another user or the operator may be received only prior to an execution mode prompt (allowing an entry from the keyboard). Since the time for the addressee to make the next entry is unknown, the message sender may wait a long time before the message is received. After sending a message, or with lock ON, a jiggle is displayed leaving the cursor indented 3 positions to indicate the expected message, and the keyboard is locked. While awaiting a message with reception lock ON, an ATTN will defeat that lock to allow one transaction entry.

If a message is expected and the reception lock is OFF, once the user phase has started, an entire execution mode transaction must be completed before that message is received. As a courtesy to the sender, with the reception lock ON, the message will appear as soon as possible, since the receiver's transaction cycle pauses before entering the user phase.

OTHER MESSAGE CONTROL SYSTEMS

The APL user may be transfer to other Message Control Systems.

)USL — determine the available Message Control Systems

)USL MCSname — transfer control to that Message Control System

For example, to transfer to CANDE:

)USE CANDE

The prefix "*SYSTEM/*" and the family suffix, if there is one, may be omitted. If a prefix or suffix is omitted and this results in an ambiguity, the transfer will not be done.

Of course, the named MCS may impose additional use restrictions. It is beyond the scope of APL to identify such restrictions, and even the method to return to APL from such a system. As a possible example, if the terminal was last used in SYSTEM/CANDE, return to the APL MCS may be achieved by entry at the left margin:

?MCS SYSTEM/APL — change to APL MCS

A family name suffix may be required such as "ON SYSTEMPACK".

KEYWORDS AND ASCII CHARACTERS

)KEYWORDS ON	Names enclosed in "< " and "> " are surrogates for certain APL characters
)KEYWORDS OFF	APL characters are used for input and display

Keywords permit APL text to be entered, displayed and edited using an ASCII character set consisting of at least the following visible characters:

Letters	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
Digits	0 1 2 3 4 5 6 7 8 9
Other	. : , ; ! ? + - * = / \ ' " $ ( ) [ ] < >

All other defined APL symbols are individually recognized by their keywords preceded by "< " and followed by "> ". The graphics, their common names, and their corresponding keywords surrogates are included in Table 2-1.

If keywords are ON, any entry is acceptable using either the keyword or its single character equivalent, except for the angle brackets, "< " and "> " which are reserved to bracket the keywords. The keywords "<*LT*>" and "<*GT*>" are used in their place. Any display appears using the keyword surrogates.

Use of the "character set in common" (those for which no keywords are needed) avoids display problems when switching between the alternative keyword settings.

The entire set of ASCII visible characters is either available in APL or synthesized by overstrikes. In addition to the above characters, other ASCII characters and their APL surrogate print images are shown below.

ASCII HEX	22	23	24	25	26	27	40	5E	5F	60	61--6F	70--7A	7B	7C	7D	7E
ASCII-68	"	#	$	%	&	'	@	^	_	`	a...o	p...z	{	¶	}	~
APL	¨	⍎	$	⍀	⍁	'	⍟	⍓	_	⍤	A...Q	P...Z	⍔	⍵	⍕	~
overstrike:		H	S	/	7		0	¨		¨	A...O	P...Z	[	⊂	]	
with:		=	|	:	\		⍺	^		'	_...._	_...._	°	∩	°	

An "ASCII" terminal may have some variations in the above graphics: The paragraph mark "¶" may appear as a broken vertical bar "¦", the underscore "_" may appear as a left arrow "←", the curly brackets may appear as the APL overstrikes, etc.

)KEYWORDS ON uses the normal edit conventions for editing both the immediate expression and a user-defined function line.

```
     )KEYWORDS OFF
WERE OFF
     SORTUP←UNORDERED[⍋UNORDERED]                              normal form
     )KEYWORDS ON
WERE OFF                                                       ATTN by user
     <S><O><R><T><U><P><IS>UNORDERED[<GRUP>UNORDERED]          keyword form
                 //////.                    //,DN             edit specs
     <S><O><R><T><D><N><IS>UNORDERED[<GRDN>UNORDERED]          enter <D><N>
     )KEYWORDS OFF
WERE ON                                                        ATTN by user
     SORTDN←UNORDERED[⍒UNORDERED]                              normal form
```

The keyword setting is a property of the terminal session, rather than of the workspace. It is initially OFF when a session begins. One consequence of this is that recovery from an implicit)COFF caused by a disconnect will not recover the state of)KEYWORDS.

If keywords are required by a specific terminal, the user should set)KEYWORDS ON prior to sign-on with the)ON command. This will insure that all necessary input characters are available. An example where all necessary characters might not otherwise be available is when the user has been involuntarily disconnected while in function definition mode. Later, when the user signs-on again, if)KEYWORDS were OFF, the user will be in function definition mode without the capability of entering the "∇" which would exit function definition mode. While in function definition mode it is not possible to enter system commands, therefore the user can not turn)KEYWORDS ON.

SECTION 4

THE APL/700 LANGUAGE

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 rank, shape, and value.

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 of its type.

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

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 Form	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 (including a decimal 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 12.5 .00 4.        fixed point entry
0.3125  ¯0.2  12.5  0  4           display form
```

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

	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:

Subtype	Domain
Boolean	0 (*FALSE*) *AND* 1 (*TRUE*)
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 one element data object with 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. Representations are:

```
character     ''
numeric       ⍳0
Boolean       0⍴0
```

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=ρ,D`
Empty	`0=ρ,D`
Character type	`' '=0\0ρD`
String of characters	`(' '=0\0ρD)∧2>ρρD`
Numeric type	`0=0\0ρD`
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 of expressions. 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 usercode. Three classes of names exist, differing in maximum length and characters. Names have no internal spaces.

variable, function, argument, result, local name, label, group
69 characters maximum
first character from *A...Z* *Δ...Z* Δ Δ
remaining characters from *A...Z* *Δ...Z* Δ Δ _ 0...9

usercode or its password
17 characters maximum from *A...Z* 0...9

workspace or file identifier or their passwords
12 characters maximum from *A...Z* 0...9 starting with a letter

Variable: 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 ⎕*EX* 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. A shared variable is one shared with another process.

User defined function: The function name of a processing algorithm thereby defined, used to execute it.

Argument: A name in the definition of a function representing a value supplied at the time of function execution.

Result: The name in the definition of a function representing the value returned after function execution.

Local name: A name used internal to a defined function.

Label: A named local constant having value the number of the line on which it occurs in a defined function.

Group name: 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, or group name. A group may also serve as documentation.

Usercode: Each user has a name supplied by the system and used for sign-on accounting purposes, synonymous with account name.

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

File name: The name by which a file created or referenced by a user is identified. It includes usercode (if owned by another usercode), its name, and its password (if locked).

Password: Each usercode, 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.

Function valence is the number of arguments to a function. Thus, a monadic (valence 1) function is elaborated when the value of its (right) argument is determined. A dyadic (valence 2) function is elaborated when both of its arguments (left and right) are determined. An argument can itself be an expression. A niladic (valence 0) user-defined function is elaborated when (and if) 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 value attached to that right argument name, the right argument becomes that new value. See Appendix D.

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 primitive function is monadic or dyadic: a function is monadic unless it has an argument to its left or is used a part of an operator (see the primitive operators in Section 5).

```
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 either is 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 contexts in which the exact number of blanks is preserved as significant are in literal character strings or comments. In a defined function, extraneous blanks are immediately eliminated after syntactically correct line entry. Extra matching pairs of parentheses in an entered expression may help to clarify it and do no harm. However, extraneous parentheses in a defined function line are removed from display of that function once the line has been elaborated.

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 inject 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 invalid 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 invalid 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
```

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, Fn 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 (vector) entry is accepted as input that replaces prompt positions by blanks. The result is a vector.

Explicit output: Assignment to the pseudo-variable ⎕←*E* causes display of the value. No time penalty is paid for use, so inclusion is recommended for readability. Each such assignment causes display of the expression 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 (containing any characters from ⎕*AV*) 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 the ⍞ 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 a fixed number of blanks are inserted between one element and the next, and folded lines are indented. Numeric array output is displayed with fixed width columns.

Failure to enter a value-producing expression for evaluated input results in another □: prompt. Termination from evaluated input, (and from the function in which it is included and any in its function call environment) is achieved by the niladic terminate entry: '→'.

Interrupt from either character input or evaluated input is by entering the letters O U T superimposed.

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

Note that combinations are meaningful:

⍞ ← ⍞	Request character input to establish a new character input prompt
□ ← ⍞	Display prompt, accept input and echo it back blanking the prompt
□ ← □	Accept and evaluate 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
      ⍞←'OUTPUT FILE ______ CHARACTERS ______',⎕M,12ρ' '
      X←⍞          ⍝ DISPLAY PROMPT FOR CHARACTER INPUT
OUTPUT FILE PETE__ CHARACTERS ⎕⍬⍬_⍙
      X            ⍝ EXCLUDES PROMPT IN X
              PETE                 ⎕⍬⍬ ⍙
                   ⍝ NOTE USERS INPUT IS POSITIONALLY
                   ⍝ CORRECT WITH RESPECT TO THE PROMPT

                   ⍝ MIXED TYPE OUTPUT
      'RANK=';ρρX;' SHAPE=';ρX;' VALUE=';X←2 3ρ⍳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

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 and assignment functions
Scalar functions
Operators
Mixed functions
Format functions
System functions and Shared variable functions (see Section 6)
File Functions (see Section 7)

The primitive functions and operators are represented by single APL characters. The same character is often used to represent both a monadic function (valence 1, having only right argument) and a related dyadic function (valence 2, having both right and left arguments). 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 vector or array 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.

Primitive functions have no hidden side-effects on the workspace state.

SELECTION AND ASSIGNMENT

Selection allows accessing elements of an array by indexing. Assignment gives value to or alters the value of a variable or indexed elements of it.

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 $I \leftrightarrow \iota(\rho V)[I]$. 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 attendant 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
      ⍝ ELEMENT VALUES SUGGEST
      ⍝ THE INDEX POSITION,
      ⍝ I.E., 12 ↔ A[1;2]
      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[;]        ⍝ ALL
11  12  13  14
21  22  23  24
31  32  33  34
```

```
      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
      V[]
ABCDE
```

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 $M \leftarrow M \circledast E$
A[L] ⍟← E	Modified Insert, short for $A[L] \leftarrow A[L] \circledast E$

Where

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

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

E is the assignor: result of evaluating an expression

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

L is an index list valid for A (origin sensitive)

⍟ is any scalar dyadic primitive function

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

Results

Assignment functions specify or alter the value associated with 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: $M \circledast E$.

Modified Insert: The result is the value inserted: $A[L] \circledast E$.

Conditions

Replace: N is specified to have the value and all attributes of E, 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 and the types of M and E 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. Any side-effects of elaborating L are only done once, rather than twice in the longer form.

For Insert or Modified Insert, if any element selected 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←X[]←Y←1        ⍝ MULTIPLE REPLACEMENTS, INSERT TO ALL X
      Y
1
      Z                ⍝ THE EXPRESSION VALUE, NOT X
1
      X                ⍝ 1 WAS COERCED TO ALL ELEMENTS
1  1  1
      ⎕←X[2 3]←2 3     ⍝ INSERT, DISPLAY EXPRESSION
2  3
      ⎕←X×←3           ⍝ MODIFY ALL ELEMENTS OF X, ↔ X←X×3
3  6  9
      ⎕←X[1 2]-←2 1    ⍝ MODIFIED INSERT ↔ X[1 2]←X[1 2]-2 1
1  5
      X
1  5  9
      A                ⍝ EXISTING ARRAY NAMED A WITH SHAPE 2 3
1  2  3
4  5  6
      ⎕←A[2;]←7 8 9    ⍝ INSERT INTO ROW 2 AND DISPLAY
7  8  9
      ⎕←A[;3]←4        ⍝ COERCE AND INSERT TO COLUMN 3 AND DISPLAY
4
      A
1  2  4
7  8  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.

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.

Conditions

The comparison tolerance affects the determination of whether or not B is an integer for floor and ceiling. In general, the numbers resulting from computation are only approximately precise, since they are represented by one of a large but still restricted set of values. APL classifies as integers, numbers that are effectively integers. See the discussion of comparison tolerance in Section 6.

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
      0⌈¯3 4 2.1⍴ 0 COERCED TO 0 0 0
0 4 2.1
```

Forms

+ B	Identity
- B	Negate
× B	Signum
÷ B	Reciprocal
| 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. $+B \leftrightarrow 0+B$

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

Signum: Return the integers ¯1, 0, 1 if B is negative, zero or positive. $\times B \leftrightarrow (B>0)-B<0$

Reciprocal: Return the reciprocal of B for non-zero B. $\div B \leftrightarrow 1\div B$

Magnitude: Return the absolute value of B (a non-negative number). $|B \leftrightarrow B\times\times B$

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 $B \leftrightarrow R+G\times A$.

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.

Identity is the additive identity. It sometimes is referred to as conjugate, which is the same thing for real numbers. 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 4, and unexpected results in Appendix D.3.

Examples

```
      +7.2
7.2
      +0 5 ¯10 15
0  5  ¯10  15
      -¯7.3  1.2E3
7.3  ¯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
      1.5E14 + 2.1E15
2.25E15
      1+0 5 ¯10 15
1  6  ¯9  16
      ⌊0.5+2.4 2.5 2.6⍝ ROUND
2  3  3
      175-225
¯50
      5-0 5 ¯10 15
5  0  15  ¯10
      3 1 ¯4×¯5 2 ¯3
¯15  2  12
      5 0 ¯5××5 0 ¯5
5  0  5
      5 ¯12 ¯15÷4
1.25  ¯3  ¯3.75
      3 3 ¯3 ¯3|4 ¯4 4 ¯4
1  2  ¯2  ¯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 B is 0 the result is 1. For positive B, the result is 0.

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 integer 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 ¯32*÷3 3 5 2.5
¯2 ¯3 ¯2  4
      0*0 0.001 1 10
1  0  0  0
```

```
      ⍟*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
```

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 3⍴1 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. The logical functions are also called Boolean functions, as are membership, subset and superset.

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 ⍝ LOGICAL IMPLICATION, A IMPLIES B
A=B ↔ ((~A)∧~B)∨A∧B ⍝ LOGICAL EQUIVALENCE

Examples

```
      ~1 0
0 1

      0 0 1 1 ∧ 0 1 0 1
0 0 0 1

      0 0 1 1 > 0 1 0 1
0 0 1 0

      0 0 1 1 < 0 1 0 1
0 1 0 0

      0 0 1 1 ⍱ 0 1 0 1
1 0 0 0

      0 0 1 1 = 0 1 0 1
1 0 0 1

      0 0 1 1 ⍲ 0 1 0 1
1 1 1 0

      0 0 1 1 ≤ 0 1 0 1
1 1 0 1

      0 0 1 1 ≥ 0 1 0 1
1 0 1 1

      0 0 1 1 ∨ 0 1 0 1
0 1 1 1

      0 0 1 1 ≠ 0 1 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+Y*2)*0.5`	`¯4○N ↔ (¯1+N*2)*0.5`	`1≤|N`	`0≤X`
`5○N ↔ sinh Y`	`¯5○N ↔ arcsinh N`		
`6○N ↔ cosh Y`	`¯6○N ↔ arccosh N`	`1≤N`	`0≤X`
`7○N ↔ tanh Y`	`¯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
Y is numeric

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 roots. 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

```
      o1 2 ¯3  ⍝ MULTIPLES OF PI
3.14159  6.28319  ¯9.42478
      1 2 3000.5 0 0.25  ⍝ SIN 90°, COS 0°, TAN 45°
1  1  1
      4 0 ¯40 0 0.8 1 ⍝ SQUARE ROOT FUNCTIONS
1  0.6  0
      5 6 700  ⍝ SINH, COSH, TANH
0  1  0
      ¯1 ¯2 ¯301 ⍝ ARCSIN, ARCCOS, ARCTAN IN RADIANS
1.5708  0  0.785398
      (¯1 ¯2 ¯301)×180÷o1 ⍝ 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 $B\times!B-1 \leftrightarrow !B$ where $1 \leftrightarrow !0$ (alternatively $\times/\iota B \leftrightarrow !B$ in origin one).

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

Gamma (B+1) $\leftrightarrow !B$

Factorial for negative integer B is a domain error. The comparison tolerance affects this integer determination. The number limit occurs for B>51.

Combinatorial: The result is $(!B)\div(!A)\times(!B-A)$ when all the indicated factorials are defined, even if beyond the number limit; e.g. 116!232.

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) $\leftrightarrow \div B\times(A-1)!A+B-1$
$\leftrightarrow (!A-1)\times(!B-1)\div!A+B-1$

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
      !¯3.9 ¯2.9 ¯1.9 ¯0.9 0.1 1.1 2.1
¯1.91843  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 $\circ B$ where $\circ$ 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 $\circ$ to the corresponding element of B the argument.

Dyadic $A \circledast B$ where $\circledast$ 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 $\circledast$ 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 $\circledast$ 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.

$A \circledast [K] B$ qualified application of $\circledast$ along dimension K

If either 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 or the arrays have the same rank and differ in shape only in dimension K where one of the arrays has length 1, then the result has the same rank and shape as the larger. Elements of the result are formed after first effectively coercing the smaller rank (or array with length 1 dimension K) array to have the same shape as the other array. This coercion is by replication of the entire smaller rank array (or array with length 1 dimension K) as a plane across each position along 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 or shape array. Let $I \leftarrow K \neq \iota \rho \rho A$ Then the coercion condition may be expressed as:

$(I/\rho A) \leftrightarrow \rho B$ if ranks differ, or $(I/\rho A) \leftrightarrow I/\rho B$ if ranks match

For J each scalar value in $\iota(\rho A)[K]$ the plane of the result R so determined is:

$R[\ldots;J;\ldots] \leftrightarrow A[\ldots;J;\ldots] \circledast B$ or $A[\ldots;J;\ldots] \circledast B[\ldots;\square IO;\ldots]$

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 may include a dimension selector appearing to the right of the function character that indicates in brackets the index number or dimension of function application. The index number is a single and is origin sensitive. If no dimension selector is used, function application is along last dimension (or first with the bar forms). The dimension selector is sometimes referred to as "axis operator", since it specifies the axis or dimension of application of the function. It differs from the operators here discussed in that it is not one of the dyadic scalar functions.

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 $(\rho\rho A)+\rho\rho B$ and shape $(\rho A),\rho B$ 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, the two operators sometimes can be combined into an inner product. 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
      5 4 3 2 1∘.=1 3 4 2 5 3 2 1   ⍝ NUMERIC COMPARE
0  0  0  0  1  0  0  0
0  0  1  0  0  0  0  0
0  1  0  0  0  1  0  0
0  0  0  1  0  0  1  0
1  0  0  0  0  0  0  1
      ' +'[1+5 4 3 2 1∘.=1 3 4 2 5 3 2 1]   ⍝ GRAPH
     +
   +
  +   +
    +  +
+       +
      ' *'[1+5 4 3 2 1∘.≤1 3 4 2 5 3 2 1]   ⍝ 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○.00÷6 3 2 1   ⍝ SIN COS 30 60 90 180 DEGREES
 5.0000E¯1     8.6603E¯1     1.0000E0     ¯5.1267E¯12
 8.6603E¯1     5.0000E¯1     2.5633E¯12   ¯1.0000E0
```

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); $K \in \iota \rho \rho A$
⊛ 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 $(K \neq \iota \rho \rho A) / \rho A$.

For scalar A, the result is A.

For nonempty 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 consecutive elements of the vector and executing the resulting expression. For empty vector A, the result is the identity element for ⊛ if it exists.

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 ⍝ 1+2+3
6

      ⍝ MATRIX
      A23
1  2  3
4  5  6
      +/A23 ⍝ SUM OF ROWS
6  15
      +/[2]A23 ⍝ LONG FORM
6  15
      +/[1]A23 ⍝ OF COL'S.
5  7  9
      +⌿A23 ⍝ OF 1ST DIM'N.
5  7  9

      ⍝ HIGHER RANK ARRAY
      B223
 1   2   3
 4   5   6

 7   8   9
10  11  12
      +/B223 ⍝ ROW SUM
 6  15
24  33
      +/[2]B223 ⍝ OF COL'S.
 5   7   9
17  19  21
      +/[1]B223 ⍝ OF PLANES
 8  10  12
14  16  18

      ⍝ SCALAR
      +/2
2

      ⍝ EXTREMUM
      ⌈/3 ¯7 0 ⍝ MAXIMUM
3
      ⌊/3 ¯7 0 ⍝ MINIMUM
¯7
```

```
      ⍝ ALTERNATING SUM
      -/1 2 3 ⍝ 1-2-3)
2
      -/1 2 3 4
¯2

      ×/1 2 3 ⍝ PRODUCT
6

      ⍝ ALTERNATING PRODUCT
      ÷/1 2 3 4 ⍝ (1×3)÷(2×4)
0.375

      ⍝ WITHIN RANGE (3,6]
      >/3 6<5
1

      ⍝ RELATIONAL ANOMOLY
      ≤/1 3 5 ⍝ LEFTMOST: 1≤1
1
      ≤/2 4 6 ⍝ LEFTMOST: 2≤1
0

      ⍝ PARITY
      =/0 1 0 ⍝ 0 PARITY
1
      =/1 0 1 ⍝ EVEN NO OF 0'S
0
      ≠/1 1 0 ⍝ 1 PARITY
0
      ≠/0 1 0 ⍝ ODD NO OF 1'S
1

      ⍝ LOGIC CIRCUITS
      ∧/1 1 1 ⍝ AND
1
      ∧/0 1 1
0
      ∨/0 0 0 ⍝ OR
0
      ∨/0 1 1 ⍝ LEFTMOST: 0∨1
1
      ~∨/0 1 1 ⍝ IN CONTRAST
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, $K \in \iota\rho\rho A$
⊕ 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 $R[I] \leftrightarrow \oplus/I\uparrow A$ (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
      A23 ⍝ARRAY
1  2  3
4  5  6
      +\A23 ⍝ OF LAST DIMENSION
 1   3   6
 4   9  15
      +⌿A23 ⍝ OF FIRST DIMENSION
1  2  3
5  7  9
      +\[1]A23 ⍝ LONG FORM
1  2  3
5  7  9
```

Examples (continued)

```
      +\2 5 1 ¯2          ⍝ CUMULATIVE SUMS
2  7  8  6
      -\3 5 2             ⍝ 3, (3-5), 3-5-2
3  ¯2  0
      -\1 2 3 4 5 6       ⍝ DIFFERENCES
1  ¯1  2  ¯2  3  ¯3
      ×\1 2 3 4 5 6       ⍝ FACTORIALS ↔ !1 2 3 4 5 6
1  2  6  24  120  720
      ÷\1 2 3 4 5 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
      =\0 0 1 0 1         ⍝ 0 PARITY, CHANGES EACH 0
0  1  1  0  0
      ≠\1 1 0 0 1         ⍝ 1 PARITY, CHANGES EACH 1
1  0  0  0  1
      ∧\1 1 0 0 1         ⍝ LEADING ONES
1  1  0  0  0
      ~∨\0 0 1 0 1        ⍝ LEADING ZEROS
1  1  0  0  0
      <\0 0 1 0 1         ⍝ FIRST ONE
0  0  1  0  0
      ~≤\1 0 1 1 0        ⍝ FIRST ZERO
0  1  0  0  0
      C                   ⍝ STRING WITH BLANKS BEFORE WORDS
 APL IS BEST
      +\' '=C             ⍝ WORD INDEX
1  1  1  1  2  2  2  3  3  3  3  3
      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             ⍝ LEADING BLANKS
1  1  0  0  0  0  0
1  1  1  0  0  0  0
1  0  0  0  0  0  0
      +/∧\' '=Y           ⍝ NUMBER
2  3  1
      (+/∧\' '=Y)⌽Y       ⍝ ROTATE TO LEFT JUSTIFY
ALIGN
ALL
LEFT
```

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 $(0\lceil {}^{-}1+\rho\rho A)+0\lceil {}^{-}1+\rho\rho B$.

The shape of the result is $({}^{-}1\downarrow\rho A),1\downarrow\rho B$.

For vector or scalar arguments: the result is scalar $\oplus/A\otimes B$.

For A vector (or scalar), B matrix the vector result R has element $R[I]\leftarrow\oplus/A\otimes B[;I]$.

For A matrix, B vector (or scalar) the vector result R has element $R[I]\leftarrow\oplus/A[I;]\otimes B$.

Generally for A and B arrays, the array result R has element $R[I;\ldots;K;L;\ldots;N]\leftarrow\oplus/A[I;\ldots;K;]\otimes B[;L;\ldots;N]$.

Conditions

Conformability requires that after allowed coercions, $({}^{-}1\downarrow\rho A)=1\downarrow\rho B$. The valid coercions are:

Scalar A becomes $(1\downarrow\rho B)\rho A$.

If $1={}^{-}1\downarrow\rho A$ then the plane across that last dimension is replicated $(1\downarrow\rho B)$ times:

$$A\leftarrow(1\phi\iota\rho\rho A)\lozenge((1\downarrow\rho B),{}^{-}1\downarrow\rho A)\rho A$$

Scalar B becomes $({}^{-}1\downarrow\rho A)\rho B$.

If $1=1\downarrow\rho B$ then the plane across that first dimension is replicated $({}^{-}1\downarrow\rho A)$ times:

$$B\leftarrow(({}^{-}1\downarrow\rho A),1\downarrow\rho B)\rho B$$

Examples

```
      1 2+.×3 4                  ⍝ (1×3)+2×4 ↔ +/1 2×3 4
11
      2 3 4+.×2                  ⍝ +/2 3 4×2 2 2, SCALAR COERCED
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 MATRIX INNER PRODUCT
36  35  26  32
34  70  44  38
      A23⌊.⌈B34                  ⍝ MINIMAX: 5↔⌊/1 5 3⌈7 2 6
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
      A32                        ⍝ FOR CHARACTER (OR BOOLEAN) ARRAY, THE
IN
ON
IO
      A32∧.='ON'                 ⍝ PREFERRED ORDER IS ARRAY∧.=VECTOR
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: $A \leftrightarrow I \circledast A$
Right identity: $A \leftrightarrow A \circledast I$
Two-sided identity: $A \leftrightarrow A \circledast I \leftrightarrow I \circledast A$

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			For Boolean arguments only		
⍟	identity	side	⍟	identity	side
⌊	MAX *	both	<	0	left
⌈	-MAX *	both	≤	1	left
+	0	both	=	1	both
-	0	right	≥	1	right
×	1	both	>	0	right
÷	1	right	≠	0	both
|	0	left	∧	1	both
*	1	right	∨	0	both
⍟	none		⍲	none	
○	none		⍱	none	
!	1	left			

*MAX is the largest numeric value directly representable:
4.31359146674E68 ↔ *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.

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: blank for character type, 0 for numeric (or Boolean) type.

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   ⍝ EMPTY
```

```
      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
```

Forms

⍳ A	Integers to A
B ⍳ 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. $,A \leftrightarrow \rho\iota,A$. 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 $\iota 1+\rho B$. For any element of C not occurring in B, the corresponding result element is $\square IO+\rho B$. There is no collating sequence built-in to APL. This function allows any desired one to be used.

Conditions

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
      ⍳5
1 2 3 4 5
      ⍳0⍝ EMPTY NUMERIC VECTOR

      ⍳1⍝ ORIGIN SINGLE VECTOR
1
      3 1 1 2⍳1 2 3 4 5
2 4 1 5 5
      'ABCDE'⍳'BEAR'
2 5 1 6
      ⎕D⍳'301'⍝ ⎕D←→'01...9'
4 1 2
      '+-×÷'⍳'A+B×CD'
5 1 5 3 5 5
      ÷⍳5⍝ HARMONIC SEQUENCE
1 0.5 0.333333 0.25 0.2
```

```
      ⎕IO←0⍝ ORIGIN 0
      ⍳5
0 1 2 3 4
      ⍳0

      ⍳1⍝ ORIGIN VECTOR
0
      3 1 1 2⍳1 2 3 4 5
1 3 0 4 4
      'ABCDE'⍳'BEAR'
1 4 0 5
      ⎕A⍳3 4⍴'APL DOESWELL'
 0 15 11 26
 3 14  4 18
22  4 11 11
      3+2×⍳5⍝ ARITH. SEQ.
3 5 7 9 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 $1\lceil(\rho\rho A)\lceil\rho\rho B$. 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 $1+(\rho\rho A)\lceil\rho\rho B$. 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               ⍝ SIZE VECTOR
1
      A                 ⍝ ARRAY
11  12  13
21  22  23
      ,A                ⍝ ROW MAJOR ORDER
11  12  13  21  22  23
      ⍴,A               ⍝ SIZE 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
AB
CD
EF
      ,CA
ABCDEF
      CA,[0.5]'?'
AB
CD
EF

??
??
??
      0,[1.5]1 3
0  1
0  3
```

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
K is a dimension selector single with integer value in $\iota\rho\rho B$

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 $\phi[K]B$. The order of the planes across dimension K is reversed. Thus, plane J of the result is plane $((\rho B)[K])-J+\square IO$ of B.

If K = $\lceil/\iota\rho\rho B$, referring to the last dimension, the $[K]$ may be elided, resulting in ϕB.

Equivalent to the general form but referenced to the end or anti-origin rather the front of the shape is $\ominus[K]B$. Thus, $\phi[K]B \leftrightarrow \ominus[(\rho\rho B)+(2\times\square IO)-K+1]$.

If K = $\lceil/\iota\rho\rho B$ referring to the last dimension from the anti-origin, the $[K]$ may be elided, resulting in $\ominus B$.

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 $A\phi[K]B$. The other forms for determining the dimension for rotation are similarly developed as above. A has shape that of a "plane" across the Kth dimension of B, i.e.,

$$(\rho A) \leftrightarrow (K \neq \iota \rho \rho B)/\rho B$$

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 $A \geq 0$, the direction is toward decreasing indices. For an element of $A < 0$, the direction is toward increasing indices.

The amount rotated is $((\rho B)[K])|A$. 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:

$$A \leftarrow ((K \neq \iota \rho \rho B)/\rho B)\rho A$$

Examples

```
      ϕ1 2 3 4
4  3  2  1
      ϕ'LIVED' ⍝ REVERSED
DEVIL
      A
11  12  13
21  22  23
      ϕA
13  12  11
23  22  21
      ϕ[1]A
21  22  23
11  12  13
      ⊖A
21  22  23
11  12  13
      C
LEVEL
      C∧.=ϕC ⍝ PALINDROME
1
```

```
      1ϕ1 2 3 4
2  3  4  1
      5ϕ1 2 3 4
2  3  4  1
      ¯3ϕ1 2 3 4
2  3  4  1
      1 2ϕA
12  13  11
23  21  22
      0 1 2ϕ[1]A
11  22  13
21  12  23
      B
TAKE   OUT    EXTRAS
      (X∨1ϕX←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 object

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 $\phi\rho B$ and element $R[I;J;...;N] \leftrightarrow B[N;...;J;I]$.

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 successive integers referring to all dimension numbers of the result $1 \leftrightarrow \wedge/(\iota\lceil/A)\in A$. Dimension numbers may reoccur. The number of different integers determines the rank of the result: $\rho\rho R \leftrightarrow \rho A\cup\iota 0$.

If A is a permutation of $\iota\rho\rho B$ (no repeated dimensions) then the result shape is the A permutation of the shape of B: $\rho B \leftrightarrow (\rho R)[A]$. Indices along the coordinate J of R come from indices along coordinate $A[J]$ of B.

If any element of A is repeated, the rank of R will be smaller than that of B. 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.

$(\phi\iota\rho\rho B)\⍉B \leftrightarrow ⍉B$
$(\iota\rho\rho B)\⍉B \leftrightarrow B$

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
```

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
V \ B	Expand with V the last dimension of B
V ⍀ B	Expand with V the first dimension of B
V \[K]B	Expand with V dimension K of B
V ⍀[J]B	Expand with V 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 $\iota\rho\rho B$
J is an index number single, $K \leftrightarrow (\phi\iota\rho\rho B)[J]$
V is a Boolean vector

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: $(\rho A)=(\rho B)[,K]$

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 ones in V as there are elements along coordinate K of B: $+/V \leftrightarrow (\rho B)[,K]$.

The result is an object with rank the same as B but having dimension K expanded to size ρV. The ones in V indicate the positions along K of the planes of B. Each 0 in V 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 single, it is coerced to the length of the indicated dimension, i.e., $(\rho B)[,K]\rho A$. If V is a scalar, it is treated as a vector with one element.

Dimension selector J counts dimensions from the end, or anti-origin whereas K counts from the beginning $J \leftrightarrow (\phi\iota\rho\rho B)[K]$. For example:

$A/[K]B \leftrightarrow A⌿[J]B$
$A⌿[K]B \leftrightarrow A/[J]B$

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 ⍝ EMPTY VECTOR

      1 0 1/A
1  3
4  6
      1 0/[1]A
1  2  3
      ⍴1 0/[1]A
1  3
      0 1⌿A
4  5  6
      0 1⌿[2]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 scalar, single, or vector $(\rho,A)=\rho\rho B$
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 $\rho\rho B$.

$A[I]$ 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, $(\rho B)[I]$. $A[I]=0$ 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 $(|A)\leq\rho B$.

Overtake: Occurs for all the dimensions $K[I]$ such that $0<K\leftarrow(|A)-\rho B$. In this case, $K[I]$ planes of <u>fill</u> are appended before (after) the $(\rho B)[I]$ 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 $0\lceil(\rho B)-|A$. 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:

$$A\leftarrow,A$$

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

$$B\leftarrow((\rho,A)\rho 1)\rho B$$

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
      ρ1 ¯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

      ρ4↓1 2
0

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

      ρ2 0↓A
0  3
      ¯2↓'ABCDE'
ABC
      (,3)↓'ABCDE'
DE
      3↓'ABCDE'
DE
      ¯2 ¯5↑'?'

    ?
```

Forms

A ∈ B	Is A a member of 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.

The comparison tolerance affects the implicit comparisons in all these functions.

Examples

```
      1 2 2 3 4∈2 5
0 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
```

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 grade it (order it) in 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.

Elements with equal values in A return indices in increasing order for either function. Comparison tolerance is not used.

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

Examples

```
      ⍋5 8 4 4 2
5  3  4  1  2
      ⍝ ASCENDING SORT
      5 8 4 4 2[⍋5 8 4 4 2]
2  4  4  5  8
      ⍋2.1 3.2 4.3 3.2
1  2  4  3
      A
1  1  0  0  1  0  1  1  0
      ⎕←B←'ABCD','12345'
ABCD12345
      ⍋A
3  4  6  9  1  2  5  7  8
      ⍋⍋A
5  6  1  2  7  3  8  9  4
      B[⍋⍋A] ⍝ MERGE
12AB3C45D
```

```
      ⍒5 8 4 4 2
2  1  3  4  5
      ⍝ DESCENDING SORT
      5 8 4 4 2[⍒5 8 4 4 2]
8  5  4  4  2
      ⍒2.1 3.2 4.3 3.2
3  2  4  1

      ⍝ IF X IS A PERMUTATION
      ⍝ VECTOR, THEN X ↔ ⍋⍋X
      X
3  4  2  5  1
      ⍋⍋X
3  4  2  5  1

      B[⍒⍋A] ⍝ REVERSE MERGE
D54C3BA21
```

Forms

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

Where N is a positive integer data object
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
4571
      ?6 6
2  2
      ?6 6 6 6 6 6
3  2  1  5  3  1
      ?2 2 2 2 100 100
2  1  2  2  95  70
```

```
      4?5
4  5  2  1
      6?6⍝ PERMUTATION
6  5  2  3  4  1
      6?6⍝ ANOTHER
3  4  5  1  2  6
      0?10⍝ EMPTY VECTOR
```

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↓⍴A),1↓⍴B`. 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↑⍴A) = 1↑⍴B
```

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 `+/⍳0`.

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 10 3
      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
      0.1⊥⍬1 2 3 4
1.234
      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
      (⍬¯2 0 2 4)⊥1 2 ¯8  ⍝ OTHER ZERO AT X = 2
¯8 ¯8 0 16
```

REPRESENT
FUNCTION (1)
T

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 $(\rho\rho A)+\rho\rho B$. The shape of the result is $(\rho A),\rho B$.

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 $(|B)<|\times/A$ then $B \leftrightarrow A \perp A \top B$

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
0123456789ABCDEF
      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⍴ BINARY
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

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 $\phi\rho B$ and the rank is 2. The result is the generalized inverse of B. If B is square, then

$$\text{Identity matrix} \leftrightarrow (\iota 1\downarrow\rho B)\circ.=\iota 1\downarrow\rho B \leftrightarrow (⌹B)+.\times B \leftrightarrow B+.\times(⌹B)$$

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

$$\text{Identity matrix } (\iota 1\downarrow\rho B)\circ.=\iota 1\downarrow\rho B \leftrightarrow (⌹B)+.\times B$$

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

$$(1\uparrow\rho A)=1\uparrow\rho B$$

The result is formally the same as $(⌹B)+.\times A$. The rank of the result is the rank of A. The shape of the result is $(1\downarrow\rho B),1\downarrow\rho A$.

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 $(⌹B)+.\times B$ or $B+.\times⌹B$. The matrix is ill-conditioned to the degree that the largest magnitude of the off-diagonal terms approaches 1.

The comparison tolerance is used to determine singularity: with large comparison tolerance most coefficient matrices are "singular"; with the comparison tolerance $\leq 1E^{-}12$, 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
```

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←10X                    ⍝ DEPENDENT VARIABLE, SIN (X)
      0.223⊥Y⌹X∘.*⍟¯1+⍳⍴X      ⍝ INTERPOLATED VALUE
0.221156329002
      100.223                  ⍝ 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      ⍝ APPLY ABOVE  (AVERAGE IF N=0)
```

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
```

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.

If S is empty, the result must not be required for further expression elaboration.

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
      ⍎B/'→LABEL'                  ⍝ COSTLY CONDITIONAL BRANCH
```

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 format
C is a character string defining the format
L is a list of components, separated by semicolons; each component is eitner a null, or a data object of any type with rank 2 or less.

Results

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

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 individual data object for the character explicit formats may be a scalar, vector or matrix. 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 format 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 are recognized as encountered along any path. The diagrams are rigorous without being cumbersome. The optional constructs are indicated, in an acceptable order. The alternatives are shown as parallel paths.

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 lines

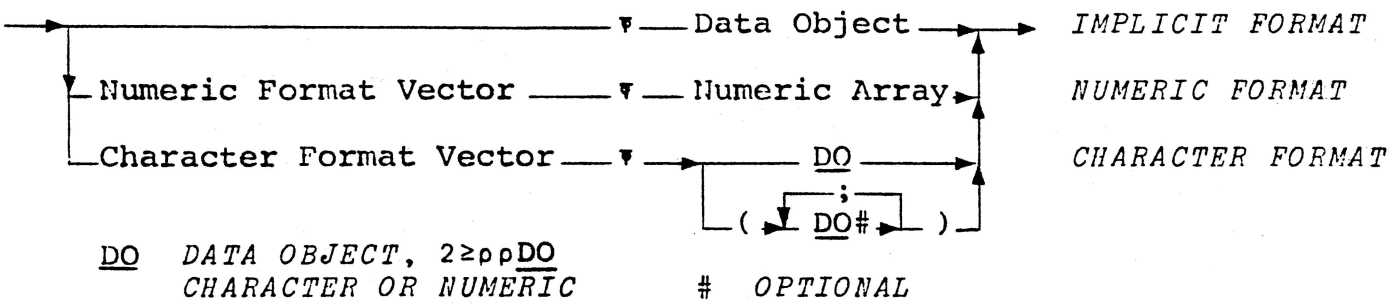
any path traced through a diagram that does not violate arrow direction will produce a syntactically valid form

Units with an appended # may be optionally omitted

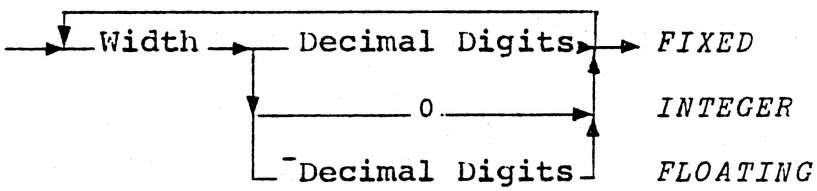
iteration is achieved by a backward line

A maximum of one occurrence of a syntactic unit in one instance of the diagram containing iteration in which it occurs is indicated by a bridge . This prevents conflicting use.

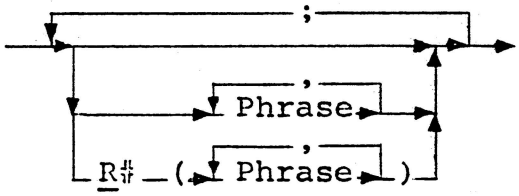
Figure 5-1 Format Syntax Diagrams



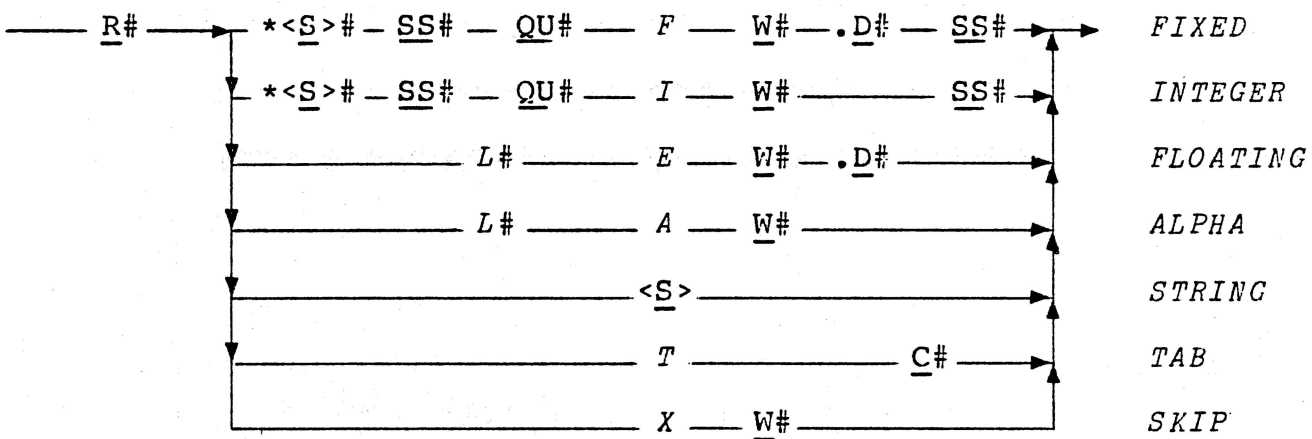
Numeric Format Vector



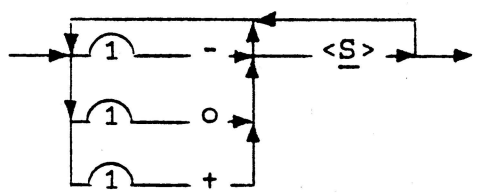
Character Format Vector



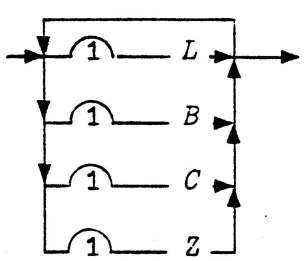
Phrase



SS Sign Selector



QU Qualifier



L LEFT JUSTIFY
B SKIP IF ZERO
C INSERT COMMAS
Z INSERT LEADING ZEROS

R REPLICATOR

S STRING

C COLUMN

W WIDTH OF FIELD

D DECIMAL DIGITS

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 any element of E is negative, a column is provided for the sign for every element of E.

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 number, and all decimal points are aligned (or omitted for small integers in E).

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.

Print precision also controls the printed numbers by determining the maximum number of significant digits displayed.

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
```

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 the leftmost column for negative mantissa sign (otherwise blank). The next column is the mantissa integer part N, $1 \le N < 10$; 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+5 and may need to be as much as D+7 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
```

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 a 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 element of the 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 until component exhausts	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 digits to right of decimal point, default is 0	decimal digits
L	is	B or C or B C or empty	left decorator
B	is	*<text>	background for field
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	1 or more (each used at most once in C) 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 (none used more than once) 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, zero or more	next column

The prior syntax chart provides named syntactic elements for semantic description only. The terminal forms (shown in APL font) correspond to those used in the syntax diagram.

In general, a right argument data object is treated as a matrix. A vector is treated as a matrix with only one row. A scalar is treated as a one-column matrix with as many rows as the maximun of any data object in the list.

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 at the bottom. 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 to exhaust the list component.

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., 3.2E¯2 or 9.73E21. A position is always used for mantissa sign. This format can be explicitly justified left, or right by default. Decimal points are aligned.

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. Decimal points are aligned.

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.

M The integer phrase replicator specifies the number of uses of the phrase before moving to the next phrase in the clause; 1 if elided.

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 optional decimal digits 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 formats.

- 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. Blank fill for the largest length of right sign selectors applies to any absent right sign selector.

*<text> A field background can be specified. The text, replicated if necessary, is initially placed in the field, and then may be partially replaced; i.e., as a check protector.

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    6.25E¯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
      ¯0.76
      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'▼Q¯1 2 34 567
¯1
2
34
567
```

Background

```
      '*<○>I5'▼¯1 0 2
○○○¯1○○○○0○○○○2
      '*</|\>I5'▼1 23 456 789012 ⍝ REPEATS, OVERFLOW OVERRIDES
/|\/1/|\23/|456*****
```

Sign Selectors

```
      '+<P>○<Z>-<N>I5'▼¯1 0 2
  N1  Z0  P2
      '-<(>I5-<)>'▼Q¯1 0 2  ⍝ (NEGATIVE)
 (1)
  0
  2
      '-<->I12+<        >○<     >-<>'▼Q¯1 32 0 541 ¯35
          -1
 32
       0
541
         -35
```

Blank Zero Field

```
      'BI5'▼¯1 0 5
   ¯1          5
      'BLI5'▼Q¯1 0 5
¯1

5
```

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'          ⍝ RIGHT JUSTIFIED
 G O O D
 W O R K

      'LA2'▼'LEFT'                 ⍝ LEFT JUSTIFIED
L E F T

      'A1,A2,A3,A4,5A1'▼'OPEN DOOR'
O P  E   N DOOR
```

Tab and Skip

```
      'I15,T0,I5,X20,I5'▼25 50 75 ⍝'X20' SKIPS 'I15,T0'
   50         25                75

      'I15,T0,I5,I25'▼25 50 75     ⍝'I25'REPLACES'I15,T0'
   50                           75
```

Text

```
      '<<>,I5;A1'▼(⍳1 10 ¯25;'>') ⍝ SCALAR REPEATS IN ALL COLUMNS
<     1>
<    10>
<    25>
```

Combined

```
      'I5;X4,A2'▼(5 6;'AB')
    5     6      A B

      'I5;I5;A2'▼(100;,2;⍳'AB')   ⍝ VECTOR SINGLE DOESN'T REPEAT
  100     2 A
  100       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
```

```
    ⍕
```

Clause is Phrase or Phrase,...,Phrase

```
          'LI5'⍕1 3 ¯8 ⍝ PHRASE FORMAT REUSED
     1    3    ¯8
          'LI5,F5.2'⍕1 3 ¯8 ⍝ CLAUSE FORMAT CYCLICALLY REUSED
     1     3.00¯8
```

Group is Clause or (Clause) or Replicator(Clause)

```
          'LI5'⍕1 22 333 ⍝ CLAUSE, AND PHRASE
     1    22   333
          '(F5.1),LI5'⍕1 3 ¯8 ⍝ 'F5.1' USED TILL EXHAUSTION
       1.0  3.0 ¯8.0
          '2(I2,2I5)'⍕1 22 333 4 5555 66666 ⍝ USE IS IN GROUP
      1   22  333 4 555566666
```

Segment is Group or Group,...,Group or empty

```
          'LI5'⍕1 3 ¯8 ⍝ GROUP, CLAUSE, AND PHRASE
     1    3    ¯8
          '2(I3,2F5.1),I5'⍕1 2.2 3.3 4 5.5 6.6 7 888
       1  2.2  3.3  4  5.5  6.6    7888
          ''⍕1 22 333 ⍝ EMPTY USES IMPLICIT FORMAT
        1 22 333
```

Format is Segment or Segment;...;Segment

```
          'I5,I3,I8,I4'⍕1 3 ¯8 ⍝ SEGMENT, GROUP, AND CLAUSE
       1  3       ¯8
          'I5,(F5.1);I3'⍕(1 3 ¯8;21 22) ⍝ COMPONENT LIST
       1  3.0 ¯8.0 21 22
          'I3;(2F5.1,I3)'⍕(1 2;0.2+3 ¯8 6 4 5;25 30)
     1  2  3.2 ¯7.8  6  4.2  5.2 25 30
          'I3;<|>;LI5;2A1'⍕(⍉1 3;;⍉2 4;2 2⍴'ABCD')
       1|2    AB
       3|4    CD
          2 11⍴'I3,<|>,LI5;2A1'⍕(1 2;'AB';3 4;'CD')
       1|2    AB
       3|4    CD
          'I3,<|>,LI5;T0,A1,X8,A1'⍕(2 2⍴1 2 3 4;2 2⍴'AECD')
     A 1|2    B
     C 2|3    C
```

SECTION 6

SYSTEM VARIABLES, SYSTEM FUNCTIONS AND SHARED VARIABLES

The system variables provided within each workspace of the APL processor specially tailor the processing to the application of that workspace.

The system functions permit the user to perform many functions that query or alter the run environment of the active workspace or to query the total environment of the APL system.

The shared variables and the system functions that handle them allow 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

System variables are used to inform the APL system of the user's desires. System variables are less general then normal variables in that they have restricted types and ranks, and may only be assigned values in limited domains. System variables always have values.

System Variable	Name	Purpose	Default	Domain
⎕LX	Load Expression	Evaluate the (non-empty) expression upon loading the workspace.	empty ''	Character vector representation of a valid APL expression, or empty.

The Load Expression allows uniform initiation of any loaded workspace. Examples of use are to display a news message, to call an APL function that may log usage of the workspace, and to start the application without ever being in APL calculator mode. There is no system command comparable to Load Expression. Load Expression may be localized.

These system variables always have scalar numeric values. Initial values are provided in a clear workspace by default from the last use in the account of the corresponding system commands. 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 results of primitive functions: < ≤ = ≥ > ≠ ∈ ⊂ ⊃ ∩ ∪ ⍳ ⌈ ⌊ !	$1E^{-}10$	$0 \leq N < 1$
⎕*IO*	Index Origin *)ORIGIN*	origin for ordinal counting, applies to the primitive functions: ⍳ ⍋ ⍒ ? [ ] ⍉	1	0 or 1
⎕*PP*	Printing 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 used in ?	131131704506	integer 0 thru $^{-}1+2*39$

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 is adjusted for the global environment origin value before return to the calling function.

In a clear workspace the suggested default value for each system variable will apply unless that default has been changed. Change is by use of the corresponding system command. 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. See also the *)CLEARACCESS* system command.

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 the dimensions in an array, and the elements in any dimension.

Origin	Denumeration begins with	Example
0	0	0 1 2 ...
1	1	1 2 3 ...

The Index Origin affects the initial number for ordinal numbering:

⍉	permute dimensions (dyadic) left argument
⍳	integers (monadic), index of (dyadic)
⍋⍒	grade up, grade down
?	roll (monadic), deal (dyadic)
[]	subscripts on arrays [bracketed]
	dimension selector [bracketed]
	laminator [bracketed]
	file component selector [bracketed]

The Printing 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, Printing 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* □*LF* □*UF*

System Function	Name	Action/Result
□*CR* N	Canonic Representation	Character matrix. Each row is a line of the function N, including the header.
□*VR* N	Vector Representation	Character vector. Each line of function N is terminated by □*R* after the last non-blank.
□*FX* C	Fix	Defined function. A valid character representation of a function is converted into a function. The result is the function name, suppressed unless required.
□*LF* A	Lock Functions	Lock all unlocked function names in rows of A. The resulting Boolean vector elements are 1 if successful, in row name order, suppressed unless required.
□*UF* A	Unlock Functions	Unlock all own locked functions named in rows of A. The resulting Boolean vector elements are 1 if successful, in row name order, suppressed unless required.

Where N is a character vector name of an unlocked defined function.
A is a character matrix with function names as rows, or a single function name as a vector.
C is a character vector or matrix representing a function.

Representation results omit line numbers and opening and closing dels(∇). An empty result means that the name is not that of an unlocked function. Canonic Representation is useful for user-written function editing routines where line rearrangement, function merging or separation is desired. This form generally takes more space than Vector Representation, particularly if the line lengths differ.

The defined function name resulting from a Fix must not have prior meaning. The name of a fixed function may be local to some other function. The definition of the fixed function disappears upon exit from the function to which it is local.

The Canonic and Vector representations of a function do not include the settings of any debugging aids. A fixed function has all debugging aids reset.

The optional result of Lock Functions or Unlock Functions is a Boolean Vector with value 1 if the indicated action took place for that name. A 0 result indicates either that the name is not a function name, that the function was already in the indicated state, or that the unlock attempt was to a function obtained in locked state (sealed) from another account library.

Name system functions work with a character 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, Δ and ∆.
□*NC* C	Name Classification	Integer vector indicates name use in the current environment: a single result for the name in character scalar or vector C; the values correspond to the rows of matrix C. 0 no associated object 1 label 2 variable 3 function 4 other (group)
□*EX* C	Expunge	Objects corresponding to names or system variables in character vector or matrix C are expunged. If required, the result is a Boolean vector with a one each place the corresponding name from C no longer exists in that environment. Names of labels, groups, or active funtions cannot be expunged, and return a zero result if required.

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.

A character string argument C to Name Classification or Expunge must contain only one name. A character matrix argument must contain one name per row. Otherwise, a (nonexistent) name is expunged; and the result is 1 for that "name".

Expunge may be used to eliminate meanings for objects from the current environment so long as they are neither names of active functions nor labels. Other local names can be expunged. Expunge of a system variable causes its value to become the default, clear workspace value.

DEBUGGING AIDS (1)

⎕ST ⎕SS ⎕SM
⎕RT ⎕RS ⎕RM
⎕QT ⎕QS ⎕QM
⎕MC ⎕MV

Result trace indicators, execution stop indicators, and both usage and computer time counters may be attached to individual 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
⎕MC F	Monitor Counts	N *⎕MC* F	V
⎕MV F	Monitor Values	N *⎕MV* F	V

Where
- F is character vector name of unlocked defined function
- N is numeric vector of valid line numbers, including 0
- L is numeric vector of lines with property (set, reset) after function execution, 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 line counts or values accumulated during executions since set.

Defaults for the debugging aids are Reset, with 0 counts and values. Set Monitor initializes the monitor result counts to zero. Reset Monitor does not alter attained monitor results.

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.

Debugging aids settings are not part of canonic or vector representations. Debugging aids are reset and zeroed on fixed functions.

Any function edit of an existing line does not alter the debugging aid settings. A new line gets the default settings. Debugging aids are attached to line content and change as the line numbers change.

The debugging aids are disabled on locked functions. Locking a function does not alter the setting or attained monitor values of a function. Subsequent unlocking of that function reenables them. Debugging aid settings are preserved by)SAVE and)LOAD; they are reset and zeroed for)COPY or)PCOPY.

During function execution, the effects are as follows on encountering a line on which one or more aids are set:

Aid	Header Line 0	Body Line
Trace	result returned by function	result
Stop	suspend prior to return	suspend before execution
Monitor	increment number of calls and time to call and return	increment count and CPU time in line execution

The Trace display forms are:

```
Function-Name [Line-Number]    if no result, or implicit output
Function-Name [Line-Number]    Type (Shape) Value     if result
```

The display form describes the value of the control transfer target or last 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. If the line result is normally an implicit output, that output is not repeated.

The Stop result form is: Function-Name [Line-Number]*

After a Stop on the header line 0 (following function body execution), the local names are still defined.

Monitor Counts indicate the number of completed executions since set of monitored lines. After a period of use, any count of 0 indicates that the line is unexecuted (and potentially erroneous or redundant).

Monitor Values only accumulate time in executing the line; not in initial compiling, line sequencing, or elaborating primitives in the string argument to the evaluate primitive. Monitor time spent in a called function is accumulated there, instead of in a line of the calling function. Unmonitored lines have 0 time. Values are returned rounded with millisecond units; but are internally accumulated with precision 0.1 millisecond. Correct values should not exceed about 100 seconds, as they are actually retained as (2*20)| value.

If Monitor is set on a line, the overhead for monitoring is equivalent to about one additional primitive function.

Normal execution can be altered using the following system functions.

System Function	Name	Result
⎕DL D	Delay	optional actual delay in seconds
⎕ED S	Edit	edited line after editing with normal entry of editing specification and alteration
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 a number indicating the 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. A minimum delay occurs for non-positive D.

Each Edit function accepts a character string as the right argument. This string may be empty, or may be composed from spaces, the visible APL characters, and null.

The monadic Edit form displays the string and returns to the left margin for entry of an edit specification line applied to the characters above using the characters ' ', '/', '.' and ',T' as described in section 2.

The Phrase Edit dyadic form uses the Boolean left argument (of the same length as the string) with each 1 indicating a phrase end. This avoids the specification line entry. It also allows multiple prompts and inputs on the same line. Single B is coerced to the length of string S.

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.

The single characters or character vectors below are the values returned by niladic system functions.

System Function	Name	Result
⎕B	Backspace	scalar backspace character
⎕F	Form	scalar form feed character
⎕I	Idle	scalar idle character
⎕L	Linefeed	scalar linefeed character
⎕M	Margin	scalar left margin character
⎕N	Null	scalar null character
⎕R	Return	scalar cursor (or carrier) return character
⎕T	Tab	scalar tab character
⎕A	Alphabet	character vector, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
⎕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 eight are associated with output processing for terminal display. Some terminals may not adequately accept these characters.

The Backspace "⎕B" 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 Linefeed "⎕L" character can be used for advancing the display line without changing the horizontal column of the cursor. The Margin "⎕M" character returns the cursor to the left margin. The Return "⎕R" character causes completion of an output line, just as the RETN key does for input. It includes both line feed and margin to the left margin. In cases where the cursor is at the left margin, Linefeed and Return have the same external effect.

The Tab "⎕T" 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 □F □I □L □M □N
□R □T □A □D □AV

The Idle "□I" 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 Null "□N" character is not transmitted so has no effect on display. It may be used as a marker or as a pad in an array containing words of variable length to simplify display message preparation. It may also be used as a marker in a □ED string. Subsequent analysis of the input can then assure that the string is not altered and that field sizes are not exceeded.

The Form "□F" feed character causes properly equipped terminals to move to the top of form.

The Alphabet "□A" and Digits "□D" are often useful in text processing.

The Atomic Vector "□AV" includes all characters defined in APL. Table 6-1 shows the displayable characters. The two lines below each character indicate the (0 origin) index and hexadecimal equivalent.

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 191 are non-printing. The last 4 rows are omitted, all entries would be ???. Any attempt to display one of these results in the squish-quad "⌷". Since these are uniquely displayed, their use should be carefully considered. The 6 characters of the extended APL character set that are on the 94 character terminals may be entered as overstrikes on any terminal. If the actual symbol is available, it may be used. The entire ASCII visible character set is also included, often entered with overstrikes on APL terminals. These extra graphic characters are not part of the necessary APL character set. The (zero origin) □AV numbers and acceptable overstrikes are:

□AV	Name	Symbol	Overstrike	With
APL				
77	left tack	⊢	⊬	(-
78	right tack	⊣	⊣	-)
133	diamond	◇	⋄	< >
134	left brace	{	⊂	[∘
135	right brace	}	⊃	] ∘
143	currency symbol	$	$	S |
ASCII				
131	cent		¢	c |
175	ampersand	&	&	7 \
176	circumflex	^	^	^ ¨
177	paragraph	¶	¶	c n
178	percent	%	%	/ :
179	accent, grave	`	`	' ¨
180	hash	#	#	H =
181	at sign	@	@	0 ⍺

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	0A	0B	0C	0D	0E	0F
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	1A	1B	1C	1D	1E	1F
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	2A	2B	2C	2D	2E	2F
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
?	○	~	∧	∨	⍲	⍱	<	≤	=	≥	>	≠	ρ	,	⍳
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
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	4A	4B	4C	4D	4E	4F
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
□I	⊃	Δ	Δ	←	:	□	(	)	[	]	'	→	;	⍪	∘
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
∇	⍫	¯	_	A	B	C	D	E	F	G	H	I	J	K	L
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	□	⍎
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
⍺	⍵	¨	¢	⍸	⌾	⍷	⍯	□B	□L	□R	???	□M	???	□T	$
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
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	9A	9B	9C	9D	9E	9F
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
⍁	⍂	⍃	⍄	⍅	⍆	A	Ä	Ö	□N	Ä	Ä	Ö	□F	???	⍔
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
⍓	⍗	⍀	⍉	H	⍝	???	???	???	???	???	???	???	???	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191

STATUS INQUIRIES
⎕PT ⎕PW ⎕WI ⎕AN ⎕AI
⎕NEWS ⎕LC ⎕TS ⎕UL ⎕UN
⎕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	
⎕AN	Account Name	character vector: identifier	⌶29 ↔ ⎕AN
⎕AI	Accounting Information	computer time, IO 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
⎕UN	Utility Names	character matrix: authorized for user	
⎕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. +/⎕LA, +/⎕FA, and +/⎕SA 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.

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 names. The utility names available to the user are given as the result of ⎕*UN*.

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. Only the surrogate name used need be known to the 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 for several offers, each with a distinct internal name. The same surrogate may be used for several offers to another process. They are matched in the time sequence they are accepted by the process. 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 (assignment by replacement) and using (once assigned, then referencing the present value of) the variable. Insertion should be avoided since insertion acts like a use immediately followed by a set, and can cause deadlock if use is constrained. Setting by modification or modified insertion is disallowed.

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 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
		first use, or two successive	requires intervening
Set A	$\begin{matrix}1 & 1\\ & \end{matrix}\uparrow ACM$	sets by A	access by B
Set B	$\begin{matrix}1 & 1\\ & \end{matrix}\uparrow ACM$	sets by B	access by A
Use A	$\begin{matrix} & 1\\ 1 & \end{matrix}\uparrow ACM$	uses by A	set by B
Use B	$\begin{matrix} & 1\\ 1 & \end{matrix}\uparrow ACM$	uses by B	set by A

Note the symmetry of the above. For elements with value 1 in the:

Top row - After an initial set by either process, two successive sets by one process requires an intervening access by the other. This assures that the second process has an opportunity to accept the value set by the first.

Bottom row - The first use after becoming 1 may be by either process. Then two successive uses by one process requires an intervening set by the other. This assures that (at least one) new value has been set prior to use.

First column - controls on one's own process setting and use.

Last column - 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 $2\ 2\rho$OWNACV and the effect of the setting by the partner process as viewed by one's own process is $\phi 2\ 2\rho$ PARTNERACV.

The resulting ACM$\leftrightarrow$(2 2ρ OWNACV)$\vee\phi$ 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.

Several offers to another process may be made with the same surrogate. This may lead to confusion. Acceptance of an offer initiates sharing with the oldest outstanding offer. Retraction only reduces the degree of coupling, it does not automatically couple with another offer by the process with which it had been coupled. Thus both former partners can have outstanding offers to the same surrogate! The time of retraction becomes the new time of offer for the former partner's outstanding offer.

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. Two processes making general offers of the same name will not couple.

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, copying, or recovery after disconnect or)COFF does not reinitiate the shared variable.

The maximum number of shares offered at one time by one process is limited (□*SA* shares availibility interrogates this) as is the total number of shared variables over all users.

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

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, with possibly blank padding needed before (between) or after the name(s).

P is a matrix with as many rows as N, or a vector (used for each row if N is a matrix). 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 (internal name conflicts with existing name that is a label, function, group, or variable shared with another processor)
- 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.

The names of all shared variables in the dynamic environment may be determined as the rows of

```
(0≠□SVO M)⌿M←□NL 2
```

Conditions

The second process is not automatically informed of the offer. The value associated with an already existing variable that becomes a shared variable is unchanged until first an assignment is made to the shared variable after sharing is initiated.

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 ONE

      ⎕←'TWO'⎕SVO'Y Z'
1
      ⎕SVO'Y'
1
      ⎕SVO'Z'
0


      Y
7700
      Y←'HI TWO'
      Y
HI TWO



      BB←'CABBAGE'
      B←'PATCH'
      ⎕←'TWO'⎕SVO'BB B'
2
      BB
1  2  3
      B
PATCH

      ⎕SVO'C'
0
      ⎕←''⎕SVO'C'
2
      C
*** VALUE ERROR ***
      ∨
      C
      C←'HELLO'
      ⎕←''⎕SVO'C'
2
```

```
      ⍝ PROCESS TWO

      ⎕←'ONE'⎕SVO'A Z'
2
      A←7700
      A
7700


      A
HI TWO
      ⎕←''⎕SVO'B'
1
      B←1 2 3

      B
1  2  3
      ⎕SVO'B'
2
      C←32
      ⎕←'ONE'⎕SVO'C'
1


      C
32

      C
HELLO
```

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.

Access control can be imposed only after an offer has been tendered.

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←4ρC                    if N is a vector
C←((1↑ρN),4)ρC           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 ONE

      □←'TWO'□SVO'X'
1
      □SVC'X'
0  0  0  0
      □SVO'X'
1
```

```
      ⍝ PROCESS TWO

      □←'ONE'□SVO'X'
2
      □SVC'X'
0  0  0  0
      □SVC'X'
0  1  0  1
      □←1 1 0 0□SVC'X'
1  1  0  1
      □←'ONE'□SVO'A B'
1
      □←1 0 0 1□SVC'A'
1  0  0  1
```

```
      □←1 0 1 0□SVC'X'
1  0  1  0
      □SVC'X'
1  1  1  0
      □←0□SVC'X'
1  1  0  0
      □←''□SVO'C B'
2
      □SVC'CX'
0  1  1  0
1  1  0  0
```

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 17 or less 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 to those placed by the formerly sharing partner's contribution.

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 (as does signing off, disconnect, or loading a new workspace, or exiting the function to which the shared variable name is local) and reduces the degree of coupling to 0.

Recovery after a)COFF or disconnect does not restore the tender.

Conditions

Once a share offer has been accepted there is no direct way to find out with whom the variable is being shared. If needed, this information should be recorded when the offer is made.

Examples

Time sequence is downward for both columns in parallel. Entries on the same line could occur in either order.

```
      ⍝ PROCESS ONE

      □←'TWO'□SVO'X Y'
1
      X←5
      □←1 0 0 0□SVC'X'
1  0  0  0

      □←□SVR'X'
2
      X
5
      □SVQ''

TWO
      □SVQ'TWO'
Y
      □←''□SVO'A B'
1
      □←□SVQ''
TWO
```

```
      ⍝ PROCESS TWO

      □SVQ''
ONE
      □SVQ'ONE'
Y
      □←'ONE'□SVO'Z Y'
2
      Z
5
      □SVC'Z'
0  1  0  0

      □SVC'Z'
0  0  0  0
      □SVQ''

      Z
5
      □SVO'Z'
1

      □SVQ''
```

The primitive monadic function defined in early APL systems for querying the environment has the form:

I 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
I20	scalar time of day	`⌊0.06×0 60 60 1000⊥3↑⎕TS`
I21	scalar CPU time used this session	`0.06×1↑⎕AI`
I22	scalar bytes remaining unused in the workspace	`1↑⎕WA`
I23	scalar number of users currently signed on	`⎕UL`
I24	scalar time of day at start of the work session	`⌊0.06×(0 60 60 1000⊥3↑⎕TS)-1↑2↑⎕AI`
I25	scalar date in form MMDDYY where M,D,Y are digits representing month, day, and year respectively	`100⊥100|1⌽3↑⎕TS`
I26	scalar first element of I27	`1↑⎕LC`
I27	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`
I29	character vector containing the 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 FUNCTIONS

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 SYSTEM CHARACTERISTICS

A file has a name and may have components that contain APL data objects. Some limits on the file system are imposed by APL. File status may be active or inactive. File updating is controlled for integrity.

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 usercode required if the file is owned by another user. The Acct is a string of 1 to 17 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 both 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. It is unavailable while the file is closed: the up-date file is merged with the old master file forming a new master file. Only after the closing is complete are the old up-date and master files destroyed. Thus the master 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 of one or more components. Any component updates while the user has the file held are provisional. They become part of the up-date file only after a file free is executed by that account. Any return to execution mode (or a terminal disconnect) before the free occurs voids the provisional updates. This capability protects the update 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 a result is 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. Protection for the double ATTN may be achieved using the function hold and free in a (multi-line) locked function.

FILE 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 (as a character string), symbolically represented as 'F'.

Forms

	▲ F	Create file F
	▲ F[O/P]	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 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 resulting file name F (or N) is returned if required.

Create: A file named F is created with no components.

Change Password: New password P replaces 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 named F is renamed to become N.

Destroy: The file named F owned by this account is destroyed if it exists and is not currently held by any other user.

Conditions

Create: The File Name must not already exist.

Change Password: This can only be done by the file owner. The file may be active.

Rename File: A file can only be renamed if inactive.

Destroy: The File Name (including lock if any) 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
```

Forms

	⊂ F	Read and pop first component from file F
	⊃ F	Read and pop last component from file F
A	⊂ F	Append component before components already in file F
A	⊃ F	Append component after components already in file F

Where F is File Name
A is any APL data object

Actions/Results

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.

Examples

```
      'JONES'⊃ 'PERSONS'
      (2 2 ⍴ 1 1 4 7) ⊂'PERSONS'
      'SMITH'⊃'PERSONS'
      ⊃ 'PERSONS'
SMITH
      ⊂'PERSONS'
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'
```

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 0 thru 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. Note that the file itself is changed, components dropped or not taken are destroyed.

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'
0⍋'FILENAME' ⍝ CLEAR FILE
```

FILE COMPONENTS
COMPRESS, EXPAND
⍁ ⍂

Forms

B ⍁ F	Compress by B components from file F
B ⍂ F	Expand by B components of file F

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 resulting file has 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 resulting file has an expanded, ordered component set preserving the order of the original components, and 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: $(\rho B)=3\circledcirc F$.

Expand: The number of ones in B must be the same as the number of components in the original file F: $(+/B)=3\circledcirc F$.

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	Detach from 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.

Detach: 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.

Detach: 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
```

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
1 file exists and is not active
2 file is active in some account(s)
4 file is unavailable
5 file is held by some account

Query: The file becomes active (if not already active). The numeric 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 cycle number of last reorganization
6 number of accounts with file open
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

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 distinguished by valance (number of arguments) and by whether or not it returns an explicit result.

	Function Valance		
	0=niladic	1=monadic	2=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 global name is any name used in the function that is not local to it. Any alteration to a global variable is called a side-effect of executing the function.

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.

There are two complementary uses of local names: to shield the local effects from need for consideration at a more global level, and to protect global meanings from unintended local change.

Note that a name global to one function may actually be a local name in another function in the calling path of the first function.

System variables and the character input prompt 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:

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 function line has a line number. The header is implicitly line 0. The body lines are 1, 2, etc.. When displayed in function definition and editing mode, each body line is preceded by a bracketed prompt including the line number. For example:

```
     ∇AVERAGE[□]∇
    ∇ AVE←VERSION AVERAGE VALUES;SUM;SIZE
[1]   →VERSION/BRIEF
[2]   ⍝ VERSION IS INTEGER, 0=LONG, 1=BRIEF FORM; VALUES IS
      A NONEMPTY NUMERIC VECTOR
[3]   ⍝ AVE IS NUMERIC VECTOR SINGLE, THE AVERAGE OF VALUES
[4]   SUM←+/VALUES⍝ SUM OF VALUES
[5]   SIZE←⍴VALUES⍝ SAMPLE SIZE
[6]   AVE←SUM÷SIZE
[7]   →0⍝ EXIT UNCONDITIONALLY
[8]  BRIEF:AVE←(+/VALUES)÷⍴VALUES
    ∇
```

The first entry requests the function display. The header line of this sample function defines a dyadic value-returning function AVERAGE. The six local names are: right argument VALUES, left argument VERSION, result AVE, local names SUM and SIZE, and label BRIEF. Line 1 branches to BRIEF if the actual left argument when the function is called has value 1. Comments in lines helps documentation, particularly regarding arguments and results. Lines 4 - 6 contain an expanded algorithm exiting at line 7. Line 8 is a briefer one with the same result. It needs no local variables. It exits thru the end. The opening and closing ∇ 'del' characters bracket the function definition. Note the display folding of line 2 before a name.

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. 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 applies to relieve the suspension and continue execution. Terminate causes abandonment of execution of the function and any other functions pending its completion.

BRANCH,
TERMINATE,
LABEL (2)
→ :

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.

In use, branching is either unconditional or conditional (the value of some expression determines the next line). Branching is also classified as decision (selecting an alternative, generally forward going), or iterative (generally backward looping).

Examples

Typical branching expressions include:

```
→L                 ⍝ GO TO L UNCONDITIONALLY, L IS LABEL OR LINE
                   ⍝ NUMBER
→0                 ⍝ EXIT THE FUNCTION
→B/L               ⍝ IF B=1 THEN GO TO L ELSE CONTINUE IF 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, SELF-RELATIVE
→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, lines with 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 calculator 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. A locked function in a workspace of one usercode cannot be unlocked in another account: either by loading that containing workspace, or by copying the locked function from it into another workspace, even though the latter workspace is owned by the copier. A non-owned locked function is sealed in the other account that uses it.

During execution of a locked function, a user-initiated interrupt or any error encountered causes function exit, and passes any error message to the caller environment. Line trace, suspend, and monitor within a locked function are ignored, even though their settings and values prior to locking are retained should the function be subsequently unlocked.

Examples

```
      ∇R←F X;Y∇                          define function F header, close it
      ∇F                                 open F
[1]   LABEL1:Y←LINE X                    prompt for next line, enter it
[2]    ⍒                                 another prompt, close with lock
      ⍒F                                 open locked function
[2]   LABEL2:R←G X+Y⍝ COMMENT∇           next line prompt, enter it, close
      ∇F                                 unlocked, open
[3]    ∇                                 close
      ∇THIS_IS_A_LONG_FUNCTION_NAME      another function
[1]    'THIS IS A LONG LINE THAT WILL     portion 1, terminal ATTN
                                       <  continuation mark
  NOT FIT ON A SINGLE DISPLAY LINE'∇     last portion, close
```

The first example creates a function header and then immediately closes, effectively reserving a name for subsequent function editing that will provide the function body. Any execution call on F must be monadic. At this stage without a body, if a result is required by the call then an error will result, trapping the incomplete definition.

The next open returns the prompt [1]. The content of line 1 is then entered. After each entry another prompt is displayed.

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 last open of function F demonstrates that the function is now unlocked. After the prompt [3] the function is again closed.

The second function illustrates entry in two portions, using a terminal ATTN to indicate the continuation request, rather than the eventual RETN that completes the entry. Portions ater the first continue at the left margin.

FUNCTION LINE
REPLACE, INSERT
ACTIONS (1)
↓ ↑

Forms

Forms		next prompt if T is empty	next prompt if T is 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 is checked for valid syntax. If acceptable, 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.

An initial ATTN after a displayed next prompt retrieves the last entered expression as the initial content of that line for possible augmentation and entry.

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.

The content of the prior executed expression can be recovered and included as a function line by first using the [A] form to display the line number prompt by itself, then an initial ATTN.

Examples

```
      R←Y+3                          execution mode entry
      ∇F                             open function F
[3]    [2]R←Y+3                      initial ATTN retrieves expression
[3]    [↑]⍝ NEW LINE 1               before old line 1
[↑2]   [↓]⍝ LAST LINE                after current last line
[5]    [↑3]⍝ NEW LINE 3              before current line 3
[↑4]   'LINE IN STRING'              also before that line
[↑5]   [⎕]∇                          display, close
    ∇ R←F X;Y
[1]    ⍝ NEW LINE 1
[2]   LABEL1:Y←LINE X
[3]    ⍝ NEW LINE 3
[4]    'LINE IN STRING'
[5]    R←Y+3
[6]    ⍝ LAST LINE
    ∇
```

FUNCTION LINE
EDIT ACTIONS (1)
∈ α ω ⍳

Forms		next prompt
[∈A]	Full edit line A	[Z]
[αA]	Prefix edit line A after line number	[Z]
[αA]T	Prefix insert text T before text of line A	[Z]
[ωA]	Suffix edit line A	[Z]
[ωA]T	Suffix insert text T after text of line A0	[Z]
[⍳A]	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 proceed using the expression edit method described in Section 2, starting at step 3. The entire line (including prompt, labels, APL expressions, and comments) may be edited. At least an action specifier must remain when the entry is made.

Prefix Edit: Display the prompt and then await entry, as if a "." specifier appeared after the prompt. A terminal ATTN is required to obtain the rest of the line. 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: Display line A, await text entry at the end. A change near the end of a line may often be made more quickly using this action (by backspaces, embedded ATTN, then correction) than using full edit.

Suffix Edit with string T appends T to the end of line A, without displaying it.

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 of a line that is longer than the display is conducted the same as is done for editing a long expression.

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.

A syntax error in any line entry causes a line marking its position with "v" and unclosed parentheses or brackets to its left with "*". Immediately thereafter the erroneous line is displayed and the display cursor returns to the left margin to await specification control entry. An error in a line that is longer than the display is indicated in a window in that line, ellipsis (...) denote that text is not shown at either end.

If)KEYWORDS are ON, function editing is done on the displayed keywords. If APL symbols are available for entry, they may be used (except for the keyword brackets "<" and ">").

Examples

```
     ∇F[∈2]                          full edit
[2]  LABEL1:Y←LINE X
           ,R
[2]  LABEL1:Y←R LINE X
[7]   [α1]                           prefix edit, terminal ATTN, then RETN
[1]   R←INTO X⍝NEW LINE 1
[7]   [α5]LABEL2:                    prefix insert, no ATTN, no display
[7]   [ωLABEL2-2]                    suffix edit, label2 relative
[3]   ⍝ NEW LINE 3                   embedded ATTN for change at end is
                 v                   often faster than full edit
                 END OF 3
[7]   [⍳5]∇                          inject in edit buffer,close
                                     initial ATTN recovers expression
     LABEL2:R←Y+3                    remove label, can execute
     ///////
     R←Y+3
     ∇F[□]∇
    ∇ R←F X;Y                        open, display, close
[1]   R←INTO X⍝ NEW LINE 1
[2]  LABEL1:Y←R LINE X
[3]   ⍝ NEW LINE END OF 3
[4]   'LINE IN STRING'
[5]  LABEL2:R←Y+3
[6]   ⍝ LAST LINE
    ∇
```

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 ⍺ is used to indicate any one of the allowable actions.

Unqualified: applies to all lines in the range.

Form	Line Range
[⍺]	0 thru Y inclusive
[⍺A]	A only
[A⍺]	A thru Y inclusive
[A⍺B]	A thru B inclusive

Name Qualified: applies to only those lines within the inclusiverange that contain the name X as a syntactic element, not in a comment or string.

Form	Line Range
[(⍺X)]	0 thru Y inclusive
[(⍺X)A]	A only
[A(⍺X)]	A thru Y inclusive
[A(⍺X)B]	A thru B inclusive

Where ⍺ is any multiline function editing action, one of

⍑ ⊥ ⌈ ⌊ ∩ ∪ ⌷ ? ~

A is line number specifier: $A \in 0, \iota Y$
B is line number specifier not less than A:

$B \in A, A + \iota Y - A$

X is a name
Y is the number of the last-defined function line

Examples

To illustrate line specifier use, the action character □ (display lines) is used.

```
      ∇F[□]                          open, display entire function
    ∇ R←F X;Y
[1]   R←INTO X⍝ NEW LINE 1
[2]  LABEL1:Y←R LINE X
[3]   ⍝ NEW LINE END OF 3
[4]   'LINE IN STRING'
[5]  LABEL2:R←Y+3
[6]   ⍝ LAST LINE
    ∇                                end of function indicator
[7]   [LABEL1+1□4]                   still open, display in range
[3]   ⍝ NEW LINE END OF 3
[4]   'LINE IN STRING'
[7]   [5□]                           display to end
[5]  LABEL2:R←Y+3
[6]   ⍝ LAST LINE
[7]   [□1]                           display only line 1
[1]   R←INTO X⍝ NEW LINE 1
[7]   [(□LINE)]                      display all containing name
[2]  LABEL1:Y←R LINE X
[7]   [3(□Y)]                        3 to end containing y
[5]   R←Y+3
[7]   ∇                              close
```

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.

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 and an execution counter is incremented. The precision of this time is 100 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 execution times for entering and properly exiting the function, and a count of the number of proper exits of the function. The display unit for these times is milliseconds and the result is rounded.

Conditions

The previously described forms apply to restrict the range of lines specified by using left and/or right line specifiers and parenthesized name qualifier.

Changing diagnostic settings affect only the specified lines. The prior settings of undesignated lines are unaffected.

Examples

```
      ∇F[□]                          display
     ∇ R←F X;Y
[1]    R←INTO X⍝ NEW LINE 1
[2]   LABEL1:Y←R LINE X
[3]    ⍝ NEW LINE END OF 3
[4]    'LINE IN STRING'
[5]   LABEL2:R←Y+3
[6]    ⍝ LAST LINE
      ∇
[7]   [⌈]                            set trace on all lines
[7]   [(⌈R)]                         set stop on lines containing R
[7]   [(∩Y)]                         set monitors on lines containing Y
[7]   ∇                              close
      □QT'F'                         query traced lines
1  1   1  1  1  1  1
      □QS'F'                         query lines with stops set
1  0   0  0  0  1  0
      □QM'F'                         query lines with monitors set
1  0   1  0  0  1  0
      ∇F[LABEL1(∪Y)]∇                reset monitors from LABEL1 with Y
      □QM'F'
1  0   0  0  0  0  0                 set monitors on lines with name LINE
      ∇F[(∩LINE)]∇
      □QM'F'                         sets are additive, exclude 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 previously described forms apply to restrict the range of lines by using left and/or right line specifiers and parenthesized name qualifier.

Examples

```
      ∇F[□]                          display
    ∇ R←F X;Y
[1]   R←INTO X⍝ NEW LINE 1
[2]  LABEL1:Y←R LINE X
[3]   ⍝ NEW LINE END OF 3
[4]   'LINE IN STRING'
[5]  LABEL2:R←Y+3
[6]   ⍝ LAST LINE
    ∇
[7]   [1(□Y)]                        from 1 thru end with y
[2]  LABEL1:Y←R LINE X
[5]  LABEL2:R←Y+3
[7]   [(?Y)]                         numbers of lines with y
0  2  5
[7]   [(□LINE)]                      with name LINE
[2]  LABEL1:Y←LINE X
[7]   ∇
```

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 previously described forms apply to restrict the range of lines by using left and/or right line specifiers and parenthesized name qualifier.

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                          open
[7]    [~6]                       delete last line
[6]    [3~4]                      delete lines 3 and 4
[4]    [0~1]                      delete line 1, locals from 0
[3]    [⎕]∇                       display
     ∇ R←F X
[1]  LABEL1:Y←R LINE
[2]    R←Y+3
     ∇
      ∇F[~]                       cannot remove template
[1]    ∇
      ∇F[⎕]∇
     ∇ R←F X
     ∇
```

EXECUTION OF DEFINED FUNCTIONS

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, but is still active. Active functions appear in the state indicator. Pendant functions are nested: each called function is nested in the environment existing at the point of call in the calling function. The top of the state indicator indicates the most nested, or local, function. Functions below it are successively more global.

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 relatively global to any function called from that function. A global name is shielded, or inaccessible, while an instance of a local use of the same name (a homonym) exists. Only local names and global names that are unshielded determine the local environment.

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 local environment is an important 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 branch control transfer, or terminate.

Normal 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. Terminate has no need for a returned value.

MULTIPLE INSTANCES

More than one instance of a function can 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. A function can include a call on another instance of itself, either directly or indirectly. The number of instances is limited by the amount of space required for each instance and the amount of space available in the workspace.

The difference between recursive and unrelated calls on a function is that for unrelated calls at least one intervening suspended function exists, as can be observed in the state indicator.

DISCARDING OUTPUT DURING DEFINED FUNCTION EXECUTION

Unwanted output display may be discarded by entering a single attention ATTN. This ATTN also requests function suspension before the next body line of an unlocked function encountered after the ATTN is recognized. →⎕*LC* resumes execution. If a request for keyboard input is received before the suspension takes effect, that suspension request is cancelled.

SUSPENDING FUNCTION EXECUTION

A suspension is a controlled interruption to processing of a defined function that occurs before a body line of an unlocked function. Suspension occurs in three ways:

The path of control reaches a body line of an unlocked function with a stop set on it

The user enters one ATTN during function execution or result display, requesting a suspension

the APL system recovers after a disconnect

The suspension request from an ATTN is not honored until the function line in progress is completed, and further all locked functions in the call sequence leading to an unlocked body line are completed. If a request for input occurs before reaching a valid suspension point, the suspension request is voided.

The result of suspension is a return to execution mode after displaying the suspension notice, typically for line 3 of the unlocked 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 global 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*.

INTERRUPTING FUNCTION EXECUTION

An Interrupt also results in suspension at some body line of an unlocked function. However may not have completed the normal processing to get there. An interrupt may occur in the middle of a function line, or even part way through a primitive function. The side-effects are uncertain of partially completed processing when the interrupt occurred. An interrupt may occur in three ways:

An error detected by APL during execution

An entry of a second ATTN before the suspension request of the first one reached a suspendable line

An entry of ⍬ (*O U T* superimposed) in response to a prompt requesting character or evaluated input

Resumption after an interrupt is the same as for suspension, although the local environment is less certain.

TERMINATING FUNCTION EXECUTION

Termination of the execution of the suspended function (and any pending its completion) may be achieved by executing the niladic →. This is also acceptable as a response to the evaluated input prompt []:. While suspended, entry of → removes one suspension level from the state indicator: entry of)*RESET* clears the state indicator.

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.

ERROR DURING EXECUTION

Any detected error during execution of an unlocked defined function yields the error message, then the function name and bracketed line number and suspension star followed by the offending line. Ellipsis may appear in the name and at either end of the display window. Then a line containing a caret pointing to the error position, and function interruption at that point. An initial ATTN in execution mode retrieves that line in the environment of the function. An error in a locked function causes similar display, but the error is attributed to the locked function name called in the suspension line, the closest unlocked function in its call path.

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 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 prefix the executable function name by 'HOW'. Space saving results from excising all the 'HOW' functions before execution, possibly by ⎕*EX* '*H*'⎕*NL* 3. 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 functionally equivalent workspaces: one for documentation, the other for execution. The documented functions can have copious comments. This documented workspace is saved. A copy of it is edited to eliminate the 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.

EXAMPLE OF FUNCTION EXECUTION, TRACE, AND EDITING

```
      ∇ AVE←VERSION AVERAGE VALUES;SUM;SIZE
[1]     →VERSION↓BRIEF
[2]     ⍝ VERSION IS INTEGER, 0=BRIEF, 1=LONG FORM; VALUES IS
        A NONEMPTY NUMERIC VECTOR
[3]     ⍝ AVE IS NUMERIC VECTOR SINGLE, THE AVERAGE OF VALUES
[4]     SUM←+/VALUES⍝ SUM OF VALUES
[5]     SIZE←⍴VALUES⍝ SAMPLE SIZE
[6]     AVE←SUM÷SIZE
[7]     →0⍝ EXIT  UNCONDITIONALLY
[8]   BRIEF:AVE←(+/VALUES)÷⍴VALUES
      ∇
      1 AVERAGE 3 4 5       ⍝ LONG FORM, LINES 1 THRU 7
4
      0 AVERAGE 3 4 5       ⍝ BRIEF FORM, LINES 1 AND 8
4
      0 AVERAGE ⍳1000
500.5
      1 AVERAGE 7           ⍝ WILL IT WORK FOR A SCALAR?

      ∇AVERAGE[4⍳6]∇        ⍝ NO, SET TRACE ON LINES 4 THRU 6
      1 AVERAGE 7
AVERAGE[4] N( ) 7
AVERAGE[5] N(0)
AVERAGE[6] N(0)

      ⍝ EMPTY RESULT, PROBLEM IS SHAPE OF SCALAR 'VALUES' IS EMPTY
      ∇AVERAGE[∈5]
[5]     SIZE←⍴VALUES⍝ SAMPLE SIZE
                ,,
[5]     SIZE←⍴,VALUES⍝ SAMPLE SIZE
[9]     [∈8]
[8]     BRIEF:AVE←(+/VALUES)÷⍴VALUES
                                ,,
[8]     BRIEF:AVE←(+/VALUES)÷⍴,VALUES∇
      AVG←1 AVERAGE 7       ⍝RETAIN VALUE IN AVG
AVERAGE[4] N( ) 7
AVERAGE[5] N(1) 1
AVERAGE[6] N(1) 7
      AVG
7
      ⎕RT'AVERAGE'          ⍝ RESET TRACE
      0 AVERAGE 7
7
      0 ⎕SS'AVERAGE'        ⍝ STOP BEFORE EXIT
      AVE                   ⍝ NO GLOBAL VALUE
*** VALUE ERROR ***
      ∨
      AVE
      100×1 AVERAGE 3 4 5   ⍝ USE RESULT IN EXPRESSION
AVERAGE[0]*
      AVE                   ⍝ LOCAL NAME FOR RESULT HAS VALUE
4
      SIZE                  ⍝ LOCAL VARIABLE HAS VALUE
3
      BRIEF                 ⍝ LABEL HAS VALUE
8
      →0                    ⍝ RESUME (EXIT FROM) SUSPENDED 'AVERAGE'
400
```

SECTION 9

ERROR REPORTS AND THEIR INTERPRETATION

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. Errors may be sensed by the system or by APL.

RESEND The system could not accept the prior entry

An APL error message 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 position of the error 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 before a ',' 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 cannot 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: a missing primitive function argument, or unmatched or mispaired parentheses or brackets (several caret marks may result).

A "CONTEXT ERROR" message indicates that a name occurs in contradiction to its current definition. An undefined name is assumed to be a variable. Three occurrences cause this error: an improper number of arguments to a defined function, requiring a result from a function that does not return one, or attempting to use a variable as though it were a function requiring arguments. Context analysis applies to the entire expression before any of it is executed.

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, 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 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 on the workspace state.

Any APL 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 established by the APL installation.
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 system 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 before a ',' 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 LENGTH ERROR	The left argument to file compress or file expand is incorrect.
FILE LENGTH LIMIT	An attempt was made to access more than 1000 components.
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.

Table 9-1 Error Reports (3)

*** Report ***	Cause
INDEX ERROR	An index into an array was out of the array bounds.
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 request was made to edit 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-SHARES LIMIT	An attempt was made to share more variables than the system will permit among all users.
SV-SPACE LIMIT	An attempt was made to use more shared variable space than the processor is permitted.
SV-USERS LIMIT	An attempt was made to start using any shared variable by a user when the system limit of shared variable users had been reached.
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.
UTIL-ACCESS ERROR	An attempt was made to offer a variable to a utility that the user is not permitted to access.
WS-ACCESS ERROR	An attempt was made to use a workspace in a manner for which access was not permitted.
WS-DUP-NAME ERROR	The)SAVE name duplicates a library workspace name.
WS-NAME ERROR	A reference was made to a nonexistent workspace, or the name was improperly formed.

Table 9-1 Error Reports (5)

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.
WS-SIZE ERROR	An attempt was made to load a workspace created on another system that is larger than the workspace size.
WS-VERSION ERROR	The workspace was created with an incompatible (later) APL system version than the version being run.

UNIMPLEMENTED CONSTRUCTS

Some constructs described in this manual are not implemented in the 2.8 release of APL/700.

1. A "SYNTAX ERROR" results from an attempted dimension selection from the anti-origin for the structure mixed primitive functions:

$\ominus[K]B$	reverse
$A\ominus[K]B$	rotate
$A/[K]B$	compress
$A\backslash[K]B$	expand
$\ominus/[K]B$	reduce
$\ominus\backslash[K]B$	scan.

2. An empty segment in the character format string gives a "FORMAT ERROR", rather than default format.

3. The Name List system function $\square NL$ does not permit specifying the value 0, meaning objects with no associated meaning. A "DOMAIN ERROR" is given instead.

4. If the result of change password on a file is required, it is the entire right argument, not just the new file name and password.

APPENDIX A

GLOSSARY

Term	Meaning
Account Name	The identification which the APL system records resources consumed by a user, see usercode.
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 contains variables, functions, groups, and state.
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 as an effective notation for algorithm description that exhibits considerable syntactic structure.
APL/700	APL enhanced for the Burroughs large system 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, or Datum	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 or axis 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 (Δ) or underscored delta (Δ) 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 a common point in a loop.
Jiggle	A terminal cursor movement (5 spaces, 2 backspaces) signifying if LOCK is ON that the execution mode prompt is deferred.
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 that the text following is a comment in execution mode 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-applied control to protect usercode, workspace, file, or function; a keyboard state.
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 primitive function(s) as arguments to produce a new function that applies to data arguments.
Orthogonal	Mutually perpendicular, or independent; referring to different dimensions of an array.
Password	User selected name for locking usercode, 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 prompts.
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 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	The values of variables and the state indicator in the workspace.
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: an external name for shared-variable reference; or keyword for APL character.
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, random link, and load expression).
Template	Specification of name and call syntax of a 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
Usercode	The name by which an account is known to the Burrough large system; synonymous with account name.
Valence	The number of arguments of a function: 0=niladic, 1=monadic, 2=dyadic; (excludes specific result).
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

The APL user generally may ignore the space required in the workspace for the desired work. However, as the number and size of retained data objects and functions increases, the available space may be inadequate for further work. A SPACE LIMIT error message will occur and corrective action is required. The way space is used, and suggestions for saving space are discussed hereafter.

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 implicit temporary results, local names on return from the function to which they are local, or a prior data object attached to a name on replacement; explicitly for other named objects by)ERASE or expunge.

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	$6+6\times R$

The space for *N* elements, regardless of shape, depends on the type:

Type	Bytes
Boolean	$6\times\lceil N\div 32$
Numeric, not Boolean	$6\times N$
Character	$6\times\lceil N\div 6$

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 function 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, at which time it reverts to a token stream.

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, or space for current value
Result	12 bytes, or space for current value
Label	18 bytes
Local function	enough for that function token stream
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.

Within the function, reassignment to any function argument changes the initial space allocation, just as assignment to 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 vector	1.0
canonic	data matrix	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 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 for 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 $6\times\lceil N\div 4$ 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. Alternatively, if "HOW" is a prefix and no other functions begin with "H", *⎕EX 'H'⎕NL 3* may be used.

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

DEFINITION OR EXAMPLE	NAME	•
LARGER OF A AND B ↔ A⌈B 7 ↔ 3⌈7 6.01 ↔ 6.01⌈6.01 ¯3 ↔ ¯3⌈¯7	MAXIMUM	⌈
SMALLER OF A AND B ↔ A⌊B 3 ↔ 3⌊7 6.01 ↔ 6.01⌊6.01 ¯7 ↔ ¯3⌊¯7	MINIMUM	⌊
1.5 ↔ ¯2+3.5 5.5 ↔ 2+3.5 ¯1.5 ↔ 2+¯3.5	ADD	+
¯1.5 ↔ 2-3.5 1.5 ↔ 3.5-2 5.5 ↔ 2-¯3.5	SUBTRACT	-
5 ↔ 4×1.25 ¯3 ↔ 6×¯.5 0 ↔ 0×¯.09	MULTIPLY	×
1.76 ↔ 3.52÷2 ¯5 ↔ 10÷¯2 4 ↔ 12÷3	DIVIDE	÷
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	|
A RAISED TO THE POWER B: 9 ↔ 3*2 1024 ↔ 2*10 2 ↔ 4*.5 ¯3 ↔ ¯27*(÷3)	POWER	*
(•B)÷•A ↔ LOGARITHM OF B FOR BASE A ↔ A•B 1.87506... ↔ 10•75 3 ↔ 2•8	LOGARITHM	•

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)		
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	CIRCULAR
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	

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	!

MONADIC SCALAR PRIMITIVE FUNCTIONS • B

NAME	DEFINITION OR EXAMPLE
CEILING	SMALLEST INTEGER NOT LESS THAN B ↔ ⌈B 4 ↔ ⌈3.141 ¯3 ↔ ⌈¯3.141 101 ↔ ⌈101
FLOOR	LARGEST INTEGER NOT GREATER THAN B ↔ ⌊B 3 ↔ ⌊3.141 ¯4 ↔ ⌊¯3.141 101 ↔ ⌊101
IDENTITY	0+B ↔ +B 3.5 ↔ +3.5 ¯3.5 ↔ +¯3.5
NEGATE	0-B ↔ -B ¯3.5 ↔ -3.5 3.5 ↔ -¯3.5
SIGNUM	SIGN OF B: 1 ↔ ×7.2 0 ↔ ×0 ¯1 ↔ ×¯3
RECIPROCATE	1÷B ↔ ÷B .5 ↔ ÷2 ¯2 ↔ ÷¯.5
MAGNITUDE	ABSOLUTE VALUE OF B ↔ |B 9.5 ↔ |9.5 9.5 ↔ |¯9.5 0 ↔ |0
BASE E POWER	(2.71828...)*B 4 ↔ *1.386294361... 20.0855... ↔ *3
BASE E LOGARITHM	(2.71828...)•B N ↔ *•N ↔ •*N 1.386294361... ↔ •4 ¯.693147... ↔ •.5
NOT	0 ↔ ~1 1 ↔ ~0 BOOLEAN DOMAIN (0 OR 1)
PI TIMES	B×3.14159... ↔ ○B 6.283185... ↔ ○2
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:

```
1 2 3 4 5 ↔ V

11 12 13 14
21 22 23 24 ↔ M
31 32 33 34

111 112 113 114
121 122 123 124
131 132 133 134
                ↔ T
211 212 213 214
221 222 223 224
231 232 233 234
```

NAME	FORM	DEFINITION	EXAMPLES
RESHAPE	A⍴B	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 ↔ ⍴A⍴B	5 5 5 ↔ 3⍴5 ,1 ↔ 1⍴V 2.5 ↔ (⍳0)⍴2.5 8.6 ¯3.1 1 2 3 4 ↔ 3 2⍴V 5 1
RAVEL	,B	THE DATA OBJECT B IS RESHAPED INTO A VECTOR. ,B ↔ (×/⍴B)⍴B	11 12 13 14 21 22 23 24 31 32 33 34 ↔ ,M 1⍴8.6 ↔ ,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 ↔ ⊖M 11 12 13 14
	⊖[K]B	⊖[K]B ↔ ⌽[1+(⍴⍴B)-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+(ρρB)-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 1ρ8 ↔ 0 0 0↓8 0 1 0 1ρ¯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 ALREADY 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.	
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	⍺	[⍺*A*]	SIMILAR TO FULL-EDIT EXCEPT SINGLE INSERTION BEFORE TEXT OF LINE *A* IS ASSUMED.	
DIRECT-PREFIX	⍺	[⍺*A*]*T*	THE TEXT *T* IS INSERTED BEFORE THE TEXT OF LINE *A*.	
SUFFIX-EDIT	⍵	[⍵*A*]	SIMILAR TO FULL-EDIT EXCEPT SINGLE INSERTION AFTER TEXT OF LINE *A* IS ASSUMED.	
DIRECT-SUFFIX	⍵	[⍵*A*]*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*

FORM	*LINE DOMAIN*
[o]	0 *THRU* Y
[o*A*]	*A ONLY*
[*A*o]	*A THRU* Y
[*A*o*B*]	*A THRU B*

QUALIFIED: *LINES CONTAINING NAME X IN DOMAIN*

FORM	*LINE DOMAIN*
[(o*X*)]	0 *THRU* Y *CONTAINING X*
[(o*X*)*A*]	*A ONLY IF CONTAINING X*
[*A*(o*X*)]	*A THRU* Y *CONTAINING X*
[*A*(o*X*)*B*]	*A THRU B CONTAINING X*

o *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	⊤	*FORM OF DISPLAY (DURING EXECUTION):* *F*[*N*]*K*(*S*)*V* *F = FUNCTION NAME* *N = LINE NUMBER* *K = VALUE TYPE* *N - NUMERIC* *B - BOOLEAN* *C - CHARACTER* *S = VALUE SHAPE* *V = VALUE* *LINE* [0] *TRACES FUNCTION RETURN. OTHER LINES TRACE LEFTMOST VALUE, IF ANY.*	[*Z*]
CLEAR-TRACE	⊥		
SET-STOP	⌈	*FORM OF DISPLAY (DURING EXECUTION):* *F*[*N*]* *F = FUNCTION NAME* *N = LINE NUMBER* *LINE* [0] *STOPS BEFORE RETURN.*	
CLEAR-STOP	⌊		
SET-MONITOR	∩	*INITIATE COLLECTION OF STATISTICS.*	
CLEAR-MONITOR	∪	*LINE* [0] *COUNTS THE NUMBER OF TIMES THE FUNCTION IS EXECUTED.*	
DISPLAY-LINES	⍞	*FORM OF DISPLAY:* *HEAD -* ∇*H* *BODY -* [*N*] *T* *TAIL -* ∇ *H = HEADER* *N = LINE NUMBER* *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 6000 / B 7000

APL / 700 SYNTAX SUMMARY

ARGUMENT LEGEND

A	*any type*
B	*Boolean type, 0 or 1*
C	*character type*
D	*decimal, numeric type*
F	*function or file name as character string*
I K	*integer type*
L	*list*
M N	*name, identifier*
R	*result*
T	*character text*

SYSTEM COMMANDS

** if command is niladic, returns setting*
X is optional field X, | is alternative
wi is (usercode) name [passw]
access-set is OWN or NORMAL or PUBLIC | NOSAVE | NOCOPY | NOLOAD | NOREPLACE

session and terminal controls

	)BLOT	
	)ON	*usercode [passw] chargec*
*	*)KEYWORDS*	*ON or OFF*
*	*)LOCK*	*ON or OFF*
	)OFF	*[oldpassw/newpassw]*
	)COFF	*[oldpassw/newpassw]*
	)USE	*mcsname*
*	*)WIDTH*	*30 thru 250*
*	*)TABS*	*0 thru 30*

clear-workspace defaults

*	*)CLEARACCESS*	*access-set*
*	*)DIGITS*	*1 thru 12*
*	*)FUZZ*	*0 to 1*
*	*)ORIGIN*	*0 or 1*
*	*)SEED*	*nonnegative integer*
*	*)SYMS*	*16 thru 1024*

workspace controls

	)CLEAR	*16 thru 1024*
	)LOAD	*wi*
*	*)ACCESS*	*access-set*
	)ERASE	*name-set*
	)RESET	
	)COPY	*wi name-set*
	)PCOPY	*wi name-set*
	)SAVE	*wi*
	)DROP	*own-wi*

name display

)LIB	*name*
)FILES	*name*
)FNS	*name*
)VARS	*name*
)GRPS	*name*
)GRP	*name*
)SI	

group controls

)ATTACH	*groupname name-set*
)DETACH	*groupname name-set*

messages

	)TO	*usercode text*
	)OPR	*text*
*	*)RECEPTION*	*ABSENT or NORMAL or URGENT*

SYSTEM VARIABLES

⎕CT	*comparison tolerance*
⎕IO	*index origin*
⎕LX	*load expression*
⎕PP	*print precision*
⎕RL	*random link*
⎕	*evaluated input*
⎕←	*explicit output*
⍞	*character input*
⍞←	*set prompt*

SYSTEM FUNCTIONS

@ dyadic = specify, monadic = all

function transformations

	⎕CR F	*canonic represent*
	⎕VR F	*vector represent*
	⎕FX C	*fix into function*
	⎕LF C	*lock functions*
	⎕UF C	*unlock unsealed functions*

name

C	*⎕NL I*	*name list @*
	⎕NC C	*name classification*
	⎕EX C	*expunge*

status inquiry

⎕AI	*account information*
⎕AN	*account name*
⎕FA	*file availability*
⎕LA	*library availability*
⎕LC	*line counter*
⎕NA	*name availability*
⎕NEWS	*sign-on news*
⎕PT	*print tabs*
⎕PW	*print width*
⎕SA	*shares availability*
⎕TS	*time stamp*
⎕UL	*user load*
⎕UN	*utility names*
⎕WA	*workspace availability*
⎕WI	*workspace identification*

```
      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  ⎕MC F     monitor counts @
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
   ⎕F        formfeed
   ⎕I        idle
   ⎕L        linefeed
   ⎕M        margin
   ⎕N        null
   ⎕R        return
   ⎕T        tab
   ⎕A        alphabet 'AB...Z'
   ⎕D        digits '01...9'
   ⎕AV       atomic vector
```

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
```

CONTROL STRUCTURES

```
            call defined function F
   F           niladic
   F A         monadic
A  F A         dyadic

          sequence of execution
       functions right-to-left in line
       lines in sequence unless →
   → I         branch
   →           terminate
   N:          label

   ( )     function precedence
   ⋄       list separator
   ⍝       comment
```

HEADER FOR DEFINED FUNCTION *F*

```
      template
      template; local-names-list

no result    result          template
   F         R ← F           niladic
   F N       R ← F N         monadic
 M F N       R ← M F N       dyadic
```

FUNCTION EDITING ACTIONS

P, Q are integer I, label, or label±I

```
   ∇ F       define
   ∇ F       open
   ⍫ F       open (locked, unsealed)
   ∇         close
   ⍫         close (locked)
   [P] T     replace
   [↑] T     append (before line 1)
   [↓] T     append (after last)
   [↑Q] T    insert (before)
   [↓Q] T    insert (after)
   [∈Q]      full edit
   [⍺Q]      prefix edit
   [ωQ]      suffix edit
   [⍳Q]      inject edit

   @     multiline group actions
   ⎕         display lines
   ?         display line numbers
   ~         delete lines
   ⍑         set trace
   ⊥         reset trace
   ⌈         set stop
   ⌊         reset stop
   ∩         set monitor
   ∪         reset monitor

with name N  action @ on lines   all
   [(@N)]       0 thru last      [@]
   [P(@N)]      P thru last      [P@]
   [(@N)Q]      Q only           [@Q]
   [P(@N)Q]     P thru Q         [P@Q]
```

TRANSACTION EDITING

```
      meaning of attention
initial:    enter edit cycle
embedded:   correct typing error
terminal:   display next phrase

      edit control characters
   /        delete
   .        mark phrase
   .T       insert text
```

SELECTION AND ASSIGNMENT

f is primitive scalar dyadic function

N[L]	select
N ← A	replace
N[L] ← A	insert
N f← A	modify
N[L] f← A	modified insert

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 function I
! D	factorial
D ! D	combinatorial

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 — 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 — 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

PRIMITIVE OPERATORS

f and g are primitive scalar dyadic functions

```
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

A f.g A    inner product
```

FORMAT PRIMITIVE FUNCTIONS

```
   ▼ A       implicit specification
K  ▼ D       numeric specification
     K         in pairs w d
       w       width
       d       decimal places:
                 <0 floating point
                 =0 integer
                 >0 fixed point
C  ▼ L       character specification
     L         expression or (list)
     C         format:  s or s;...;s
       s       segment: g or g,...,g
       g       group:   c or r(c)
       r       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

FILE FUNCTIONS

file name F is (usercode) name [lock]
X is optional X

```
    ⍁ F      create file
N   ⍁ F      rename file F to N
    ⍂ F      destroy file

    ⍁ F      hold file
    ⍂ F      free hold on file
    ⍈ F      detach from 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      value component map
    ⍈ F      null component map

    ⍁ F      test file status, result is
    0          nonexistent
    1          inactive
    2          active
    4          unavailable
    5          held

I   ⍁ F      query file
    1          bytes in use
    2          maximum bytes
    3          components
    4          modified
    5          cycle
    6          users
    7          last update

  ⍁ I        interrogate file system
    1          file users
    2          files active
    3          maximum file users
    4          maximum files active
```

IDENTIFIER *letter, underscored letter, Δ or Δ, followed by 0 or more of the above, _ or the digits*

CONSTANT
```
  'KEN''S'      character ken's
¯1 1.2 3.4E¯5   numeric
```

1079936-2

APPENDIX D

AIDS IN PUZZLING CIRCUMSTANCES

The symptoms of some infrequently occurring events are described here, and your corrective actions are suggested. The underlined headings denote general symptoms. Then more detailed conditions may be given to help diagnose the events. Likely explanations and recommended corrections are shown indented.

Long Wait For Response

Your terminal may be awaiting an entry:

> Try making an entry.

Output was partially completed before a print pause:

> Large output objects are released in chunks for display. Patience! The system may be heavily loaded or the system may be recovering from trouble. Recovery may cause an implicit)COFF. After signing on again your work usually may be resumed by →⎕*LC*.

The function may be in a loop:

> Stop output and request suspension with a single ATTN. If this fails because the function is locked, a second ATTN can be used to request interruption. After suspension the state indicator may be examined. Execution may be resumed by →⎕*LC*.

You just sent a message:

> You must wait for the addressee's terminal to reach an acceptable reception point: An ATTN entered before your message is delivered will withdraw it.

You may have)LOCK ON:

> In)LOCK ON state APL defers execution mode prompts. The "jiggle" (5 spaces, 2 backspaces) display indicates when the deferral is starting. The keyboard is locked. A subsequent ATTN elicits one execution mode prompt. If also)RECEPTION is set URGENT then all prompts will be deferred.

Unwanted Output or Processing

Unwanted output:

A single attention ATTN discards output and requests suspension.

Processing generally leads output. Thus a suspension may occur several lines beyond the line that generated the discarded output.

A request for suspension in a locked function discards the output but the function continues execution. A request for input before the suspension point is reached nullifies that suspension request.

Suspension followed by continuation:

A suspension occurs immediately prior to the next body line of an unlocked function that is encountered during execution.

If complete actions on data objects or file components occur on a single line, then suspension will not cause damage from partially changed values. The local state may be examined during a suspension.

Execution may be resumed by →⎕*LC*.

Interruption with no intention of continuing:

A second ATTN requests an interrupt if the first ATTN has not already resulted in suspension. Interruption stops processing as soon as it is recognized by the system (even in the middle of a line or function), and moves out to the first suspension point.

Interrupt may also be caused by entry of ⍬ (the superimposed characters *O U T*) in response to either a ⎕: or ⍞ input prompt.

Termination of the current execution state:

Execution of a function line containing niladic →, or entry of → in response to a ⎕: prompt causes the state indicator to be cleared to the next prior suspended function.

Unexpected Prompt

The character input prompt was assigned a non-empty character vector. ⍞ may be in the local name list of the function that accepts the character input. Clear by exiting that function or by ⍞←''. Note that ⍞←'⎕:',⎕*L*,' ' is a valid way to simulate the evaluated input prompt by a character input request ⍞.

Evaluated Input Prompt

Looping input request prompt:

The program logic may require a particular entry value to escape. For the ⍞: prompt, a (non-blank) entry of a valid APL expression is required, else the ⍞: prompt is repeated. An interrupt may be achieved by ⍓ (*O U T* superimposed). An abort may be achieved by '→'. For the ⍞ prompt, ⍓ causes an interrupt.

Definition Mode

Definition error:

The name cannot be used as a function at this time since it is otherwise in use, or the function is locked.

The function cannot be examined:

The function does not exist.

The function is locked: You may unlock it by ⍒*FV*.

The function is sealed: the function cannot be unlocked.

The function is local: definition mode only applies to global functions. System functions ⎕*VR* and ⎕*CR* apply to the most local instance of a name.

The existing function does not appear to exist:

The global function name may be shielded by a local name.

The function cannot be edited in definition mode:

The function is in the state indicator but not the most recently suspended:)RESET or → to clear sufficient levels of function suspension. Function line 0 cannot be edited if the function is suspended.

Display

Line folding:

Output line length N is longer than printer width, ⎕*PW*. Increase by)WIDTH N; or format output to fit within ⎕*PW*.

Irregular columns:

Terminal physical tabs may be set to a non-uniform interval or an interval different from ⎕*PT*. Repair by setting physical tabs to a uniform interval N.)TABS N informs the system of this interval. N = 0 causes tabs to be ignored.

Variable Value

Assignment may have occurred as a side effect of execution of some defined function. Make the name local to that function.

A local name may be shielding a global variable of the same name.

Elaboration Order of Dyadic Function Arguments

If the right argument *R* to a dyadic function *L F R* is just a variable name (not an assignment to it or itself a left argument of another function) then any change to the value of that variable as a result of elaborating the left argument will be the current value used when finally *F* is evaluated:

```
      A←3
      (A←5)=A
1
```

If the above sequence is used but the prior value is desired, a temporary value must be developed for the right argument by some auxiliary function that preserves the property important to the desired function (in this case the value property):

```
      A←3
      (A←5)=+A
0
      (A←5)=A←3       ⍝ VALUE OF THE 'NAME' 3 DID NOT CHANGE
0
      (A←5)=A←2+1     ⍝ A IS THE RIGHTMOST NAME
1
```

Scalar vs. Single Element Array

Displays of all singles with the same element value appear the same regardless of rank. Some structure functions are restricted in the ranks of arguments. For example, grade or index will not accept scalars. A vector can be formed from a scalar by ravel.

A scalar can be formed from the first element of a data object *D* by (⍳0)⍴*D*. An alternative is (,*D*)[⎕*IO*] or just *D*[⎕*IO*] if *D* is a vector.

Space Limit

The workspace is full. There may be unwanted suspended functions in the state indicator, possibly caused by repeated function initiation while earlier instances were suspended, or by excessive recursion:)RESET to clear.

Unnecessary functions or data objects may exist: eliminate by ⎕*EX* expunge or)ERASE. See Appendix C for space saving techniques.

System Commands

Name erase:

Erase by)ERASE nameset only applies to global names. Erase is prevented for any function name in the state indicator, for any local name, and for a variable name that is the right argument of a dyadic function for which the left argument invoked a function that became suspended (or tried to expunge the name). □*EX* can be used for local names other than labels.

Name application:

The name set argument to any of)COPY)PCOPY)ERASE)ATTACH)DETACH applies only to global names. If any name is currently local, that local name is unaffected.

Inability to access workspace:

Access may be restricted by OWN, NOLOAD and/or NOCOPY. The workspace owner must relax these by)ACCESS PUBLIC or NORMAL,)ACCESS LOAD,)ACCESS COPY.

Inability to save workspace:

Access may be restricted by NOSAVE. If you own the workspace, one save is allowed after)ACCESS NOSAVE. Save may also be restricted by NOREPLACE. Either)ACCESS REPLACE or)DROP wi is required before)SAVE wi is acceptable.

System Variables

Value:

The system commands to set the default clear workspace controls do not alter the values of the system variables in the current workspace, even if it had just been cleared. The system variables must be used for local alteration to the default values.

Index Error:

The workspace or function may have the wrong origin: localize □*IO*.

Unexpected Comparison Result:

The comparison tolerance □*CT* may be too large.

```
      □CT←0.5
      1=2
0
      2=1
1
```

Computational round-off error from finite precision computation may cause comparison to fail if the comparison tolerance is too small.

```
      ⎕CT←0
      1=999×÷999
0
```

If no round-off error is expected, some primitive functions are more efficient if ⎕*CT*←0.

Result rounding:

Print precision ⎕*PP* affects the number of significant digits displayed for non-integer numbers without an overriding format. It has no effect on integers. Rounding occurs, based on the internal representation of non-integer numbers.

Repeatable "random" numbers:

The seed for the random number generator is part of the workspace. For a just loaded workspace its value will be the same as when the workspace was last saved. The seed may be explicitly set by ⎕*RL*← N.

Miscellaneous

System ignores an ATTN, or an entry ending with RETN:

Communication is broken or uncoordinated, or the system is unresponsive. Try: (1) another RETN or ATTN, (2) switching the terminal to local, then back to communicate (however labeled, on the terminal in use), (3) breaking the connection momentarily by lifting the telephone handset from the acoustic coupler, (4) hanging up and redialing, (5) redialing to a different number. All of these procedures are external to APL. None are assured to work. A character error may have occurred during transmission. The system may be unavailable or down. If so, try again later.

Abnormal Interruption

A transmission error may have caused a spurious ATTN. A stop may have been set on the function line at which the suspension occurred. Reset the stop if any. Resume by →⎕*LC*.

*** CHARACTER ERROR ***:

The prior entry was improperly received. Either reenter the line, or use line editing to repair the prior entry. Squish quads ⎕ indicate unrecognizable characters.

*** ERROR message ***:

APL errors interrupt processing. Error message displays are surrounded with asterisks. Additional details on the meaning of these messages is contained in Section 9.

Failure message:

An APL system failure. Please report it along with a description of your actions immediately prior to the failure to your system representative.

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.

A

Absolute value | 5-8
Accent, ASCII
    aigu ' (see Quote) 3-15, 6-10
    grave ` 3-15, 6-10
Access
    command)*ACCESS* 3-8
    shared variable 6-13
    to APL system 2-6
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)
Activation of workspace 3-7
Active
    file 7-2
    function 8-18
    workspace 1-8
Add + 5-8
Additive identity + 5-8
Along dimension 4-2
Algorithm 8-1
Alpha ⍺ 8-10
Alphabet ⎕*A* 6-9
Alternating
    product ≠/ 5-23
    sum +/ 5-23
Ampersand, ASCII & 3-15, 6-10
And, logical ∧ 5-13
APL
    character set 2-4
    keyboards 2-2
    MCS identifier *?APL* 2-6
Apostrophe (see Quote)
Arccos ¯2○ 5-14
Arccosh ¯6○ 5-14
Arcsin ¯1○ 5-14
Arcsinh ¯5○ 5-14
Arctan ¯3○ 5-14
Arctanh ¯7○ 5-14
Areas (see Availabilities)
Arguments
    conformability (see Coercion)
    definition 1-5
    in function header 8-1
    left and right 4-7, 8-1
    number of 4-7
    of operators 5-19
Arithmetic
    functions 5-8
    sequence 5-31
Arrays 4-2
Arrow characters
    down ↓ 5-40, 7-7
    left ← 5-4
    right → 8-3, 8-20
    up ↑ 5-40, 7-7
ASCII characters 3-15, 6-10
Assignment ← 5-4
Asterisk * 5-10
At sign, ASCII @ 3-15, 6-10
Atomic vector ⎕*AV* 6-9
Attach to group command)*ATTACH* 3-10
Attention ATTN
    discard output 8-19, D-2
    editing prior entered expression 2-9
    embedded 2-9
    entry typing error correction 2-9
    function interruption 8-20, D-2
    function suspension 8-9, D-2
    initial 2-8, 2-9
    key 2-3
    LOCK ON break 3-14
    output discarding 8-19
    terminal 2-8, 2-10

Availability
file □FA 6-12
library □LA 6-12
name □NA 6-12
shares □SA 6-12
workspace □WA 6-12
Average 8-2, 8-22

B

Backslash (see Slope)
Backspace
character □B 6-9
correct entry errors 2-9
key BKSP 2-3
Bar - 5-8
overbar ¯ 4-3
underbar _ 4-6
Base ⊥ 5-46
e logarithm ⍟ 5-10
e power * 5-10
jot ⍤ 5-54
value ⊥ 5-46
Beta function 5-16
Binomial (see Combinations)
Blank as separator 4-9
Blot command)BLOT 3-3
Body of defined function 8-2
Boolean
functions 5-13, 5-42
type 4-4
Braces ⊧ ⊨ 2-4, 3-15, 6-10
Brackets [ ]
after argument
index list 4-8, 5-2
after function symbol
dimension selector 4-8
prompt 8-2
suspension notice 8-19
Branch →
conditional 8-4
decision 8-4
examples 8-4
iterative 8-4
looping 8-4
unconditional 8-4
niladic (see Terminate) 8-4, D-2
no argument (see Terminate) 8-4
Break key (see Attention)
Built-in functions (see Functions, primitive)
Bytes, space measure 6-12

C

Calculator mode, (see Execution mode) 1-4
Call
defined function 8-18
recursive 8-19
Canceling a line 2-8
Canonic representation □CR 6-4
Cap ∩ 5-42, 8-14
jot ⍝ (see Comment)
Caret characters
as error pointers ^ v 9-1
down v 2-9, 9-2 (see Or)
left < 2-8 (see Less than)
not down ⍱ (see Nor)
not left ≤ (see Less or equal)
not right ≥ 2-8 (see Greater or equal)
not up ⍲ (see Nand)
right > (see Greater than)
up ^ 9-2 (see And)
Carrier return, RETN (see Return)
Catenate , 5-32
Ceiling ⌈ 5-7, 8-14
Cent sign ¢ 2-4, 6-10
Chaining arrays (see Catenate)
Change password
account 3-2
file □ 7-3
usercode 3-2
workspace 3-7
Character
alphabet □A 6-9
argument to primitive function 4-3
ASCII 3-15, 6-10
atomic vector □AV 6-11
constant 4-4
data displayed 4-3
data type 4-3
digits □D 6-9
error 2-10
format ⍕ 5-62
input □ 4-11
prompt □ 1-4, 4-11, 8-2
interrupt ⍎ 4-12
quote ' 4-3
set
APL 2-4, 6-11
ASCII 3-15, 6-10
atomic vector □AV 6-11
keywords 2-4, 3-15, 3-16
string 4-4
type 4-3
mixed with numeric output ; 4-8
vectors 4-3

Character names 2-4
accent
aigu ' 3-15, 6-10
grave ` 3-15, 6-10
alpha ⍺ 8-10
alphabet 2-4
ampersand & 3-15, 6-10
and ∧ 5-13
apostrophe (see quote) ' 4-3
arrow,
down ↓ 5-40, 8-8
left ← 5-4
right → 8-3, 8-20
up ↑ 5-40, 8-8
asterisk (see star) *
at sign @ 3-15, 6-10
braces { } 3-15, 6-10
brackets [ ] 4-8, 5-2, 8-2
box (see quad) ⍞
cap ∩ 5-42, 8-14
ceiling ⌈ (see upstile)
cent ¢ 2-4
circle ○ 5-14
bar ⊖ 5-34
slope ⍉ 5-36
star ⍟ 5-10
stile ⌽ 5-34
circumflex ^ 3-15, 6-10
colon : 4-11, 8-2
comma , 2-9, 5-32, 8-10
cup ∪ 5-42, 8-14
del ∇ 8-6
stile ⍛ 5-44
tilde ⍢ 8-6
delta ∆ 4-6
stile ⍋ 5-44
underbar ⍙ 2-4, 4-6
diamond ⋄ 6-10
dieresis ¨ 2-4, 3-15
digits 2-4
divide ÷ 5-8
dollar $ 3-15, 6-10
domino ⌹ 5-50
dot . 2-9, 4-3, 5-20, 5-26, 8-10
down
arrow ↓ 5-40, 7-7
stile ⌊ 5-7, 8-14
epsilon ∈ 5-42, 8-10
equal = 5-12, 5-13
exclamation ! (see quote dot)
floor ⌊ (see downstile)
grave ` 3-15, 6-10
hash # 3-15, 6-10
greater or equal ≥ 5-12, 5-13
greater than > 2-8, 5-12, 5-13
I-bar ⌶ 6-22
iota ⍳ 5-31, 8-10

Character names (cont)
jot ∘ 5-20
base ⍤ 5-54
cap ⍝ 4-10, 8-2
top ⍥ 5-55
less or equal ≤ 5-12, 5-13
less than < 2-8, 5-12, 5-13
letter 2-4
underbar 2-4
log ⍟ 5-10
nand ⍲ 5-13
national 1 thru 6 2-4
negative ¯ 4-3
nor ⍱ 5-13
not ~ 5-13
not equal ≠ 5-12, 5-13
null ∘ (see jot)
omega ⍵ 8-10
or ∨ 2-9, 5-13, 8-11
paragraph ⍝ 3-15, 6-10
parentheses
left (4-7, 8-12
right) 3-1, 4-7, 8-12
percent % 3-15, 6-10
period . (see dot)
plus + 5-8
quad ⍞ 4-11, 8-16
and ⍲ 7-9
circle ⍜ 7-6
del ⍫ 7-3
delta ⍒ 7-3
divide (see domino)
down arrow ⍗ 7-7, 7-9
equal ⍞ 7-10
greater than ⍞ 7-5
jot ⍜ 7-11
left arrow ⍞ 7-4
less than ⍞ 7-5
not equal ⍞ 7-10
or ⍱ 7-9
quote ⍞ 4-11
right arrow ⍞ 7-4
slash ⍁ 7-8
slope ⍂ 7-8
tilde ⍞ 2-4
up arrow ⍞ 7-7
query ? 5-45, 8-16
quote ' 4-3
dot . 5-16
quad ⍞ (see quad quote)
rho ⍴ 5-30
semicolon ; 4-8, 4-11, 5-55, 8-1
slash / 2-9, 5-38, 8-10
bar ⌿ 5-38
slope \ 5-38
bar ⍀ 5-38
star * 5-38, 8-11
stile | 5-8

Character names (cont)
subset ⊂ 5-42
superset ⊃ 5-42
tack
down ⊥ (see base)
left ⊣ 2-4, 6-10
right ⊢ 2-4, 6-10
up ⊤ (see top)
tilde ~ 5-13, 5-42, 8-17
times × 5-8
top ⊤ 5-48
jot ⍤ 5-55
underbar, underscore _ 2-4, 4-6
alphabet 2-4, 4-6
delta ⍙ 4-6
unequal ≠ (see not equal)
up
arrow ↑ 5-40, 7-7
stile ⌈ 5-7, 8-14
Check protector 5-65
Circle ○ 5-14
bar ⌽ 5-34
star ⍟ 5-10
slope ⍉ 5-36
stile ⌽ 5-34
Circular functions ○ 5-14
Circumflex, ASCII ^ 3-15, 6-10
Clear
access command
)*CLEARACCESS* 3-8
file function ⍞ 7-7
workspace
command)*CLEAR* 3-4
default controls 3-4
Close
bracket] (see brackets)
function action ∇ 8-6
lock function action ⍫ 8-6
Coefficients of polynomial 5-47
Coercion 5-17
Collating sequence 5-31
Colon : 4-11, 8-2
Combinations ! 5-16
Combinatorial, generalized ! 5-16
Comma ,
catenate 5-3
edit text insert 2-9, 8-10
laminate 5-32
not in single number 4-4
ravel 5-32
Command (see System command)
Comment ⍝ 4-10, 8-2
Comparison tolerance
default command)*FUZZ* 3-5
variable ⎕*CT* 6-2
Complement, logical ~ 5-13
Complete Beta function 5-16
Component
of file 7-1
of list 4-8
Composite functions (see Operators)
Compress / ⌿ 5-38
file components ⍁ 7-8
Computer time 6-12
Concurrent process (see Shared variable)
Conditional branch → 8-4
Conformable arguments 5-17
Conjugate + 5-8
Connect time 6-12
Connecting with the APL/700 system 2-6
Constant 1-5, 4-5
Continuation indicator ← 2-8
Continue off command)*COFF* 3-2
implicit, after disconnect D-1
Control structures 1-7, 4-7, 8-18
Conversion to other type
character to numeric 5-54
numeric to Boolean B-4
numeric to character 5-55
Coordinates of an array (see Dimensions)
Copy
command)*COPY* 3-7
protect command)*PCOPY* 3-7
workspace access 3-8
Corner element 4-2
Correction
defined function 8-10
typing error 2-9
Cosh 6○ 5-14
Cosine 2○ 5-14
Coupling degree 6-16
Create file ⍐ 7-3
Cup ∪ 5-42
Currency symbols
cent ¢ 2-4, 6-10
dollar, ASCII $ 3-15, 6-10
Cursor 1-4
Curve fitting 5-53

D

Dash - (see Bar) 5-8
Data Communications Processor (DCP) 2-1
Data entry mode 1-4

Data object, data structure 4-2
    character 4-3
    display forms 4-3
    list 5-62
    mixed types 4-8
    numeric 4-4
    tests of properties 4-5
Date, in timestamp ⎕*TS* 6-12
Deal ? 5-45
Deblank 5-35
Decimal point . 4-4
Decision branching → 8-4
Decode function (see Base value)
Decorator 5-65
Default
    clear workspace controls 3-4
    format ⍕ 5-58
Deferred prompts 3-14, D-1
Define header action
Defined function editing actions
    action specifier 8-5
    Compared to primitive 8-1
    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 line
        before ↑ 8-8
        after ↓ 8-8
    multiline group specifier 8-12
    name qualified 8-12
    prefix edit ⍺ 8-10
    prompt 1-4
    replace [ ] 8-8
    reset
        monitor ∪ 8-14
        stop ⌊ 8-14
        trace ⊥ 8-14
    set monitor ∩ 8-14
        stop ⌈ 8-14
        trace ⊤ 8-14
    suffix edit ⍵ 8-10
    unqualified 8-12
Defined functions by user
    arguments 8-1
    body 8-2
    branch → 8-3
    canonic represent ⎕*CR* 6-4
Defined functions by user (cont)
    comment 8-2
    definition 8-1
    display line folding 8-22
    documentation 8-21
    editing actions (see Function
      editing actions)
    editing as data 8-21
    examples 8-2, 8-22
    execution (see Execution of
      defined functions) 8-18
    fix ⎕*FX* 6-4
    header 8-1
    line
        numbers 8-5
        renumbering 8-5
        zero 8-1
    local names 8-1
        list 8-1
    list command)*FNS* 3-9
    name 8-1
    name list ⎕*NL* 3 6-5
    nested 8-18
    recursive 8-19
    syntax 8-1
    template 8-1
    terminate → 8-3
    transformations 6-4
    vector represent ⎕*VR* 6-4
    valence 8-1
Definition and editing mode
    function 8-1
    inoperable D-2
Degree of coupling 6-16
Del ∇ 8-6
    stile ⍒ 5-44
    tilde ⍫ 8-6
Delay ⎕*DL* 6-8
Delete
    file (see Destroy) 7-3
    line action ~ 8-17
    name (see Erase, Expunge)
    workspace (see Drop) 3-8
Delta ∆ 2-3, 4-6
    stile ⍋ 5-44
    underbar ⍙ 2-4, 4-6
Desk calculator mode (see Execution
  mode)
Destroy file ⍗ 7-3
Detach
    from file ⍇ 7-9
    group command)*DETACH* 3-10
Diagnostics 6-6, 8-14
Diagonal, selecting from
  an array 5-36

Diamond * 2-4, 6-10
Dieresis ¨ 2-4, 6-11
    for ASCII double quote 3-15
Difference
    arithmetic - 5-8
    set ~ 5-42
Digits command)*DIGITS* 3-5
    in names 4-6
    precision ⎕*PP* 6-2
    system function ⎕*D* 6-9
Dimension
    function (see Shape)
    selector 5-19, 5-29
Dimensions 4-2
Discard output, one ATTN 8-19, D-2
Display
    array 4-3
    defined function 8-16
    discard unwanted 8-19
    empty vector 4-3
    explicit output 4-11
    form 4-3
    fractional number 4-4
    heterogeneous output 4-11
    implicit output 4-11
    irregular columns D-3
    line numbers action ? 8-16
    lines action ⎕ 8-16
    matrix 4-3
    number in E notation 4-4
    prompts 1-4
    result of computation 4-3
    significant digits
        default)*DIGITS* 3-5
        display 4-4
        print precision ⎕*PP* 6-2
    value of expression 4-3
    vector 4-3
Distinguished names (see System functions, System variables)
Divide ÷ 5-8
    by zero 5-9
Documentation 8-21
Dollar sign, ASCII $ 3-15, 6-10
Domain
    of function argument 1-5
    of numbers 4-4
Domino ⌹ 5-50
Dot .
    decimal point 4-3
    edit phrase mark 2-9, 8-10
    inner product 5-26
    outer product 5-20
Double quote, ASCII ¨ 3-15
Down
    arrow ↓ 5-40, 8-8
    caret, error ∨ 9-1
    stile ⌊ 5-3, 8-14
Drop ↓ 5-40
    file components ⌹ 7-7
    function line ~ 8-17
    name (see Erase, Expunge)
    workspace command)*DROP* 3-8
Dyadic
    function 4-7, 8-1
    identity elements of scalar functions 5-28
    transpose ⍉ (see Permute dimensions)

E

e, base of natural logarithm 5-10
E notation for numbers *E* 4-4
Edit
    characters / . , 2-9, 8-10
    environment 2-11
    function 8-10, 8-21
    invocation 2-9
    precedence levels 2-11
    prior expression 2-9
    system function ⎕*ED* 6-8
Elaboration of expression 4-7, D-4
Element of a data object 4-2
Empty
    array 4-4
        reduction along 5-28
    branch → (see Terminate) 8-3
Endless loop D-1, D-3
Encode (see Represent) 5-48
Entry
    editing 2-9
    length 2-5
    numbers 4-4
Epsilon ∈ 5-42, 8-10
Equal = 5-12
Erase
    dynamic expunge ⎕*EX* 6-5
    file (see Destroy)
    names command)*ERASE* 3-9
    typing error 2-9
    workspace (see Drop command)
Error
    descriptions 9-1
    during execution 8-21
    in a defined function 9-2
    in expression entry 4-9
    reports, table 9-4
    system function ⎕*ER* 6-8
Escape from input prompt
    character ⍞ 4-12, D-2
    evaluated → ⍞ 4-12, D-2
Evaluate ⍎ 5-5
Evaluated input ⎕ 4-11
    interrupt ⍞ 4-12, D-2
    terminate → 4-12, D-2

Evaluation of expressions 4-7
Exclamation ! 5-16
Exclusive or ≠ 5-13
Execute (see Evaluate) 5-54
Execution
    controls 6-8
    mode 1-4
    state 2-1
Execution of defined function
    active function 8-18
    call 8-18
    control flow 8-18
    diagnostics 6-6, 8-14
    discard output 8-19
    dynamic expunging 8-18
    environment 8-18
    global name 8-18
    instance 8-18
    interrupt 8-20
    local name 8-18
    monitor 6-6,8-14
    multiple instances 8-19
    name scope 8-18
    nested calls 8-18
    pendant 8-18
    recursive calls 8-19
    result 8-18
    scope of local name 8-18
    sequence 8-18
    shielded name 8-18
    structured program 8-18
    stop 6-6, 8-14
    suspended function 8-18, 8-19
    terminate 8-20
    trace 6-6, 8-14
Expand \ ⍀ 5-38
Expand components ⍁ 7-8
Explicit
    output □← 4-11
    result of defined function 8-1
Exponent 4-4
Exponential notation *E* 4-4
    in output 5-58
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

F

Factorial ! 5-16
Failure message D-7
False (0) Boolean value 4-4
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
    detach from file ⍈ 7-9
    drop components ⍇ 7-7
    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
    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
Floating point number 4-4
Floor ⌊ 5-7, 8-14
Forgotten password 1-9
Formally equivalent
  expressions ↔ 5-1

Format ⍕ 5-55
Form character ⎕F 6-9
Fractional numbers 4-4
Free file ⎕FX 7-9
Full edit action ∈ 8-10
Function
    definition and editing mode 8-1
    establishment ⎕FX 6-4
Functions
    defined (see Defined functions) 8-1
    file (see File functions) 7-1
    list ⎕NL 2 6-5
        command)FNS 3-9
    no precedence 4-7
    primitive (see Primitive functions) 5-1
    system (see System functions) 6-1
Future value 5-47
Fuzz
    command)FUZZ 3-5
    system function ⎕RL 6-2

G

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
Grave accent, ASCII ` 3-15, 6-10
Greater or equal ≥ 5-12
Greater than > 5-12
Grid 5-31
Group 3-10
    content command)GRP 3-11
    dispersal command)DETACH 3-10
    display ⎕NL 4 6-5
    formation command)ATTACH 3-10
    name of 4-6
    names command)GRPS 3-9

H

Halted functions (see Interrupt, Suspend)
Harmonic sequence 5-31
Hash, ASCII # 3-15, 6-10
Header, function 8-1
Heterogeneous output 4-11
Hexadecimal/decimal conversion 5-47, 5-49
Histogram construction 5-21
Hold file ⎕FH 7-9
Homonym 8-18
Hyperbolic functions ○ 5-14

I

I-bar functions ⌶ 6-22
I-beam (see I-bar)
Identifier 4-6
Identity ⍳ 5-8
    Identity elements for scalar dyadic functions 5-28
    matrix 5-50
Idle character ⎕I 6-9
Illegal character display ⍰ 2-10, 6-11
Implicit
    continuation after unexpected disconnect 2-12
    format ⍕ 5-58
    output 4-11
Inactive
    file 7-2
    workspace 1-8
Index 5-2
    generator ⍳ 5-31
    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
    effect on functions 6-3
    system variable ⎕IO 6-2
Indexing [ ] 5-2
Indices 5-2
Infinity (see largest number) 4-4
Inject line action ⍳ 8-10
Inner product operator ⍤.⍤ 5-26
Input
    communicators ⎕ ⎕ 4-11
    output time ⎕AI 6-12
    transaction 2-8
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
Interrupt
abnormal D-6
by error 8-20, D-2
by two ATTNs 8-20, D-2
character input ⍝ 8-20, D-2
evaluated input ⍝ 8-20, D-2
strong (also see Terminate)
weak (see Suspend)
Intersection ∩ 5-42
Inverse
hyperbolic functions ○ 5-14
matrix ⌹ 5-50
trigonometric functions ○ 5-14
Iota ⍳ 5-31, 8-10
Iterative branching → 8-4

J

Jiggle 3-13, 3-14, D-1
Jot ∘ 5-20
base ⍣ 5-54
cap ⍠ 4-10
quad ⌸ 7-11
top ⍤ 5-55
Join (see Catenate) 5-32
Justify ragged array 5-25

K

Key (see Password)
Keyboard 2-2
Keywords
command)*KEYWORDS* 2-7, 3-15
error 2-12
table 2-4

L

Label
display ⎕*NL* 1 6-5
name 4-6
use of 8-2, 8-3
Laminate , 5-32
Lamp ⍠ (see Comment) 4-10
Largest number 4-4
Last entered expression 4-9
Last file component
in ⍇ 7-5
out ⍈ 7-5
Latent expression (see Load expression) ⎕*LX* 6-1
Least squares estimation 5-53
Leaving function definition mode 8-8
Left
arrow ← 5-4
brace ⊂ 2-4, 3-15, 6-10
bracket [4-8, 5-2, 8-5
identity 5-28
justify 5-25
parenthesis (4-2, 8-12
tack ⊣ 2-4, 6-10
Length
of names 4-6
of vector (see Shape) 5-32
Less or equal ≤ 5-12
Less than < 5-12
Letters of alphabet ⎕*A* 6-9
in atomic vector ⎕*AV* 6-11
in names 4-6
Library
area ⎕*LA* 6-12
of files command)*FILES* 3-6
of inactive workspaces 1-8
public, other usercode 3-6
own 3-6
Line
correction 2-9, 8-10
count ⎕*LC* 6-12
drops, communication 1-8
editing ⍺ ⍷ ⍵ 8-10
group specifier 8-12
in function definition 8-2
insertion ↑ ↓ 8-8
monitoring 6-6, 8-14
number 8-5
number specifier 8-5
renumbering 8-5
replacing [ ] 8-8
suspend 6-6, 8-14
timing 6-6, 8-14
trace 6-6, 8-14
width (see Print width)
zero of defined function 8-1
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
data for format 5-62
delimiter ; 4-8
display defined function □ 8-16
expression 4-8
local names 8-1
Literal character constant 4-5
Load
access 3-8
expression □LX 6-1
workspace command)LOAD 3-7
Local
environment 8-18
functions 8-2
label constant 8-3
left argument 8-1
name 8-2, 8-18
names list 8-1
result 8-1
right argument 8-1
system variables 8-2
variables 8-2
Local/communicate switch 2-3
Location in ⍳ 5-31
Lock
account 3-2
command)LOCK 3-14
file ⍐ 7-3
function ⍒ 8-6
functions □LF 6-4
keyboard 1-3, 2-5
)LOCK OFF 3-14
)LOCK ON 3-14
usercode 4-6
workspace 3-6
Locks and passwords 1-9
Log ⍟ 5-10
Logarithm ⍟ 5-10
Logical
complement ~ 5-13
data (see Boolean)
functions 5-13, 5-42
negation ~ 5-13
station number LSN 2-6
Looping
by backward branch → 8-4
input request prompt D-3
interrupt by ATTN 2-10
no terminal response D-1

M

Magnitude | 5-8
Main diagonal 5-37
Mantissa 4-4
Map file components
non-null ⍌ 7-10
null ⍬ 7-10
Margin
character □M 6-9
key 2-5
Match ≡.≠ 5-27
Matrix
described 1-4, 9-2
display of 4-3
divide ⌹ 5-50
inverse ⌹ 5-50
multiply
inner product +.× 5-26
outer product ∘.× 5-20
scalar × 5-8
nonsingular 5-50
singular 5-50
Maximum ⌈ 5-7
likelihood estimator 5-53
Membership ∈ 5-42
Merge 5-44
Message control system MCS 2-7
command ?MCS 3-14
Messages 3-13
acceptability)RECEPTION 3-13
from operator 3-13
keyboard)LOCK 3-14
to operator)OPR 3-13
to other user)TO 3-13
Minimax ⌊.⌈ 5-27
Minimum ⌊ 5-7
Minterm ∨.∧ 5-27
minus - (see Subtract) 5-8
Mixed
primitive functions 5-29
radix 5-46, 5-48
type list 4-11, 5-55
Modes 1-4
calculator (see Execution)
character data input 4-11
evaluated data input □: 4-11
execution 1-4
function definition and
editing 8-1
prompts 1-4
Modify ⍣← 5-4
Modified insert []⍣← 5-4
Modulo | (see Residue) 5-8

Monadic
function defined 4-7
transpose ⍉ (see Transpose dimensions) 5-36
Monitor
counts ⎕*MC* 6-6
execution 6-6, 8-14
Monitor values ⎕*MV* 6-6
Most recently entered expression retrieval 4-9
Multiline group specifier 8-12
Multiple
assignments ← 5-5
linear regression 5-55
spaces 4-9
Multiply × 5-8

N

N-dimensional data object 4-2
Name
argument 4-6
availibility ⎕*NA* 6-12
characters 2-4
classification ⎕*NC* 6-5
defined function 4-6, 8-1
distinguished (see System functions, System Variables)
eliminate primary)*ERASE* 3-9
expunge ⎕*EX* 6-5
file 4-6
formation rules 4-6
global 8-18
group 4-6
keywords 2-4
label 4-6
list ⎕*NL* 6-5
local 4-6, 8-2, 8-18
password 4-6
restrictions 4-6
result 4-6
shared variable surrogate 6-13
space as separator 4-9
symbols 2-4, 4-6
variable 4-6
uses 4-6
usercode 4-6
workspace 4-6
Name displays
files)*FILES* 3-6
functions)*FNS* 3-9 ⎕*NL* 3 6-5
group)*GRP* 3-11
groups)*GRPS* 3-9 ⎕*NL* 4 6-5
labels ⎕*NL* 1 6-5
library of workspaces)*LIB* 3-6
list ⎕*NL* 6-5
variables)*VARS* 3-6 ⎕*NL* 2 6-5
Nand ⍲ 5-13
National characters 2-4
Natural logarithm 5-10
Negate
arithmetic - 5-8
logical ~ 5-13
Negative sign for number 4-3
Nested function 8-18
News
sign-on message 2-7
system function ⎕*NEWS* 6-12
Niladic
branch → (see Terminate) 8-3
function 4-7, 8-1
No access to workspace 3-8
nocopy, noload, noreplace, nosave
No-element array 4-3
Nonscalar arguments used with scalar functions 5-17
Nonsingular matrix 5-50
Nor ⍱ 5-13
Normal workspace access 3-8
Not ~ 5-13
equal ≠ 5-12
greater ≤ 5-12
less ≥ 5-12
Null
array (see Empty array)
character ⎕*N* 6-9
file component ⍈ 7-4
file map ⍌ 7-10
list component 5-62
obscure name for jot
Number
base conversion 5-46, 5-48
entry 2-5
of users ⎕*UL* 6-12
Numeric
character representation 6-9
data type 4-3
format ⍕ 5-60
odometer order 5-32
sort ⍋ ⍒ 5-44
vector 4-3

O

Of (see Combinations) 5-16
Off commands
abandon)*OFF* 3-2
continue)*COFF* 3-2
keywords 3-16
lock 3-14
Offer, shared variable ⎕*SVO* 4-6
Omega ω 8-10

On 3-2
keywords 3-16
lock 3-14
sign-on command)*ON* 3-2
One
element array 4-4
origin indexing 6-3
Open
bracket [(see Brackets)
file 7-2
function action ∇ 8-6
locked function action ⍢ 8-6
paren ((see Parentheses)
Operators, primitive
inner product ⊛.⊛ 5-26
outer product ∘.⊛ 5-20
reduction ⊛/ 5-22
scan ⊛\ 5-24
Operator message command)*OPR* 3-13
Or ∨ 5-13
Order
of array elements 5-32
of elaboration 4-7
Origin (see Index origin)
Orthogonal 4-2
OUT superimposed ⌿ 4-12, 8-7, 8-20
Outer product operator ∘.⊛ 5-20
Output for display
communicators ⎕← ⍞← 4-11
display form 4-3
formatting ⍕ 5-55
implicit 4-11
mixed type 4-11
Over (see Reduction)
Overbar, for negative numbers ¯ 4-3
Overstruck characters
invalid ⍁ 2-10, 6-11
valid 2-4
Overtake ↑ 5-40
Own workspace access 3-8

P

Padding (see Fill) 5-38, 5-40
Page width (see Print width)
Paragraph, ASCII ⍝ 3-15, 6-10
Parallel elaboration 1-7, 4-7
Parentheses ( )
in an expression 4-70
order of elaboration 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 usercode 3-2
on workspace 3-7
security use 1-9
Pendant function 8-18, 8-20
in state indicator 3-12, D-3
Percent, ASCII ⍀ 3-15, 6-10
Period . (see Dot)
Permutation
grade ⍋ ⍒ 5-44
random ? 5-45
Permute dimensions ⍉ 5-36
Pi times ○ 5-14
Plane across dimensions 4-2
Plotting 5-20
Plus + 5-8
Polynomial 5-47
Power ⋆ 5-10
Precedence, right to left 4-7
Precision of numbers 4-4
comparison tolerance 3-5, 6-2
displayed 3-5, 6-2
internal representation 4-4
Prefix edit action ⍺ 8-10
Present value 5-47
Prime (see Quote)
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

Primitive functions (cont)
greater or equal ≥ 5-12
greater than > 5-12
identity + 5-8
index of ⍳ 5-31
integers to ⍳ 5-31
intersection ∩ 5-42
laminate , 5-32
less or equal ≤ 5-12
less than < 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 ~ 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
reciprocal ÷ 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
subtract - 5-8
superset ⊃ 5-42
take ↑ 5-40
transpose dimensions ⍉ 5-36
union ∪ 5-42
Primitive operators (see Operators, primitive)
Principal diagonal of matrix 5-37
Print pause D-1
Print precision
default command)*DIGITS* 3-5
variable ⎕*PP* 6-2
Print tabs
default command)*TABS* 3-3
function ⎕*PT* 6-12
Print width 3-3
default command)*WIDTH* 3-3
function line folding 8-2
variable ⎕*PW* 6-12
Privacy (see Security, Lock)
Private workspace (see Access) 3-8
Procedures for terminal 2-6
Processor phase 1-3
Product set of indices 5-2
Program (see Defined function)
Progressive expression development 4-9
Prompts 1-4
Properties of data objects 4-5
Protect copy command)*PCOPY* 3-7
Protecting functions
constraint)*ACCESS* 3-8
nocopy, noload, noreplace, nosave
lock
one function ⍎ 8-6
many ⎕*LF* 6-4
own 3-8
seal 8-6, D-3
Pseudo-random (see Random Number)
Public workspace
access 3-8
library command)*LIB* 3-6

Q

Quad
display lines action ⎕ 8-16
evaluated input prompt ⎕: 4-11
explicit output ⎕← 4-11, D-2
overstrikes 2-4
Qualification, line group 8-12
Query ? 5-45, 8-16
file attribute ⎕ 7-11
monitors ⎕*QM* 6-6
stop ⎕*QS* 6-6
trace ⎕*QT* 6-6
Question mark ? 5-45, 8-16
Quitting
APL (see Sign-off)
input
character (see Interrupt ⍬)
evaluated (see Interrupt ⍬; Terminate →)

Quotas
computer use 1-7
files, number 1-7, +/⎕FA 6-12
files, space, 1-7, ⎕SIZE 7-11
on user account 1-7
shared variables 1-7 +/⎕SA 6-12
workspaces 1-7, +/⎕LA 6-12
Quote-quad ⍞ 4-11
Quotes
around character data ' 4-3
in character data '' 4-3

R

Radians ○ 5-14
Radices 5-8
Random number
deal ? 5-45
link default command)SEED 3-5
link variable ⎕RL 6-2
roll ? 5-45
Range of a function result 1-5
Rank
determined by shape function ⍴⍴ 5-30
of a data object 4-2
of function result (see individual functions)
of indexing result 5-2
Rank-n arrays
described 4-2
display of 4-3
Ravel , 5-32
Read component ⎕READ 7-4
Recall of prior entry 2-11
Reception command)RECEPTION 3-13
Reciprocal ÷ 5-8
Recovery 1-8, 2-14
Redundant
blanks 4-9
parentheses 4-9
Recursive function 8-19
Reduction operator ⍟/ 5-22
along empty dimension 5-28
Relational functions 5-12
Release file (see Detach) ⎕UNTIE 7-9
Remainder (see Residue) |
Rename
file ⎕NAME 7-3
function 8-10
workspace command)SAVE 3-7
Reordering a vector 7-3
Repeat key 2-5
Replace
function ← 5-4
line action 8-8
multiple 5-5
workspace access 3-8
Replicator 5-63, 5-65
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
Restructure (see Reshape) 5-30
Result
explicit 8-1, 8-18
value of expression 4-7
Return
character ⎕R 6-9
completing entry 2-8
key RETN 2-3
Reverse ⌽ ⊖ 5-34
file components ⌽ 7-6
Rho ⍴ 5-30
Right
arrow → 8-3
brace } 3-15, 6-10
bracket] 4-8, 5-2, 8-5
identity 5-28
justify 5-25, 5-65
parenthesis) 4-7, 8-12
tack ⊣ 6-10
Right-to-left elaboration 4-7, D-4
Roll ? 5-45
Root (see Power) * 5-10
Rotate ⌽ ⊖ 5-34
file components ⌽ 7-6
Rounding 5-9, 5-58
Row major order 5-32

S

Save
access 3-8
workspace command)SAVE 3-7
Scalar
contrasted to single number
number 4-3
one character in quotes 4-3

Scalar primitive functions
definition 5-6
extension to arrays 5-17
Scan operator ⊕\ 5-24
Scientific notation 4-4
Scope of local name 8-18
Seal function action ⍎ 8-6, D-3
Security 1-9
Seed command)*SEED* 3-5
Selection [ ] 5-2
Self protections 1-8
Self-relative branch →⎕*LC* 8-4
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
Session controls 3-2
Set character input prompt ⎕← 4-11
exclusion ~ 5-42
functions 5-42
monitors ⎕*SM* 6-6, ∩ 8-14
shared variable control 6-18
stop ⎕*SS* 6-6, ⌈ 8-14
trace ⎕*ST* 6-6, ⊤ 8-14
Shape ⍴ 5-30
of data object 4-2, 4-4
of result (see individual functions)
Shared file 7-2, 7-9
Shared variables
access control 6-13
availability ⎕*SA* 6-12
control ⎕*SVC* 6-18
coupling ⎕*SVO* 6-16
name 4-6, 6-13
offer ⎕*SVO* 6-16
query ⎕*SVQ* 6-20
recovery 6-15
retract ⎕*SVR* 6-20
surrogate 6-13
Shares availability ⎕*SA* 6-12
Sharing 1-9
Shielding by local name 8-2
Shift key 2-3
Shoe characters ⊂ ⊃ (see Subset, Superset)
Shriek ! 5-16
Side-effects
none for primitive function 5-1
on global variables 8-1
Sign-off 2-13
continue)*COFF* 3-2
discard)*OFF* 3-2
Sign-on 2-6
connect command)*ON* 3-2
MCS specifier ?MCS 2-7, 3-14
message from operator 2-7, 6-12
Significant digits (see Print Precision)
Signum × 5-8
Simultaneous linear equations 5-52
Sine 1○ 5-14
Single 4-4, D4
Singular matrix 5-50
Sinh 5○ 5-14
Size
data object 4-2 ⍴, 5-33
file
availability ⎕*FA* 6-12
query 7-11
system interrogate 7-11
library ⎕*LA* 6-12
names table ⎕*NA* 6-12
shared variables ⎕*SA* 6-12
workspace ⎕*WA* 6-12
Slash / 2-9, 5-38, 8-10
bar ⌿ 5-38
quad ⍁ 7-8
Slope \ 5-38
bar ⍀ 5-38
circle ⍉ 5-36
quad ⍂ 7-8
Solidus / (see slash)
Sorting 5-44
Space
bar 2-3
blank
in formatted data 5-55
redundant 4-9
separator 4-9
with constant vector 4-9
file ⎕ 7-11
limit B-1, D-4
workspace 6-12, B-1
Special
characters 2-4, 3-15
keys 2-2, 2-3
Specify ← 5-5
Square root
general (see Power) * 5-11
normalized ○ 5-14
Squish quad ⌷ 2-10, 6-11
Standard functions (see Primitive functions)
Star * 5-10

State indicator)*SI* 3-12
clear
command)*RESET* 3-12
one level → 4-12, D-2
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 in workspace □*WA* 6-12, B-1
Stored
clear workspace controls 3-4
files command)*FILES* 3-6
workspaces command)*LIB* 3-6
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 (see Index list) 5-2
Subset ⊂ 5-42
Subtract - 5-8
Sudilos \ (see Slope)
Suffix edit action ⍵ 8-10
Superset ⊃ 5-42
Surrogate
keywords 2-4
shared variable name 6-13
Suspend
by one ATTN 8-19, D-2
recovery after disconnect 2-12
Suspended functions
clearing state indicator command)*RESET* 3-12, 8-20
correcting errors in 8-20
detection of 8-20
meaning of 8-18
suspension notice 8-19
termination → 8-20
Switches on the terminal 2-3
Symbol table
availability □*SA* 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
)*ACCESS* 3-8
)*ATTACH* 3-10
)*BLOT* 3-3
)*CLEAR* 3-4
)*CLEARACCESS* 3-8
)*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
)*KEYWORDS* 2-7, 3-15, 3-16
)*LIB* 3-6
)*LOAD* 3-7
)*LOCK* 3-14
?MCS 2-7, 3-14
)*OFF* 3-2
)*ON* 2-6, 3-2
)*OPR* 3-13
)*ORIGIN* 3-4
)*PCOPY* 3-7
)*RECEPTION* 3-13
)*RESET* 3-12, 8-20
)*SAVE* 3-7
)*SEED* 3-5
)*SI* 3-12
)*SYMS* 3-4
)*TABS* 2-7, 3-3
)*TO* 3-14
)*USE* 2-7, 3-13
)*VARS* 3-9
)*WIDTH* 3-3
)*WSID* (see Workspace Identification) □*WI* 6-12
System functions
account name □*AN* 6-12
accounting information □*AI* 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
dynamic erase (see Expunge)
edit □*ED* 6-8
error □*ER* 6-8
establish functions (see fix)
expunge □*EX* 6-5

System functions (cont)
file availability ⎕FA 6-12
fix ⎕FX 6-4
form character ⎕F 6-9
idle character ⎕I 6-9
latent expression (see load expression)
library availability ⎕LA 6-12
line count ⎕LC 6-12
linefeed character ⎕L 6-9
lock functions ⎕LF 6-4
margin character ⎕M 6-9
monitor counts ⎕MC 6-6
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
unlock functions ⎕UF 6-4
user load ⎕UL 6-12
utility names ⎕UN 6-12
vector representation ⎕VR 6-4
working availability ⎕WA 6-12
workspace identity ⎕WI 6-12
System information ⌶ 6-20
System variables 6-2
comparison tolerance ⎕CT 6-2
in local name list 8-2
index origin ⎕IO 6-2
load expression ⎕LX 6-1
print precision ⎕PP 6-2
random link ⎕RL 6-2

T

Tab character ⎕T 6-9
command)TABS 2-7, 3-3
inquiry ⎕PT 6-12
interval 2-7
irregular D-3
key, SET/CLR 2-5
Table lookup 5-27, 5-32
Tables 5-21
Tack characters
down ⊥ (see Base)
left ⊣ 2-4, 6-10
right ⊢ 2-4, 6-10
up ⊤ (see Top)
Take ↑ 5-40
file components ⍐ 7-7
Tangent 3○ 5-14
Tanh 6○ 5-14
Telephone 2-6
Template of defined function 8-1
Terminal
keyboard 2-2
logical station number 2-6
station name 2-6
tab interval 2-7
width of display 2-8
Terminating function execution
all suspensions)RESET 3-12
from
character input ⍞ then → 4-12, D-2
evaluated input → 4-12, D-2
interrupted → 4-12, D-2
suspended → 4-12, D-2
Test file status ⍓ 7-11
Text (see string) 4-4, 8-11
Tilde ~ 5-13, 5-42, 8-17
Time
connect ⎕AI 6-12
input/output ⎕AI 6-12
monitor value ⎕MV 6-6
processor ⎕AI 6-12
stamp (date and time) ⎕TS 6-12
Times × 5-8
To message command)TO 3-13
Top ⊤ (see Represent) 5-48
jot ⍤ (see Format) 5-55
Trace
function execution 6-6, 8-14
of matrix 5-37
Transaction
definition 1-3
editing 2-8
entries 2-8

Transformations, function 6-4
Transpose dimensions ⍉ 5-36
Triangular numbers 5-25
Trigonometic functions ○ 5-14
True (1) Boolean value 4-4
Truth table 5-12
Type
    conversion
        character to numeric 5-54
        numeric to Boolean 5-12
        numeric to character 5-55
    of data object 1-4
    of function result (see
      individual function)
Typing errors 2-8
Twitch (see Jiggle)

U

Unconditional branch → 8-4
Underbar _ 4-6
    delta Δ 4-6
Unequal ≠ (see Not equal) 5-12
Unexpected prompt D-2
Union ∪ 5-42
Unlock
    functions ⎕*UF* 6-12
    locked functions ⍎ 8-6
    not sealed functions 8-6
Unprotected copy (see Copy)
Unquote (see Evaluate) 5-54
Unused space ⎕*WA* 6-12
Unwanted
    output 8-19, D-2
    processing 8-19, D-2
Up
    arrow ↑ 5-40, 8-8
    caret, error ∧ 5-13, 9-1
    stile ⌈ 5-7
Use command)*USE* 2-7, 3-14
User
    account 1-7
        name ⎕*AN* 6-12
    code 1-7, 4-6
    defined functions (see Defined
      functions) 8-1
    load ⎕*UL* 6-12
    phase 1-3
Utility
    communication with (see Shared
      variables)
    names ⎕*UN* 6-12

V

Valence, argument 4-7, 8-1
Value of a data object 4-2
Variable
    global 8-18
    list ⎕*NL* 3 6-5
        command)*VARS* 3-9
    local 8-1, 8-18
    name 4-6
        unexpected value D-4
    shared 6-13
Vector
    described 4-2
    display of 4-3
    empty 4-4
    represent ⎕*VR* 6-4
Visual fidelity 2-5, 4-9

W

Wait for response D-1
Weightings 5-46
Width
    default command)*WIDTH* 3-3
    format field 5-58, 5-60, 5-65
    print ⎕*PW* 6-12
Working space availability ⎕*WA* 6-12
Workspace
    active 1-8
    attributes 3-4
    clear command)*CLEAR* 3-4
        controls 3-4, 3-8
    identification ⎕*WI* 6-12
    inactive (see library)
    library 3-6
    locking
        functions ⎕*LF* 6-4
        name by password 3-7
    name restrictions 4-6
    names availability ⎕*NA* 6-12
    recovery 2-12
    space availability ⎕*WA* 6-12
    space limitation B-1, D-4
    unlocking functions ⎕*UF* 6-4
Write component ⊞ 7-4
WSID (see Workspace identification)

Z

Zero
    division by 5-8
    origin indexing 6-3
        primitive functions
          affected 6-3
    size data object 4-4