CAI CENTER

SYSTEMS MEMO

A PROGRAMMING LANGUAGE \ 1500

(APL\1500)

Thomas D. McMurchie, Scott E. Krueger
and Henry T. Lippert

Systems Memo No. 8
November 30, 1970

Project NR 154-280
Sponsored by
Personnel & Training Research Programs
Psychological Sciences Division
Office of Naval Research
Washington, D.C.
Contract No. N00014-68-A-0494

FLORIDA STATE UNIVERSITY

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Florida State University
Computer-Assisted Instruction Center
Tallahassee, Florida 32306

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

A Programming Language \ 1500
(APL\1500)

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Systems Memo 8, November 30, 1970

5. AUTHOR(S) (First name, middle initial, last name)

Thomas D. McMurchie, Scott E. Krueger, and Henry T. Lippert

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
November 30, 1970	86	13

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N00014-68-A-0494	
b. PROJECT NO. NR 154-280	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
c.	
d.	

10. DISTRIBUTION STATEMENT
This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Personnel & Training Research & Program Office of Naval Research Washington, D.C.

13. ABSTRACT

~~This~~ document describes the implementation of APL (A Programming Language) for the IBM 1500 Instructional System. It is a revision of the original User's Guide supplied with the first release of the APL System for the 1500.

~~This~~ revised document incorporates a number of extensions to the implementation of APL, particularly the inclusion of the light pen and file handling capabilities. The file items are variable-length, index sequential, randomly accessable and can be any arbitrary APL array (characters or numbers as a scalar, vector or matrix). The full APL is implemented on the 1500 system with only minor restrictions, primarily relating to the smaller size of the workspace as compared to the IBM

FORM 1473 (PAGE 1)
DD 1 NOV 65
S/N 0101-807-6811

Security Classification
A-31408

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT

FORM 1473 (BACK)
DD 1 NOV 65
S/N 0101-807-6821

Security Classification
A-31409

ABSTRACT - continued

360 implementation. Extensions, in addition to the files, include operation of the special terminal hardware of the 1500, primarily the CRT and the film image projector.

A PROGRAMMING LANGUAGE \ 1500

(*APL*\1500)

Thomas D. McMurchie, Scott E. Krueger
and Henry T. Lippert

Systems Memo No. 8
November 30, 1970

Project NR 154-280
Sponsored by
Personnel & Training Research Programs
Psychological Sciences Division
Office of Naval Research
Washington, D.C.
Contract No. N00014-68-A-0494

A PROGRAMMING LANGUAGE \ 1500

(*APL*\1500)

ABSTRACT

This document describes the implementation of APL (A Programming Language) for the IBM 1500 Instructional System. It is a revision of the original User's Guide supplied with the first release of the APL System for the 1500.

This revised document incorporates a number of extensions to the implementation of APL, particularly the inclusion of the light pen and file handling capabilities. The file items are variable-length, index sequential, randomly accessable, and can be any arbitrary APL array (characters or numbers as a scalar, vector or matrix). The full APL is implemented on the 1500 system with only minor restrictions, primarily relating to the smaller size of the workspace as compared to the IBM 360 implementation. Extensions, in addition to the files, include operation of the special terminal hardware of the 1500, primarily the CRT and the film image projector.

ACKNOWLEDGEMENTS

The *APL* language was first defined by K. E. Iverson in A Programming Language (Wiley, 1962) and has since been developed in collaboration with A. D. Falkoff. The *APL*\1500 Terminal System is patterned after the S/360 implementation of *APL* which was written by L. M. Breed, R. D. Moore, and R. H. Lathwell. Moreover, the MAT System for the IBM/1500 (developed by Service Bureau Corporation) provided the basic floating point arithmetic and trigonometric routines, and certain fundamental execution logic. The development of the system was further aided by the disk file-service routines and file-search commands written by H. A. Driscoll.

In the preparation of the original *APL*\1500 User's Guide, the authors are indebted to K. E. Iverson and A. D. Falkoff for permission to adapt text from their *APL*\360 User's Manual; and to S. S. Pakin and D. L. Krueger for critical reading and assistance in production of the final copy.

The *APL*\1500 system was originally released to users of the IBM 1500 Instructional System from the Computer-Related Instructional Systems Center of Science Research Associates (CRIS) in Chicago. Version two of the system was a more extensively debugged and enhanced (e.g., inclusion of light-pen capability) release of version one. Version three of the *APL*\1500 System has several extensive changes, primarily the inclusion of a dynamically controlled file handling capability. Both versions two and three were the work of Thomas D. McMurchie of the Florida State University's Computer-Assisted Instruction Center, with the encouragement of Duncan N. Hansen, who is the Director of the FSU CAI Center.

The body of the entire document was edited and composed on *APL*\1500 and *APL*\360 using T. D. McMurchie's implementation of a text processing package developed by M. M. Zyrl and A. P. Mullery (IBM Research).

The authors wish to acknowledge the contributions made by Henry T. Lippert, David B. Thomas, Edward V. Harris, and Paul G. Merrill of the FSU CAI Center in the production of this revised version of the original User's Guide for version three of the *APL*\1500 System.

INTRODUCTION

System features

APL\1500 is based upon APL, the language first defined by K. E. Iverson in A Programming Language (John Wiley, 1962). It is further based on the IBM/360 implementation of APL, *APL*\360. *APL*\1500 is an interpretative time-sharing system that builds upon the array operations and structural integrity of APL to provide a system with the following salient characteristics:

Simple, uniform rules of syntax

Use of common symbols for ordinary arithmetic operations

Free-form decimal input

A large set of primitive operators

Use of defined functions (programs) with the same facility and syntactic variety as primitive operators

An immediate-execution mode completely free of unnecessary keywords

A comprehensive, integrated set of system commands for managing workspaces and other essential functions

Ability to dynamically manipulate large data bases

Three levels of security; account numbers, workspaces, and programs can be individually locked against use or display

A built-in plot routine

Ability to have CRT and typewriter devices as a single instructional station

Visual fidelity between hard copy and transmitted entries, which ensures reproducibility of results

Succinct diagnostic reports

APL\1500 is a conversational multi-terminal system that was developed at the CRIS Center of SRA. It was written as a stand-alone program to replace the MAT package provided with the IBM/1500 system. Its purpose is to combine the simplicity, power, and conciseness of the *APL*\360 system with the special hardware features of the 1500 -- the CRT display unit and the film projector unit. The *APL*\1500 system supports up to 32 terminals. Minimum configuration:

32K 1130 (or 1800) CPU;

1502 Display Control Unit;

2310 Disk Drive;

1132 (or 1443) Printer;

1442 Card Read-Punch;

1518 Typewriters and/or 1510 CRTs with keyboard; and optionally 1512 Film Projectors, and/or 1510 light pens.

APL\1500 is now running on IBM/1130 and IBM/1800 based 1500 Systems. Furthermore, the same *APL*\1500 System will run on either the IBM/1800 or IBM/1130 with no modifications. CPU usage will be approximately 30 percent higher with an IBM/1130 based system than on an IBM/1800 based system with equal core storage cycle times due to hardware considerations.

Average reaction time of an IBM/1800 based system (i.e., time to respond to trivial requests from a terminal) with 12 stations in immediate execution mode is generally less than one second. With light function execution usage, most such responses are essentially instantaneous; when heavily loaded, there are occasional delays of as much as six seconds.

The time for serving non-trivial requests naturally varies according to the extent to which the CPU must be shared during the computation. Because the primitive operations of APL are defined on arrays, relatively little interpretive overhead is needed for many large computations, and the actual CPU time used for a typical immediate execution mode computation may run from 10 to 30 times that for efficiently compiled code; but the overall efficienty is likely to be comparable, if not superior, to batch processing in many applications if the usual compiling and loading times for batch work are taken into account. If debugging time is included, the advantage of interpretive APL becomes even greater.

TABLE OF CONTENTS

Table of Contents - continued

Table of Contents - Continued

Table of Contents - Continued

LIST OF ILLUSTRATIONS

PART I

GAINING ACCESS

This part of the manual describes the characteristics of the *APL*\1500 terminals, the establishment of a connection between the terminal and the central computer, and the procedures for starting a work session.

TERMINAL DEVICES

Each terminal may be comprised of an IBM 1518 Typewriter or an IBM 1510 CRT with Keyboard, or both. In addition, the terminal configuration may include an IBM 1512 Film Projector and an IBM 1510 Light Pen.

IBM 1518 TYPEWRITER:

Each typewriter should be equipped with an IBM Standard Selectric APL printing element; part number 1167987. The standard line width is 120 characters. When the system is waiting for input, the light on the typewriter keyboard will be on and the keyboard will be unlocked. The typewriter cursor is defined as the position, marked by the location of the typeball, where the next character typed will be printed.

IBM 1510 CRT WITH KEYBOARD:

The CRT screen is composed of 16 lines of 40 characters each. The sixteenth line is reserved for messages transmitted between terminals. During display, if the output is incomplete at the bottom of the CRT screen, the output is halted and the system will wait for any key press to continue the display starting at the top of the screen. The CRT cursor is defined as the position, marked by a dotted box, where the next character typed will be displayed. When the system is waiting for input, the cursor position is displayed.

IBM 1512 FILM PROJECTOR:

The film projector is an output device only, and allows for the display of any one of 1022 frames of film. See IBM 1500 Film Preparation Guide for a complete description.

IBM 1510 LIGHT PEN:

The light pen is an input device only, and allows for the selection of the row and column coordinates of any character on the CRT screen.

THE *APL*\1500 KEYBOARD AND CHARACTER SET

The numerals, alphabetic characters, and punctuation marks appear in their usual places on the keyboard. The letters are displayed as upper-case italics, but are produced only when the keyboard is in the lower case position (i.e. not shifted). The special characters are generally produced with the keyboard shifted. The *APL*\1500 keyboard is shown in figure 1.

Figure 1. *APL*\1500 KEYBOARD

ENTRIES FROM THE KEYBOARD

Normal communication between a terminal and the central computer is carried on by means of entries from the keyboard, which locks when each entry is made and unlocks when the computer completes its work. The general procedure is to type an instruction or command, and strike the RETURN key to indicate the end of the message. In the remainder of this manual the need for the RETURN key will not be explicitly mentioned, since it is required for every entry.

Mistakes:

Errors in typing can be corrected before the RETURN key completes an entry:

1. Backspace to the point of error and then depress the INDEX key. This will have the effect of deleting everything to the right of, and including, the position of the cursor. The corrected text can be continued from that point, on the new line.

2. The entire line may be deleted by simultaneously depressing the ALT CODE key and the + (plus) key. A new line can then be entered.

3. If the terminal input device is a CRT, characters may be erased by holding down the ALT CODE key and striking the BACK SPACE key until the error has been erased. Entry can then be continued from that point.

REMEMBER: Each entry is interpreted exactly as it appears, regardless of the time sequence in which the characters were typed.

Continuation

If entries are longer than a single line, they may be continued on the next line by simultaneously depressing the ALT CODE key and the RETURN key. The cursor will be positioned on the next line at the left margin. Errors on a line ended with a continuation can not be corrected.

Underscored alphabet

The APL alphabet consists of the letters *A* thru *Z*. The underscored letters can be formed in either of two ways:

1. Overstrike the letter with an underscore (up-shifted *F*).

2. Hold down the ALT CODE key and depress the desired letter. This operation will automatically result in the underscored letter.

Attention

Attention is obtained by holding down the ALT CODE key and striking the INDEX key. This operation has two effects:

1. If the terminal is signed off, the attention signal will initially establish communication between the terminal and the central computer. The keyboard will unlock and the system will be ready to accept input from the user. If the sign-on is not completed in 60 seconds the input will be ignored and the sign-on must be re-accomplished.

2. If the terminal is already signed on, the attention signal will stop execution or output in progress. If the system is waiting for input, the attention signal will have no effect. See Part 2 for a further explanation of attention.

STARTING A WORK SESSION

Each user of the system is assigned an account number. This number is used to effect the sign-on that initiates a work session and is used to identify the work that the user may store in the system between work sessions.

The following is a description of the sign-on command which is entered after attention has been signaled.

START A WORK SESSION:)nnnnnn

Enter a right parenthesis followed by an account number and, if required, a key (i.e. a colon and a password). The use of passwords as locks and keys is described in Part 2.

Effect:

A workspace will be activated for the terminal and the accumulation of time charges will begin. A workspace can be thought of as both a notebook and a scratch pad. The details are explained in Part 2.

Response:

1. The port number, user name associated with the account number, date, and time of day will be displayed.

2. The system identification *'A P L \ 1 5 0 0'* will be displayed.

Trouble reports:

NUMBER NOT FOUND
Either no such number has been assigned or the number has a lock associated with it and the wrong key was used. The APL operator should be consulted if help is required.

INCORRECT COMMAND
The form of the transmitted command was faulty.

TIME AND DATE NOT SET
The first user to sign on has not yet set the time and date.

If you are the first user to sign on, then you must set the time and date. See Part 2 for a description of the commands used to set the time and date.

Once the sign-on is accomplished, a work session is started, and the full APL system becomes available.

PART 2

SYSTEM COMMANDS

APL operations deal with transformations of abstract objects, such as numbers and symbols, whose practical significance, as is usual in mathematics, depends on the (arbitrary) interpretation placed upon them. System commands in the *APL\1500* System, on the other hand, have as their subject the structures which comprise the system, and control functions and information relating to the state of the system, and therefore have an immediate practical significance independent of any interpretation by the user.

This section describes the structure of the *APL\1500* system, introduces the various notations essential to the understanding of system commands, and describes the complete set of system commands in detail.

WORKSPACES AND LIBRARIES

Workspaces

The common organizational unit in the *APL\1500* system is the workspace. When in use, a workspace is said to be active and occupies a block of working storage in the central computer. The size of the block, which is preset at a fixed value, determines the combined working area and storage capacity of each workspace in the system. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, for storing items of information and for containing transient information generated in the course of a computation.

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

Libraries

Inactive workspaces are stored in libraries. They occupy space in secondary storage (disks) and cannot be manipulated directly. When required, copies of stored workspaces can be made active, or selected information may be copied from them into an active workspace.

Libraries are associated with individual users of the system, and are identified by the user's account number. Access to them is restricted in that one user may not store workspaces in another person's library. However, one user may activate a copy of another user's workspace if he knows the library number and the key (if one is required).

NAMES

Names of functions and variables may be any single alphabetic character (*A* to *Z*, and *Δ* to *Ω*).

The environment in which APL operations take place is bounded by the active workspace. Hence, the same name may be used to designate different objects (i.e., functions or variables) in different workspaces, without interference. However, the objects within a workspace must have distinct names, except as explained below.

Local and global significance

In the execution of defined functions it is often necessary to work with intermediate results which have no significance either before or after the function is used. To avoid cluttering the workspace with a multitude of variables introduced for such transient purposes, and to allow greater freedom in the choice of names, the function definition process (see Part 3) provides a facility for designating certain variables as local to the function being defined. Variables not so designated, and all functions, are said to be global.

A local variable may have the same name as a global object, and any number of variables local to different functions may have the same name.

During the execution of a defined function, a local variable will supersede a function or global variable of the same name, temporarily excluding it from use. If the execution of a function is interrupted (leaving it either suspended or pendant, see Part 3), the local variables retain their dominant position during the execution of subsequent APL operations, until such time as the halted function is completed. System commands, however, continue to reference the global homonyms of local variables under these circumstances.

LOCKS AND KEYS

Stored workspaces and the information they hold can be protected against unauthorized use by associating a lock, comprising a colon and a password of the user's choice, with the workspace, when the workspace is stored. In order to activate a locked workspace or copy any information it contains, a colon and the password must again be used, as a key.

Account numbers can be similarly protected by locks and keys, thus maintaining the security of a user's private library and avoiding unauthorized charges against his account.

Passwords for locks and keys may be formed of any sequence of characters up to six characters long, without blanks or colons. Characters beyond the sixth are ignored. In use as either a lock or key, a password is set off by a preceding colon.

ATTENTION

Printed output at a terminal can be cut off, or the execution of an APL operation can be interrupted, and control returned to the user, by means of an attention signal. Attention is obtained by holding down the ALT CODE key and pressing the INDEX key.

Following an attention signal the keyboard will unlock, and the cursor will return to the normal position for input (two spaces from the left margin). In some cases a line will be printed before the keyboard unlocks, telling where a function in progress was interrupted.

The execution of system commands, once entered, cannot be interrupted. However, the printed responses or trouble reports following a system command can be suppressed by a properly timed attention signal.

USE OF SYSTEM COMMANDS

System commands and APL operations are distinguished functionally by the fact that system commands can be called for only by individual entries from the keyboard, and cannot be executed dynamically as part of a defined function. They are distinguished in form by the requirement that system commands be prefixed by a right parenthesis, which is a syntactically invalid construction in APL.

It may be desirable to perform dynamically some system control, and to use some items of system information during the execution of a program. For these purposes *APL\1500* provides appropriate system-dependent functions, which can be used like other APL operations. These functions are described in Part 3.

System commands are conveniently grouped into six classes with regard to their effect upon the state of the system. The summary table of commands and the descriptive text associated with them are based upon this classification:

1. Terminal control commands affect the relation of the terminal to the system.

2. Workspace control commands affect the state of the active workspace.

3. Library control commands affect the state of the user's stored library.

4. Inquiry commands provide information without affecting the state of the system.

5. Communication commands effect the transmission of messages among terminals.

6. System start up commands must be executed by the first signed-on user in order to fully activate *APL\1500*. These become privileged (inactive to all but the system operator) after they have been executed.

Any entry starting with a right parenthesis will be interpreted by the system as a system command. When the command is successfully executed, the normal response, if any, will be printed.

If, for any reason, a command cannot be executed, an appropriate trouble report (error report) will be printed. The most common report is *INCORRECT COMMAND*. This means that the command was incomplete, misspelled, modified incorrectly, or otherwise malformed. If the time and date have not yet been set, the system will reject all attempted system commands except the first sign-on; in this case the report *TIME AND DATE NOT SET* will be given.

Where the command name is more than four characters long, only the first four are significant. The others are included for mnemonic reasons, and may be dropped or replaced as desired. For example, *)CLEAR*, *)CLEA*, *)CLEANSE*, etc., are all equivalent. In general, the elements of a command must be separated by one or more spaces. Spaces are not required immediately following the right parenthesis, or on either side of the colon used with passwords, but can be used without harm.

PURPOSE	COMMAND	NORMAL RESPONSE	ERROR REPORTS
SIGN-ON USER AND START SESSION	)wsid [key]	Terminal,Name,Date,Time; *APL*\1500;[*OPR*: text]	1 2
TERMINATE SESSION	)*OFF* [lock]	Terminal,Date,Time; Connect;Latency;CPU	1
ACTIVATE CLEAR WS	)*CLEAR*		1
CLEAR STATE INDICATOR	)*PURGE*		1
ACTIVATE A STORED WS	)*LOAD* [wsid] [key]	*SAVED*,Date,Time	1 2 3 9
COPY A GLOBAL OBJECT	)*COPY* wsid [key] A [B]	*SAVED*,Date,Time	1 2 3 4 5 6 9
COPY ALL GLOBAL OBJECTS	)*COPY* wsid [key]	*SAVED*,Date,Time	1 2 3 4 6 9
COPY A GLOBAL OBJECT, PROTECT ACTIVE WS	)*PCOPY* wsid [key] A [B]	*SAVED*,Date,Time	1 2 3 4 5 6 9
COPY ALL GLOBAL OBJECTS, PROTECT ACTIVE WS	)*PCOPY* wsid [key]	*SAVED*,Date,Time	1 2 3 4 6 9
ERASE GLOBAL OBJECT[S]	)*ERASE* name[s]		1 7
SET INDEX ORIGIN	)*ORIGIN* integer,0-1	*WAS*,former origin	1
SET OUTPUT LINE WIDTH	)*WIDTH* integer,20-120	*WAS*,former width	1
SET MAX FOR SIGNIFICANT DIGITS IN OUTPUT	)*DIGITS* integer,1-6	*WAS*,former significance	1
SET WS IDENTIFICATION	)*WSID*		1
STORE A COPY OF ACTIVE WS	)*SAVE* [lock]	*SAVED*,Date,Time	1 8 9
ERASE A STORED WS	)*DROP*	*DROPPED*,Date,Time	1 9

PURPOSE	COMMAND	NORMAL RESPONSE	ERROR REPORTS
LIST NAMES OF DEFINED FUNCTIONS	*)FNS*	List of function names and header syntax.	1
LIST NAMES OF GLOBAL VARIABLES	*)VARS*	List of variable names and rank.	1
LIST STATE INDICATOR	*)SI*	List of halted functions.	1
LIST ACTIVE USERS	*)PORTS*	List of active ports and users.	1
LIST ENABLED FILES	*)FILES*	List of names of files enabled for this user.	1
SEND TEXT TO A PORT	*)MSGN* port [text]	*SENT*	1 10
SEND TEXT TO OPERATOR	*)OPRN* [text]	*SENT*	1

The System Commands *)TIME* and *)DATE* must be executed by the first user to sign on the system (usually the operator). Once these commands have been executed, they become privileged. Until these commands are executed, all other System Commands will yield the report:
TIME AND DATE NOT SET

SET THE TIME	*)TIME* hours minutes seconds	1
SET THE DATE	*)DATE* month day year	1

NOTES:
1. Items in brackets are optional.
2. key or lock: a password (1-6 characters) set off by a preceding colon. A colon alone following a command removes a lock.
3. wsid: workspace number (1-6 digits).
4. '[B]' of Copy commands will copy object 'A' and change its name to 'B'.

ERROR REPORTS	
1	*INCORRECT COMMAND*
2	*NUMBER NOT FOUND*
3	*WS NOT FOUND*
4	*NOT COPIED:* list of objects
5	*OBJECT NOT FOUND*
6	*WS FULL ERROR*
7	*NOT ERASED:* list of objects
8	*NOT SAVED*
9	*PACK ERROR*
10	*STATION SIGNED OFF*

TERMINAL CONTROL COMMANDS

There is one command for starting a work session and one command for ending it. The starting command has been described in Part 1.

A work session can be stopped remotely, from a privileged user's terminal, in an action known as a bounce. The bounce may be used when a terminal is required for a special purpose, or to clear the system of all users before stopping the *APL\1500* operation completely. The bounce performs as *)OFF*.

If a work session is ended because of a failure of the central computer, the active workspace is not stored.

Elapsed time, latency, and time of day, given as a system response, are always in hours, minutes, and seconds; two digits for each, separated by colons. A date response is given as month, day, and year; two digits for each, separated by slashes. Clock hours are counted continuously from midnight of the indicated day, and if the system runs past midnight it is possible to have time readings above 24 hours. For example, 34:22:00 10/01/68 would be 22 minutes past 10 AM on October 2, 1968.

START A WORK SESSION:
This is the sign-on described in Part 1.

END A WORK SESSION. *)OFF*
Enter *)OFF* optionally followed by a colon and a password. Passwords longer than 6 characters are accepted but only the first 6 are meaningful. Spaces around the colon are neutral.

Effect:

1. The currently active workspace will vanish. There is no effect on any stored workspace.

2. The duration of the work session, the user input latency, and the amount of computer time used will be noted internally for later accounting.

3. The password, if used, will become a new lock on the account number. Once applied, a lock stays in effect until explicitly changed by an *)OFF* command that contains a colon. An existing lock is removed if no password follows the colon.

Response:

1. The port number, date, and time of day will be printed on one line.

2. Accounting information will be printed on three lines giving terminal connect time, user input latency, and central computer time. Input latency is defined as the total time the keyboard was unlocked and waiting for input. The time used in this session and cumulative time since the last accounting are given in the standard format.

WORKSPACE CONTROL COMMANDS

The commands in this class can replace the active workspace with a clear one, or with a copy of a stored workspace; bring together, in the active workspace, information from many stored workspaces; remove unwanted objects from the active workspace; remove all levels of suspension; and set controls governing certain operations. The commands in this class affect only the active workspace.

The usefulness of a terminal system is enhanced by the availability of many different collections of functions and variables, each of which is organized to satisfy the computational needs of some area of work such as standard statistical calculations, exercises for teaching a subject, complex arithmetic, business accounting, simulations, etc. The workspace-centered organization of *APL\1500* lends itself to such packaging, because each collection moves as a unit when the workspace containing it is stored or activated.

The copy commands provide a convenient way to assemble packages from components in different workspaces. Information entered or developed within one workspace can be made available within another by means of the copy and protecting-copy commands, which reproduce, within the active workspace, objects from a stored workspace. These are two sets of parallel commands which differ only in their treatment of an object in the active workspace which has the same name as an object being reproduced: the copy commands will replace the existing object, whereas the protecting-copy commands will not make the replacement.

A copy command of either type can be applied to an entire workspace or to a single object (i.e., a function or variable). When an entire workspace is copied, all the functions and global variables within it are subject to the operation, but its index origin and output control settings, state indicator, and local variables are left behind. Either copy command may copy a single object and change its name in the active workspace.

NOTES: 1. The term wsid (workspace identification) is used here to mean a library number (account number). When the wsid is omitted, the reference is to the user's library.

2. A key is a colon followed by a password.

3. The system response, *INCORRECT COMMAND*, may occur for any system command. This means that the command was incomplete, misspelled, modified incorrectly, or otherwise malformed.

ACTIVATE A CLEAR WORKSPACE: *)CLEAR*

Enter *)CLEAR*

This command is used to make a fresh start, discarding whatever is in the active workspace.

Effect:

A clear workspace will be activated, replacing the presently active workspace. A clear workspace has no variables or defined functions. Its control settings are: index origin, 1; significant digits, 6; line width, 120 on the typewriter and 40 on the CRT; maximum input latency, 32767. Its workspace identification does not match that of any stored workspace.

Response:

None.

CLEAR THE STATE INDICATOR: *)PURGE*

Enter *)PURGE*

Effect:

The state indicator is cleared (see *)SI* command).

Response:

None.

ACTIVATE A COPY OF A STORED WORKSPACE *)LOAD*

Enter *)LOAD* optionally followed by a space and a wsid (with the key, if required). This command may be used to obtain the use of any workspace whose identification (and password) is known. If the wsid is omitted, the user's workspace is assumed.

Effect:

A copy of the designated workspace will be activated, replacing the presently active workspace.

Response:

SAVED, followed by the date and time of day that the source workspace was last stored.

Trouble reports:

NUMBER NOT FOUND
There is no library for the entered number.

WS NOT FOUND
There is no stored workspace with the given identification, the key was omitted when one was required, or the wrong key was used.

PACK ERROR
The disk pack containing the referenced library was not mounted and ready.

COPY A GLOBAL OBJECT FROM A STORED WORKSPACE: *)COPY*

Enter *)COPY* followed by a space, a wsid (with the key, if required), a space, and the name of the object to be copied; then, optionally, a space and the new name the object is to have in the active workspace. A global object may be a function or global variable.

Effect:

1. The global homonym in the active workspace will be erased.

2. A copy of the designated object will appear in the active workspace with global significance.

Response:

SAVED, followed by the date and the time of day the source workspace was last stored.

Trouble reports:

NUMBER NOT FOUND
see *)LOAD*

WS NOT FOUND
see *)LOAD*

PACK ERROR
see *)LOAD*

OBJECT NOT FOUND
The designated workspace does not contain a global object with the given name.

NOT COPIED:
The listed object was not copied because the active workspace was full or the state indicator was not clear.

WS FULL ERROR
The active workspace could not contain all the material requested. If copied at all, a variable or function will be copied completely.

COPY ALL GLOBAL OBJECTS FROM A STORED WORKSPACE: *)COPY*

Enter *)COPY* followed by a space, and a wsid (with the key, if required).

Effect:

1. All global homonyms in the active workspace will be erased.

2. A copy of all functions and global variables in the source workspace will appear in the active workspace with global significance. Local variables, the state indicator, and settings for origin, significant digits, and width will not be copied.

Response:

SAVED, followed by the date and the time of day the source workspace was last stored.

Trouble reports:

NUMBER NOT FOUND
see *)LOAD*

WS NOT FOUND
see *)LOAD*

PACK ERROR
see *)LOAD*

NOT COPIED:
The listed objects were not copied because the state indicator was not clear, or the active workspace function file or data storage area was full.

WS FULL ERROR
The active workspace could not contain all the material requested. If copied at all, a variable or function will be copied completely.

COPY A GLOBAL OBJECT FROM A STORED WORKSPACE, PROTECTING THE ACTIVE WORKSPACE: *)PCOPY*

Enter *)PCOPY* followed by a space, a wsid (with the key, if required), a space, and the name of the object to be copied; then, optionally, a space and the new name the object is to have in the active workspace.

Effect:

A copy of the designated object will appear in the active workspace unless there is an existing global homonym.

Response:

SAVED, followed by the date and the time of day the source workspace was last stored.

Trouble reports:

NUMBER NOT FOUND
see)LOAD

WS NOT FOUND
see)LOAD

PACK ERROR
see)LOAD

OBJECT NOT FOUND
The designated workspace does not contain a global object with the given name.

NOT COPIED:
The listed object was not copied because the active workspace was full, the state indicator was not clear, or there was a global homonym in the active workspace.

WS FULL ERROR
see)COPY

COPY ALL GLOBAL OBJECTS FROM A STORED WORKSPACE, PROJECTING THE ACTIVE WORKSPACE:)*PCOPY*

Enter)*PCOPY* followed by a space and a wsid (with a key, if required).

Effect:

A copy of all global objects in the source workspace which do not have global homonyms in the active workspace will appear in the active workspace.

Response:

SAVED, followed by the date and the time of day the source workspace was last stored.

Trouble reports:

NUMBER NOT FOUND
see)LOAD

WS NOT FOUND
see)LOAD

PACK ERROR
see)LOAD

NOT COPIED:
The listed objects were not copied because the state indicator was not clear, the active workspace function file or data storage area was full, or there were global homonyms in the active workspace.

WS FULL ERROR
see)COPY

ERASE GLOBAL OBJECTS: *)ERASE*

Enter *)ERASE* followed by a space and the names of global objects to be deleted, separated by spaces. This is the only way to remove global variables and the most convenient way to remove functions.

Effect:

Named objects having global significance will be expunged. Names which do not refer to global objects will be ignored.

Response:

None.

Trouble report:

NOT ERASED:

The listed functions were not erased because the state indicator was not clear.

SET INDEX ORIGIN FOR ARRAY OPERATIONS: *)ORIGIN*

Enter *)ORIGIN* followed by a space and a 0 or 1. See also ⎕2 and I12 in Part 3.

Effect:

The first element of arrays in the workspace will be numbered zero or one, as indicated, and subsequent use of index-dependent APL operations will be appropriately affected. Index origin is more fully explained in Part 3.

Response:

WAS, followed by the former origin.

SET MAXIMUM WIDTH FOR AN OUTPUT LINE: *)WIDTH*

Enter *)WIDTH* followed by a space and an integer between 20 and 120 inclusive. If the input device is a CRT, the line width will be set to 40 if the entered value is greater than 40. See also ⎕6 and I16 in Part 3.

Effect:

Subsequent output of all kinds will be limited to a line width no greater than the number of spaces indicated. This command will not affect the length of input lines.

Response:

WAS, followed by the former maximum width.

SET MAXIMUM FOR SIGNIFICANT DIGITS IN OUTPUT: *)DIGITS*

Enter *)DIGITS* followed by a space and an integer between 1 and 6 inclusive. See also ⎕9 and I29 in Part 3.

Effect:

Subsequent output of numbers will show no greater number of significant digits than indicated. This command has no effect on either input or the precision of internal calculations, which is approximately 7 decimal digits.

Response:

WAS, followed by the former maximum.

SET IDENTIFICATION OF ACTIVE WORKSPACE TO THAT OF STORED WORKSPACE: *)WSID*

Enter *)WSID*

Effect:

The identification of the user's active workspace is made to match that of the user's stored workspace, if one exists. The effect of this command is to override the *NOT SAVED* report of the *)SAVE* command (see *)SAVE* command), permitting the user to save his active workspace when the system would normally prevent the save.

Response:

None.

LIBRARY CONTROL COMMANDS

There are two basic operations performed by the commands in this class. The save command causes a copy of an active workspace to be stored in the user's library, and the drop command causes such a stored copy to be destroyed.

The save command and the load command are symmetric, in the sense that a load command destroys an active workspace by replacing it with a copy of a stored workspace, while a save command destroys a stored workspace by replacing it with a copy of an active workspace.

When a workspace is stored, an exact copy of the active workspace is made, including the state indicator and intermediate results from partial execution of halted functions. These functions can be restarted without loss of continuity (see Part 3), which permits considerable flexibility in planning use of the system. For example, lengthy calculations do not have to be completed at one terminal session; student work can be conducted over a series of short work periods; and mathematical experimentation or the exploration of system models can be done over long periods of time, at the investigator's convenience.

A library number uniquely identifies each stored workspace in the system. An active workspace is also identified by a library number, and as copies of stored workspaces are activated, or copies of the active workspace are stored, the identification of the active workspace may change according to the following rules:

1. A workspace activated from a library assumes the identification of its source.

2. When a copy of the active workspace is stored, the active workspace the identification of the subject library.

3. A clear workspace activated by a *)CLEAR* command, a sign-on, or a system failure will not match the identification of any stored workspace.

The identification of active workspaces is used as a safeguard against inadvertent replacement of a stored workspace by an unrelated one. The *)SAVE* command compares the identification of the active workspace with that of the user's library to determine if the save operation will be performed.

Summary

Each stored workspace has implicitly associated with it the account number signed on at the terminal from which the save command was entered, and may not be either replaced or erased, except from a terminal signed on with the same account number. Thus, one user is prevented from affecting the state of another user's private library. The user may, of course, activate a copy of any workspace stored in the system, if he knows the library number (and password, if required).

A user of *APL\1500* is assigned library space for, at most, one workspace in his private library. A user's account number is also the number of his private library.

RE-STORE A COPY OF THE ACTIVE WORKSPACE: *)SAVE*

Enter *)SAVE* optionally followed by a colon and a password.

Effect:

1. A copy of the active workspace will replace the user's stored workspace.

2. The password, if used, will become a new lock on the workspace. Once applied, a lock stays in effect until explicitly changed by a *)SAVE* command that contains a colon. An existing lock is removed if no password follows the colon.

Response:

SAVED, followed by the date and the time of day.

Trouble reports:

PACK ERROR
see *)LOAD*

NOT SAVED
The active workspace can be stored only if the wsid of the active workspace agrees with the wsid of the stored workspace, or the stored workspace has been dropped.

ERASE A STORED WORKSPACE: *)DROP*

Enter *)DROP*

Effect:

The stored workspace will be expunged. Since a key is not used, a locked workspace whose key has been lost can always be removed from the system. This command has no effect on the active workspace.

Response:

DROPPED, followed by the date and the time of day.

[redacted]

[redacted] information about the [redacted], or the *APL\1500* System [redacted]

[redacted]

[redacted] in the active workspace

[redacted]

LIST NAMES AND PORTS OF ALL ACTIVE USERS: *)PORTS*

Enter *)PORTS*

Effect: None.

Response:

The ports and names of all active users will be listed. This information may then be used to determine if a particular user is signed on, or to ascertain his port number in order to direct a message to him (see *)MSGN* command).

LIST NAMES OF ENABLED FILES: *)FILES*

Enter *)FILES*

Effect: None.

Response:

The names of files which have been enabled for this user will be listed. The names of public files will not appear unless they have also been enabled for this user.

COMMUNICATION COMMANDS

There are two commands in this class. One command addresses any connected terminal, and one command addresses only the system recording terminal (operator's terminal).

Messages can be received by a terminal only when its keyboard is locked. Incoming messages from the system recording terminal are prefixed by *OPR:*. The length of a message is restricted to a maximum of 114 characters. However, messages are not subject to width settings of either the sending or receiving terminal. Messages sent to a CRT will appear at the bottom of the screen and are physically limited to a display of 34 characters.

ADDRESS TEXT TO DESIGNATED TERMINAL: *)MSGN*

Enter *)MSGN* followed by a space, a port number, a space, and the desired text.

Effect:

The text will be displayed at the receiving terminal, prefixed by the port number of the sending terminal.

Response:

SENT

Trouble Reports:

STATION SIGNED OFF

The message was lost because the designated terminal was signed off.

ADDRESS TEXT TO THE SYSTEM RECORDING TERMINAL: *)OPRN*

Enter *)OPRN* followed by a space and the desired text.

Effect:

The text will be displayed at the system recording terminal, prefixed by the port number of the sending terminal. If the recording terminal does not exist or is not operational, the message will be lost.

Response:

SENT

SYSTEM START UP COMMANDS

There are two commands in this class. These commands must be executed by the first signed-on user in order to activate the entire *APL\1500* System. Until the time and date commands have been executed, all other system commands, including attempted sign-ons by other users, will yield the report: *TIME AND DATE NOT SET.*

After the time and date commands have been entered, they will become unavailable for normal execution (privileged). These commands are usually entered by the system operator when the system is initially started for the day.

SET THE TIME OF DAY: *)TIME*

Enter *)TIME* followed by a space, the number of hours past midnight, a space, the number of minutes past the hour, a space, and the number of seconds past the minute.

Effect:

The time of day will be set and the user's sign on time will be reset. After execution, this command will become privileged.

Response: None.

SET THE DATE: *)DATE*

Enter *)DATE* followed by a space, the number of the month, a space, the day of the month, a space, and the last two digits of the year.

Effect:

The date will be set. After execution, this command will become privileged.

Response: None.

PART 3

THE LANGUAGE

The *APL\1500* System executes system commands and mathematical statements entered at a terminal. The system commands were treated in Part 2; the mathematical statements will be treated here.

Acceptable statements may employ either primitive functions (e.g., + - × ÷) which are provided by the system, or defined functions, which the user provides by entering definitions at the terminal.

If system commands are not used, the worst that can possibly result from erroneous use of the keyboard is the printing of an error report. It is, therefore, advantageous to experiment freely and to use the system itself for settling any doubts about its behavior. For example, to find what happens in an attempted division by zero, simply enter the expression 4÷0.

The Sample Terminal Session in Appendix A shows actual intercourse with the system and may be used as a model in gaining facility with the terminal. The examples generally follow the text and may well be studied concurrently.

FUNDAMENTALS

Statements

Statements are of two main types, the branch (denoted by → and treated in the section on Defined Functions), and the specification. A typical specification statement is of the form:

```
X←3×4
```

This statement assigns the variable *X* the value resulting from the expression to the right of the specification arrow. If the variable name and arrow are omitted, the resulting value is displayed. For example:

```
      3×4
   12
```

Results displayed by the system begin at the left margin, whereas entries from the keyboard are automatically indented 2 spaces. The keyboard arrangement is shown in Part 1.

Scalar and vector constants

All numbers entered via the keyboard or displayed by the system are in decimal, either in conventional form (including a decimal point if appropriate) or in exponential form. The exponential form consists of an integer or decimal fraction followed immediately by the symbol *E* followed immediately by an integer. The integer following the *E* specifies the power of ten by which the part preceding the *E* is to be multiplied. Thus 1.44*E*2 is equivalent to 144.

Negative numbers are represented by a negative sign immediately preceding the number, e.g., ¯1.44 and ¯144E¯2 are equivalent negative numbers. The negative sign can be used only as part of a constant and is distinguished from the negation function which is denoted, as usual, by the subtraction symbol -.

A constant vector is entered by typing the constant components in order, separated by one or more spaces. A character constant is entered by typing the character between quotation marks. A sequence of characters, entered in quotes, represents a vector whose successive components are the characters themselves. Such a vector is displayed by the system as the sequence of characters, with no enclosing quotes and with no separation of the successive elements. The quote character itself must be typed in as a pair of quotes. Thus, the contraction of *CANNOT* is entered as *'CAN''T'* and is displayed as *CAN'T*.

Names and Spaces

As noted in Part 2, the name of a variable or defined function may be any letter. A letter may be any of the characters *A* to *Z*, or any one of the characters underscored, e.g., *A* or *B*. The underscored letters may be formed by overstriking or by using the ALT CODE and letter keys simultaneously.

Spaces are not required between primitive functions and constants or variables, or between a succession of primitive functions, but they may be used if desired. Spaces are needed to separate names of adjacent defined functions, constants, and variables. For example, the expression 2+3 may be entered with no spaces, but if *F* is a defined function, then the expression 2 *F* 3 must be entered with the indicated spaces. The exact number of spaces used in succession is of no importance and extra spaces may be used freely.

Overstriking and Erasure

Backspacing alone serves only to position the cursor and does not cause erasure or deletion of characters. It can be used:

1. to insert missing characters (such as parentheses) if space has previously been left for them,

2. to form compound characters by overstriking (e.g., ϕ and !),

3. to position the cursor for erasure which is effected by striking the INDEX key (erases the character at the position of the cursor and all characters to the right), and

4. in conjunction with the ALT CODE key to erase characters on the CRT only.

End of Statement

The end of a statement is indicated by striking the RETURN key. The typed entry is interpreted exactly as it appears, regardless of the time sequence in which characters were typed.

Order of execution

In a compound expression such as 3×4+6÷2, the functions are executed (evaluated) from rightmost to leftmost, regardless of the particular functions appearing in the expression. (The foregoing expression evaluates to 21.) When parentheses are used, as in the expression $W\leftarrow(3\lceil Q)\div X\times Y-Z$, the same rule applies, but, as usual, an enclosed expression must be completely evaluated before its results can be used. Thus, the foregoing expression is equivalent to $W\leftarrow((3\lceil Q)\div(X\times(Y-Z)))$.

In general, the rule can be expressed as follows: every function takes as its righthand argument the entire expression to its right, up to the right parenthesis of the pair that encloses it.

Error reports

The attempt to execute an invalid statement will cause one of the error reports given in Table 1 to be displayed. The error report will be followed by the offending statement with a caret displayed under the point in the statement where the error was detected.

If an invalid statement is encountered during execution of a defined function, the error report includes the function name and the line number of the invalid statement. The recommended procedure at this point is to enter)*PURGE*, amend the statement, and then try again. This matter is treated more fully in the section on Suspended Function Execution.

Names of primitive functions

The primitive functions of the language are summarized in Tables 2 and 8, and will be discussed individually in subsequent sections. The tables show one suggested name for each function. This is not intended to discourage the common mathematical practice of vocalizing a function in a variety of ways (for example, $X\div Y$ being expressed as "X divided by Y," or "X over Y"). Thus, the expression ρM yields the dimension of the array M, but the terms size or shape may be preferred both for their brevity and for the fact they avoid potential confusion with the dimensionality or rank of the array.

The importance of such names and synonyms diminishes with familiarity. The usual tendency is toward the use of the name of the symbol itself (e.g., "rho" (ρ) for "size," and "iota" (⍳) for "index generator"), probably to avoid unwanted connotations of any of the chosen names.

NOTE:

The symbol ↔ is used throughout the remainder of this manual to indicate that the expression to its left is equivalent to the expression to its right. This symbol is not an APL operator, it is only used to clarify definitions of APL operatons.

TABLE 1

ERROR REPORTS

TYPE	Cause: CORRECTIVE ACTION
CHARACTER	Illegitimate overstrike.
DEPTH	Excessive depth of function execution. PURGE THE STATE INDICATOR.
DOMAIN	Arguments not in the domain of the function.
DEFN	Misuse of ∇ or ⍞ symbols: 1. Use of other than the function name alone in reopening a definition. 2. Improper request for a line edit or display. 3. The function is locked.
INDEX	Index value out of range.
LENGTH	Shapes not conformable.
RANK	Ranks not conformable or resultant rank is greater than 2.
SYNTAX	Invalid syntax; e.g., two variables juxtaposed; function used without appropriate arguments as indicated by its header; unmatched parentheses or brackets.
VALUE	Use of name which has not been defined. ASSIGN A VALUE TO THE VARIABLE, OR DEFINE THE FUNCTION.
SUSPENSION	Function editing attempted while in suspension. PURGE THE STATE INDICATOR.
WS FULL	Workspace is filled (perhaps by temporary values produced in evaluating a compound expression). PURGE THE STATE INDICATOR, ERASE NEEDLESS VARIABLES, OR REVISE ALGORITHM TO USE LESS SPACE.
SYSTEM	Fault in internal operation of *APL\1500*, or possible hardware failure. RELOAD OR CLEAR AND COPY SEND TYPED RECORD, INCLUDING ALL WORK LEADING TO THE ERROR, TO THE SYSTEM MANAGER.

Monadic form fB		f	Dyadic form AfB	
Definition or example	Name		Name	Definition or example
$+B \leftrightarrow 0+B$	Identity	+	Plus	$2+3.2 \leftrightarrow 5.2$
$-B \leftrightarrow 0-B$	Negative	-	Minus	$2-3.2 \leftrightarrow {}^{-}1.2$
$\times B \leftrightarrow (B>0)-(B<0)$	Signum	×	Times	$2\times 3.2 \leftrightarrow 6.4$
$\div B \leftrightarrow 1\div B$	Reciprocal	÷	Divide	$2\div 3.2 \leftrightarrow 0.625$
B | $\lceil B$ | $\lfloor B$ 3.14 | 4 | 3 $^{-}$3.14 | $^{-}$3 | $^{-}$4	Ceiling Floor	⌈ ⌊	Maximum Minimum	$3\lceil 7 \leftrightarrow 7$ $3\lfloor 7 \leftrightarrow 3$
$*B \leftrightarrow e*B$ $e \leftrightarrow 2.71828...$	Exponential	*	Power	$2*3 \leftrightarrow 8$
$\bullet N \leftrightarrow e\bullet N$ $e \leftrightarrow 2.71828...$	Natural logarithm	●	Logarithm	$A\bullet B \leftrightarrow$ Log B base A $A\bullet B \leftrightarrow (\bullet B)\div\bullet A$
$\lvert {}^{-}3.14 \leftrightarrow 3.14$	Magnitude	|	Residue	Case: A | B $A\neq 0$: $B-(\lvert A)\times\lfloor B\div\lvert A$ $(A=0)\wedge B\geq 0$: B $(A=0)\wedge B<0$: Domain error
$!B \leftrightarrow B\times !B-1$ $!0 \leftrightarrow 1$	Factorial	!	Binomial coefficient	$A!B \leftrightarrow (!B)\div(!A)\times !B-A$ $2!5 \leftrightarrow 10$ $\quad 3!5 \leftrightarrow 10$
$?B \leftrightarrow$ Random choice from ιB	Roll	?	Deal	A mixed function (See Table 8)
$oB \leftrightarrow B\times pi$ $pi \leftrightarrow 3.14159...$	Pi times	o	Circular	See Table at lower left
$\sim 1 \leftrightarrow 0$ $\quad \sim 0 \leftrightarrow 1$	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	Relations
		≤	Not greater	Result is 1 if the relation holds, 0 if it does not:
		=	Equal	
		≥	Not less	
		>	Greater	$3\leq 7 \leftrightarrow 1$
		≠	Not equal	$7\leq 3 \leftrightarrow 0$

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

$(-A)oB$	A	AoB
$(1-B*2)*.5$	0	$(1-B*2)*.5$
Arcsin B	1	Sine B
Arccos B	2	Cosine B
Arctan B	3	Tangent B
$(^{-}1+B*2)*.5$	4	$(1+B*2)*.5$
Arcsinh B	5	Sinh B
Arccosh B	6	Cosh B
Arctanh B	7	Tanh B

Table of Dyadic o Functions

Table 2: PRIMITIVE SCALAR FUNCTIONS

SCALAR FUNCTIONS

Each of the primitive functions is classified as either scalar or mixed. Scalar functions are defined on scalar (i.e., individual) arguments and are extended to arrays in five ways: element-by-element, reduction, scan, inner product, and outer product, as described in the section on Functions on Arrays. Mixed functions are discussed in a later section.

Each scalar function is defined on real numbers or, as in the case of the logical functions and and or, on some subset of them. No functional distinction is made between "fixed point" and "floating point" numbers and the user of the terminal system need have no concern with such questions unless his work strains the capacity of the machine with respect to either space or accuracy. All numbers are carried to a precision of about 7 decimal digits.

For operations such as floor and ceiling, and in comparisons, a "fuzz" of about $7.63E^{-}6$ is applied in order to avoid anomalous results that might otherwise be brought about by doing decimal arithmetic in a binary machine.

Two of the functions of Table 2, the relationals ≠ and =, are defined on characters as well as on numbers.

Monadic and dyadic functions

Each of the functions defined in Table 2 may be used in the same manner as the familiar arithmetic functions + - × and ÷. Most of the symbols employed may denote either a monadic function (which takes one argument) or a dyadic function (which takes two arguments). For example, $\lceil Y$ denotes the monadic function ceiling applied to the single argument Y, whereas $X \lceil Y$ denotes the dyadic function maximum applied to the two arguments X and Y. Any such symbol always denotes a dyadic function if possible, i.e., it will take a left argument if one is present.

At this point it may be helpful to scrutinize each of the functions in Table 2 and to work out some examples of each, either by hand or on a terminal. However, it is not essential to grasp all of the more advanced mathematical functions (such as the hyperbolic functions sinh, cosh, and tanh) in order to proceed. Treatments of these functions are readily available in standard texts.

Certain of the scalar functions deserve brief comment. The residue function $A|B$ has the usual definition of residue used in number theory. For positive integer arguments this is equivalent to the remainder obtained by dividing B by A, and may be stated more generally as the smallest non-negative member of the set $B-N\times A$, where N is any integer.

This formulation covers the case of a zero left argument as shown in Table 2. The conventional definition is extended in two further respects:

1. The left argument *A* need not be positive; the value of the result depends only on the magnitude of *A*.

2. The argument need not be integral. For example, `1|2.6` is `0.6` and `1.5|8` is `0.5`.

The function *A*!*B* (pronounced *A* out of *B*) is defined as (!*B*)÷(!*A*)×!*B*-*A*. This is the number of combinations of *B* things taken *A* at a time.

The symbols < ≤ = ≥ > and ≠ denote the relations less than, less than or equal, etc., in the usual manner. However, an expression of the form *A*<*B* is treated not as an assertion, but as a function which yields a `1` if the proposition is true, and `0` if it is false. For example:

```
  3≤7
1
  7≤3
0
```

When applied to logical arguments (i.e., arguments whose values are limited to `0` and `1`), the six relations are equivalent to six of the logical functions of two arguments. For example, ≤ is equivalent to material implication, and ≠ is equivalent to exclusive-or. These six functions together with the and, or, nand, and nor shown in Table 2 exhaust the nontrivial logical functions of two logical arguments.

Vectors

Each of the monadic functions of Table 2 applies to a vector, element by element. Each of the dyadic functions applies element by element to a pair of vectors of equal dimension or to a scalar and a vector of any dimension, the scalar being used with each component of the vector. For example:

```
  1 2 3 4×4 3 2 1
4  6 6 4
  2+1 2 3 4
3  4 5 6
  1 2 3 4⌈2
2  2 3 4
```

Index generator

If *N* is a non-negative integer, then ⍳*N* denotes a vector of the first *N* integers. The dimension of the vector ⍳*N* is therefore *N*; in particular, ⍳1 is a vector of length one which has the value 1, and ⍳0 is a vector of dimension zero, also called an empty vector. The empty vector prints as a blank. For example:

```
      ⍳4
1  2  3  4
      ⍳5
1  2  3  4  5
      ⍳0

      6-⍳6
5  4  3  2  1  0
      2×⍳0
      2×⍳6
2  4  6  8  10  12
```

Empty vector prints as a blank

Scalar applies to all (i.e., 0) elements of ⍳0, resulting in an empty vector

The index generator is one of the class of mixed functions to be treated in detail later; it is included here because it is useful in examples.

DEFINED FUNCTIONS

Introduction

It would be impracticable and confusing to attempt to include as primitives in a language all of the functions which might prove useful in diverse areas of application. On the other hand, in any particular application there are many functions of general utility whose use should be made as convenient as possible. This need is met by the ability to define and name new functions, which can then be used with the convenience of primitives.

This section introduces the basic notions of function definition and illustrates the use of defined functions. Most of the detailed mechanics of function definition, revision, and display, are deferred to the succeeding section.

The sequence

```
     ∇S
[1]  S←4×3.14159×R×R
[2]  V←S×R÷3
[3]  ∇
```

is called a function definition; the first ∇ (pronounced del) marks the beginning of the definition and the second ∇ marks the conclusion: the name following the first ∇ (in this case S) is the name of the function defined, the numbers in brackets are statement numbers, and the accompanying statements form the body of the function definition.

The act of defining a function neither executes nor checks for validity the statements in the body; what it does is make the function name thereafter equivalent to the body. For example:

```
      ∇S                          Definition of the
[1]   S←4×3.14159×R×R             function S
[2]   V←S×R÷3
[3]   ∇
      R←2                         Specification and display
      R                           of the argument R
2
      S                           S has not yet been
VALUE ERROR                       assigned a value
      S
      ∧
      S                           Execution of S
      S                           A and V now have
50.2654                           values assigned by the
      V                           execution of S
33.5103
```

Branching

Statements in a function are normally executed in the order indicated by the statement numbers, and execution terminates at the end of the last statement in the sequence. This normal order can be modified by branches. Branches make possible the construction of iterative procedures.

The expression →4 denotes a branch to statement 4 and causes statement 4 of the function to be executed next. In general, the arrow may be followed by any expression which, to be effective, must evaluate to an integer. This value is the number of the statement to be next executed. If the integer lies outside the range of statement numbers of the body of the function, the branch ends the execution of the function.

If the value of the expression to the right of a branch arrow is a non-empty vector, the branch is determined by its first component. If the vector is empty (i.e., of zero dimension) the branch does not take place, and the normal sequence is followed.

The following examples illustrate various methods of branching used in three equivalent functions (A, B, and C) for determining S as the sum of the first N integers:

```
      ∇A
[1]   S←0
[2]   I←1
[3]   →4×I≤N                      Branch to 4×1 or to 4×0 (out)
[4]   S←S+I
[5]   I←I+1
[6]   →3                          Unconditional branch to 3
[7]   ∇
      N←1
      A
      S
1
```

```
      N←2
      A
      S
3
      N←5
      A
      S
15
      ∇B                          Equivalent to A
[1]   S←0
[2]   I←1
[3]   →0×⍳I>N                     Branch to 0 (out) or continue to next
[4]   S←S+I                         line since 0×⍳0 is an empty vector
[5]   I←I+1
[6]   →3                          Unconditional branch to 3
[7]   ∇
      N←5
      B
      S
15
      ∇C                          Equivalent to A
[1]   S←0
[2]   I←0
[3]   S←S+I
[4]   I←I+1
[5]   →3×⍳I≤N                     Branch to 3 or fall through (and out)
[6]   ∇
```

From the last two functions in the foregoing example, it should be clear that the expression ×⍳ occurring in a branch may often be read as "if". For example, →3×⍳*I*≤*N* may be read as "Branch to 3 if *I* is less than or equal to *N*".

Local and global variables

A variable is normally global in the sense that its name has the same significance regardless of what function or functions it may be used in. However, the iteration counter *I* occurring in the foregoing function A is of interest only during execution of the function; it is frequently convenient to make such a variable local to a function in the sense that it has meaning only during the execution of the function and bears no relation to any object referred to by the same name at other times. Any number of variables can be made local to a function by appending each (preceded by a semicolon) to the function header. Compare the behavior of the function D, which has a local variable *I*, with the behavior of the previously defined function C in which *I* is global:

```
      ∇D;I
[1]   S←0
[2]   I←0
[3]   S←S+I
[4]   I←I+1
[5]   →3×⍳I≤N
[6]   ∇
```

Execution of Q

```
  I←20
  N←5
  Q
  S
15
  I
20
```

Execution of C

```
  I←20
  N←5
  C
  S
15
  I
6
```

Since *I* is local to the function *Q*, execution of *Q* has no effect on the global variable *I* referred to before and after the use of *Q*.

Explicit argument

A function of the form

```
  ∇S X
[1]  S←4×3.14159×X×X
[2]  ∇
```

defines *S* as a function with an explicit argument; whenever such a function is used it must be provided with an argument. For example:

```
  S 2
  S
50.2654
  S 1
  S
12.5664
```

Any explicit argument of a function is automatically made local to the function; if *E* is any expression, then the effect of *S E* is to assign to the local variable *X* the value of the expression *E* and then to execute the body of the function *S*. Except for having a value assigned initially, the argument variable is treated as any other local variable and, in particular, may be respecified within the function.

Explicit result

Each of the primitive functions produces a result and may therefore appear within compound expressions. For example, the expression +Z produces an explicit result and may appear in a compound expression such as *X*++*Z*. A function definition of the form

```
  ∇Z←S X
[1]  Z←4×3.14159×X×X
[2]  ∇
```

defines *S* as a function with an explicit result; the variable *Z* is local, and the value it assumes at the completion of execution of the body of the function is the explicit result of the function.

For example:

```
      Q←3×⍟ 1
      Q
37.6991
      R←2
      (⍟ R)×R+3
33.5103
```

Forms of defined functions

Functions may be defined with 2,1, or 0 explicit arguments and either with or without an explicit result. The form of the header used to define each of these six types is shown in Table 3. Each of the six forms permits the appending of semicolons and names to introduce local variables. The names appearing in any one header must all be distinct; e.g., the header `Z←F Z` is invalid.

Number of Arguments	Number of Results 0	1
0	∇F	∇Z←F
1	∇F Y	∇Z←F Y
2	∇X F Y	∇Z←X F Y

Table 3: FORMS OF DEFINED FUNCTIONS

It is not necessary that the arguments or local variables be used within the body of a defined function. A function definition which does not assign a value to the result variable will cause a value error report upon completion of execution.

Use of defined functions

A defined function may be used in the same way as a primitive function. In particular, it may be used within the definition of another function. For example, the function ⍻ determines the hypotenuse of a right triangle of sides A and B by using the square root function ⍟:

```
      ∇Z←⍟ X
[1]   Z←X*.5∇

      ∇L←A ⍻ B
[1]   L←⍟ (A*2)+B*2∇

      5 ⍻ 12
13
```

A defined function must be used with the same number of arguments as appear in its header.

Recursive function definition

A function may be used in the body of its own definition, in which case the function is said to be recursively defined. The names of all defined functions are global. The following program *F* shows a recursive definition of the factorial function. The heart of the definition is statement 2, which determines factorial *N* as the product of *N* and *F N*-1, except for the case *N*=0 when the result is determined (by statement 4) as 1:

```
    ∇R←F N
[1]  →4×⍳N=0
[2]  R←N×F N-1
[3]  →0
[4]  R←1∇
```

Trace control

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

The tracing of *P* is controlled by the trace vector for *P*, denoted by *T*Δ*P*. If one types *T*Δ*P*←2 3 5 then statements 2, 3, and 5 will be traced in any subsequent execution of *P*. More generally, the value assigned to the trace vector may be any vector of integers. Typing *T*Δ*P*←0 will discontinue tracing of *P*. A complete trace of *P* is set up by entering *T*Δ*P*←⍳*N*, where *N* is the number of statements in *P*. Editing a function cancels the trace vector, if one exists.

MECHANICS OF FUNCTION DEFINITION

There are two modes of operation in the APL system: execution mode and function definition mode. In execution mode, every APL expression is executed immediately after entry. In definition mode, statements are collected to form the body of a defined function for later execution.

Function definition is opened by typing a ∇ followed by a header. The system automatically displays successive statement numbers enclosed in brackets, and accepts successive entries as the statements forming the body of the definition.

Definition mode is closed when another ∇ is entered as the last character of a statement. At that time the system returns to execution mode. After function definition has been closed, there are convenient ways to re-open the definition so that the function may be revised or displayed.

Revision

A function may be edited only during definition mode. Statements may be added, inserted, deleted, and replaced. Any statement number (including the one displayed by the system) can be overridden by typing [N], where N is any positive number less than 100, with or without a decimal point and with at most two digits to the right of the decimal point.

If any statement number is repeated, the statement following it supersedes the earlier specification of the statement. If any statement is empty -- that is, the bracketed statement number was followed by a RETURN -- the statement is deleted.

When function definition mode is ended, the statements are reordered according to their statement numbers and the statement numbers are replaced by the integers 1, 2, 3, and so on.

The particular statement on which the closing ∇ appears is not significant, since it marks only the end of the definition mode, not necessarily the last line of the function. Moreover, the closing ∇ may be entered either alone or at the end of a statement.

Reopening function definition

If a function *R* is already defined, the definition mode for that function can be re-established (edit mode) by entering ∇*R* alone; the rest of the function header must not be entered. The system responds by displaying [N+1], where N is the number of statements in *R*. Function definition then proceeds in the normal manner.

Function definition may also be established with editing or display requested on the same line. For example, ∇*R*[3]*X*←*X*+1 initiates editing by entering a new line 3 immediately. The system responds by displaying [4] and awaiting continuation. The entire process may be accomplished on a single line. Thus, ∇*R*[3]*X*←*X*+1∇ opens the definition of *R*, enters a new line 3, and terminates the definition mode.

Display

During function editing, statements which had previously defined the function are available for edit and display. Statements entered during the function definition or edit mode are not merged with the function until definition or editing is closed. This means that only the definition of the function at the last closing is available for display.

As in simple revision, any statement number can be overridden by a request for display or display and edit. This can be accomplished by one of the four methods of display or display and edit:

1. [□] Results in a full display of the defined function (including the header and the opening and closing ∇) which existed at the last closing. The system then awaits entry of additional statements.

2. [□N] Displays all statements from N onward and awaits entry of additional statements.

3. [N□] Displays statement N and awaits replacement of statement N.

4. [N□M] Initiates line editing if the input device is a typewriter. If the input device is a CRT, then a replace statement edit (see 3 above) will be effected.

The closing bracket may be followed by a ∇, in which case the display or display and edit operation returns to the execution mode after it is complete.

Line editing on a typewriter

During function definition mode, statement N can be partially modified by the following mechanism:

1. Type [N□M] where M is an integer.

2. Statement N is displayed and the carriage stops under position M.

3. A decimal digit or the symbol / may be typed under any of the positions in the displayed statement. Any other characters typed in this mode are ignored. The ordinary rules for typewriter erasure apply.

4. When RETURN is pressed, statement N is redisplayed. Each character understruck with a / is deleted and each character understruck with a digit K is preceded by K added spaces. Finally, the carriage moves to two spaces beyond the end of the line and awaits the typing of modifications to the statement in the usual manner. The final effect is to define the statement exactly as if the entry had been made entirely from the keyboard.

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

Locked functions

If the symbol ⍢ (formed by a ∇ overstruck with a ~ and called del-tilde) is used instead of ∇ to close a function definition, the function becomes locked. A locked function cannot be revised or displayed in any way. Moreover, an error stop within the function will print only the function name and statement number, not the statement.

Locked functions are used to keep a function proprietary. For example, in an exercise in which a student is required to determine the behavior of a function with a variety of arguments, locking the function prevents him from displaying its definition.

Deletion of functions and variables

A function *F* (whether locked or not) is deleted by the command *)ERASE F* (see Part 2). It may also be deleted by deleting every one of its statements. A variable may be deleted only by the *)ERASE* command.

System commands entered during function definition

A system command entered during function definition will not be executed, but will be accepted as a statement in the definition. However, system commands may not be called for execution from a function and an error report will result from an attempted execution.

SUSPENDED FUNCTION EXECUTION

Suspension

The execution of a function may be stopped before completion in two ways: by an error report or by an attention signal. In any case, the function is still active and its execution can later be resumed. In this state the function is said to be suspended. Typing →K will cause execution of the suspended function to be resumed, beginning with statement K. A branch out (→0) will terminate execution of the function.

The function I26 (described in the section on System Dependent Functions) yields the number of the next statement to be executed. Hence, the expression →I26 provides a safe way to cause normal resumption of execution.

In the suspended state almost all normal activities are possible. In particular, the system is in the following condition:

1. Expressions and most system commands can be executed. Names of local variables in the latest suspended function take precedence. Suspended or pendent functions cannot be deleted or modified in any manner.

2. No functions may be defined or edited (functions may be displayed) during any suspended state.

3. Execution may be resumed for the last suspended function at an arbitrary statement number *N* (by entering →*N*).

State Indicator

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

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

The foregoing display indicates that execution was halted during execution of statement 7 of function *H*, that the current use of function *H* was invoked in statement 2 of function *G*, and that the use of function *G* was in turn invoked in statement 3 of *F*. The * appearing to the right of *H*[7] indicates that the function *H* is suspended; The function *G* and *F* are said to be pendent.

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

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

The entire sequence can be cleared by typing)*PURGE*. If this command were entered under the conditions of the foregoing example, the state indicator would be cleared:

```
    )PURGE
    )SI
```

HOMONYMS

Variable names

The use of local variables introduces the possibility of having more than one object in a workspace with the same name. Confusion is avoided by the following rule: the local variables of the latest function being executed supersede other objects of the same name.

Function names

All function names are global. If a function *P* has a local variable *R*, then *P* could not invoke a function *R* since the name *R* would have local significance during execution of *P*.

System commands concern global objects only (see Part 2), regardless of the current environment.

INPUT AND OUTPUT

The following function determines the value of an amount *A* invested at interest *B*[1] for a period of *B*[2] years:

```
    ∇Z←A C B
[1]  Z←A×(1+.01×B[1])*B[2]∇
```

For example:

```
      1000 C 5 4
1215.51
```

The casual user of such a function might, however, find it onerous to remember the positions of the various arguments and whether the interest rate is to be entered as the actual rate (e.g., .05) or in percent (e.g., 5). An exchange of the following form might be more palatable:

```
      D
ENTER CAPITAL AMOUNT IN DOLLARS
□:
      1000
ENTER INTEREST IN PERCENT
□:
      5
ENTER PERIOD IN YEARS
□:
      4
RESULT IS 1215.51
```

It is necessary that each of the keyboard entries (1000, 5, and 4) occuring in such an exchange be accepted not as an ordinary entry (which would only result in the response 1000, etc.), but as data to be used within the function D. Facilities for this are provided in two ways, termed evaluated input, and character input.

The definition of the function D is shown later in this section.

Evaluated Input

The quad symbol □ appearing anywhere other than immediately to the left of a specification arrow accepts keyboard input as follows: the two symbols □: are displayed, and the system awaits input on the next line. Any valid expression entered at this point is evaluated and the result is substituted for the quad. For example:

```
      ∇Z←F
[1]   Z←4×□*2
[2]   ∇
      F
□:
      3
36
      F
□:
      3÷2
9
      F
□:
      (1÷4)*.5
1
```

An invalid entry in response to a request for quad input results in an appropriate error report and a re-request for input. An attempt to execute system commands or to open function definition will yield an error report since neither entry is an expression which may be evaluated. An empty input (i.e., RETURN alone or spaces and a RETURN) is rejected and the system again displays ⎕: and awaits input.

The symbols ⎕: are displayed to alert the user to the type of input expected.

Character input

The quote-quad symbol ⍞ (i.e., a quad overstruck with a quote) accepts character input: the system awaits input on the next line, at the left margin, and all data entered is accepted as characters. For example:

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

Normal output

The quad symbol appearing immediately to the left of a specification arrow indicates that the value of the expression to the right of the arrow is to be displayed. Hence, ⎕←*X* is equivalent to the statement *X*. The longer form ⎕←*X* is useful when employing multiple specification. For example, ⎕←*Q*←*X**2 assigns to *Q* the value *X**2 and then displays the value of *X**2.

The page width (measured in characters) may be set to any value *N* in the range 20-120 by entering the command *)WIDTH N*. If the input device is a CRT, then the maximum width is 40. Line width may also be dynamically set by using the System Dependent Function domino.

Heterogeneous output

A sequence of expressions separated by semi-colons will cause the values of the expressions to be displayed, with no intervening line advances or spaces except those implicit in the display of the values. The expressions need not be enclosed in parentheses.

The primary use of this form is for output in which some of the expressions yield numbers and some yield characters. For example, if *X*←¯14 and *Y*←10 , then:

```
      'THE PRODUCT OF X AND Y: ';X×Y;'=';X;'×';Y
THE PRODUCT OF X AND Y: ¯140=¯14×10
```

A further example of mixed output is furnished by the definition of the function *Q* which introduced the present section:

```
∇D;A;I;Y
[1]  'ENTER CAPITAL AMOUNT IN DOLLARS'
[2]  A←⍞
[3]  'ENTER INTEREST IN PERCENT'
[4]  I←⍞
[5]  'ENTER PERIOD IN YEARS'
[6]  Y←⍞
[7]  'RESULT IS ';A×(1+.01×I)*Y∇
```

RECTANGULAR ARRAYS

Introduction

A single element of a rectangular array can be selected by specifying its indices; the number of indices required is called the dimensionality or rank of the array. Thus, a vector is of rank 1, a matrix (in which the first index selects a row and the second a column) is of rank 2, and a scalar (since it permits no selection by indices) is an array of rank 0.

This section treats the reshaping and indexing of arrays, and the form of array output. The following section treats the five ways in which the basic scalar functions are extended to arrays, and the next section thereafter treats the definition of certain mixed functions on arrays.

Vectors, dimension, catenation

If X is a vector, then ρX denotes its dimension. For example, if X←2 3 5 7 11, then ρX is 5, and if Y←'ABC' , then ρY is 3. A single character entered in quotes or in response to a ⍞ input is a scalar, not a vector of dimension 1; this parallels the case of a single number, which is also a scalar.

Catenation chains two vectors (or scalars) together to form a vector; it is denoted by a comma. For example:

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

In general, the dimension of X,Y is equal to the total number of elements in X and Y. A numeric vector cannot be catenated with a character vector. (However, see Heterogeneous Output.)

Matrices, dimension, ravel

The monadic function ρ applied to an array A yields the size of A, that is, a vector whose components are the dimensions of A. For example, if A is the matrix

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

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

Since ⍴*A* contains one component for each coordinate of *A*, The expression ⍴⍴*A* is the rank of *A*. Table 4 illustrates the values of ⍴*A* and ⍴⍴*A* for arrays of rank 0 (scalars) thru rank 2. In particular, the function ⍴ applied to a scalar yields an empty vector.

Type of Array	⍴*A*	⍴⍴*A*	⍴⍴⍴*A*
Scalar		0	1
Vector	*N*	1	1
Matrix	*M N*	2	1

Table 4: DIMENSION AND RANK VECTORS

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

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

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

Reshape

The dyadic function ⍴ reshapes its right argument to the dimension specified by its left argument. If *M*←*D*⍴*V*, then *M* is an array of dimension *D* whose elements are the elements of *V*. For example, 2 3⍴1 2 3 4 5 6 is the matrix

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

If *N*, the total number of elements required in the array *D*⍴*V*, is equal to the dimension of the vector *V*, then the ravel of *D*⍴*V* is equal to *V*. If *N* is less than ⍴*V*, then only the first *N* elements of *V* are used; if *N* is greater than ⍴*V*, then the elements of *V* are repeated cyclically. For example, 2 3⍴1 2 is the matrix

```
1  2  1
2  1  2
```

and 3 3⍴1 0 0 0 is the identity matrix

```
1  0  0
0  1  0
0  0  1
```

More generally, if *A* is any array, then *D*ρ*A* is equivalent to *D*ρ,*A*. For example, if *A* is the matrix

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

then 3 5ρ*A* is the matrix

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

The expressions 0ρ*X* and 0 3ρ*X* and 3 0ρ*X* and 0 0ρ*X* are all valid; any one or more of the dimensions of an array may be zero. The result is an empty array.

Uses of empty arrays

A vector of dimension zero contains no components and is called an empty vector. Three expressions which yield empty vectors are ⍳0 and '' and ρ applied to any scalar. An empty vector prints a blank line.

One important use of the empty vector has already been illustrated: when one occurs as the argument of a branch, the effect is to continue the normal sequence.

The following function for determining the representation of any positive integer *N* in a base *B* number system shows a typical use of the empty vector in initializing a vector *Z* which is to be built up by successive catenations:

```
   ∇Z←B R N
[1]  Z←⍳0
[2]  Z←(B|N),Z
[3]  N←⌊N÷B
[4]  →2×N>0 ∇
  10 R 1776
1  7  7  6
  8 R 1776
3  3  6  0
```

Empty arrays of higher rank can be useful in analogous ways in conjunction with the mesh function described in the section on Mixed Functions.

Indexing

If *X* is a vector and *I* is a scalar, then *X*[*I*] denotes the *I*th element of *X*. For example, if *X*←2 3 5 7 11 then *X*[2] is 3.

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

In general, ρ*X*[*I*] is equal to ρ*I*. In particular, if *I* is a scalar, then *X*[*I*] is a scalar, and if *I* is a matrix, then *X*[*I*] is a matrix. For example:

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

3 1 4
2 1 4
4 1 2
4 1 4
      A[M]

CAD
BAD
DAB
DAD
```

If *M* is a matrix, then *M* is indexed by a two-part list of the form *I*;*J* where *I* selects the row (or rows) and *J* selects the column (or columns). For example, if *M* is the matrix used in the example above, then *M*[3;3] is the element 2 and *M*[1 3 4;1 3] is the matrix

```
3 4
4 2
4 4
```

In general, ρ*M*[*I*;*J*] is equal to (ρ*I*),ρ*J*. Hence, if *I* and *J* are both vectors, then *M*[*I*;*J*] is a matrix; if both *I* and *J* are scalars, *M*[*I*;*J*] is a scalar; if *I* is a vector and *J* is a scalar (or vice versa), *M*[*I*;*J*] is a vector. The indices are not limited to vectors, but may be of higher rank. For example, if *I* is a 3 by 4 matrix, and *J* is a scalar, then *M*[*I*;*J*] is of dimension 3 4, and *M*[*J*;*I*] is of dimension 3 4.

The form *M*[*I*;] indicates that all columns are selected, and the form *M*[;*J*] indicates that all rows are selected. For example, *M*[2;] is 2 1 4 and *M*[;2 1] is

```
1 3
1 2
1 4
1 4
```

Permutations are an interesting use of indexing. A vector *P* whose elements are some permutation of its own indices is called a <u>permutation of order</u> ρ*P*. For example, 3 1 4 2 is a permutation of order 4. If *X* is any vector of the same dimension as *P*, then *X*[*P*] produces a permutation of *X*. Moreover, if ρ*P* is equal to (ρ*M*)[1], then *M*[*P*;] permutes the column vectors of *M* (i.e., interchanges the rows of *M*) and is called a <u>column permutation</u>. Similarly, if ρ*P* equals (ρ*M*)[2], then *M*[;*P*] is a <u>row permutation</u> of *M*.

Indexing on the left

An array appearing to the left of a specification arrow may be indexed, in which case only the selected portions are affected by the specification. For example:

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

The normal restrictions on indexing apply; in particular, a variable which has not already been assigned a value cannot be indexed, and an out-of-range index value cannot be used.

Index origin

In 1-origin indexing, $X[1]$ is the leading element of the vector X and $X[\rho X]$ is the last element. In 0-origin indexing, $X[0]$ is the leading element and $X[^{-}1+\rho X]$ is the last. 0-origin indexing is instituted by the command *)ORIGIN 0*. The command *)ORIGIN 1* restores 1-origin indexing. The index origin in effect applies to all coordinates of rectangular arrays. Index origin may be changed dynamically by the System Dependent Function domino.

In certain expressions such as $+/[J]M$, $+\backslash[J]M$, and $X\phi[J]M$ (to be treated more fully in the two following sections), the value of J determines the coordinate of the array M along which the function is to be applied. Since the numbering of coordinates follows the index origin, a change of index origin also affects the behavior of such expressions.

The index origin also affects six other functions, the monadic and dyadic forms of ? and ⍳, and ⍋ and ⍒. The expression ιN yields a vector of the first N integers beginning with the index origin. Hence $X[\iota N]$ selects the first N components of X in either origin. Moreover, ⍳1 is a one-element vector having the value 0 in 0-origin and 1 in 1-origin; ⍳0 is an empty vector in either origin.

The index origin remains associated with a workspace; in particular, the index origin of an active workspace is not affected by a copy command. A clean workspace provided at sign-on or by the command *)CLEAR* is in 1-origin. All definitions and examples in this text are expressed in 1-origin.

Array output

Character arrays print with no spaces between components in each row; other arrays print with at least one space between components. If a vector or a row of a matrix requires more than one line, succeeding lines are indented.

A matrix prints with all columns aligned and with a blank line before the first row. A matrix of dimension $N,1$ prints as a single column.

FUNCTIONS ON ARRAYS

There are five ways in which the scalar functions of Table 2 extend to arrays: element-by-element, reduction, scan, inner product, and outer product. Reduction, scan, and outer product are defined on any arrays, but the other two extensions are defined only on arrays whose sizes satisfy a certain relationship called conformability. For the element-by-element extension, conformability requires that the shapes of the arrays agree, unless one is a scalar or one-element array. The requirements for inner product are shown in Table 6.

Scalar functions

All of the scalar functions of Table 2 are extended to arrays element by element. Thus if *M* and *N* are matrices of the same size, f is a scalar function, and *P*←*M*f*N*, then *P*[*I*;*J*] equals *M*[*I*;*J*]f*N*[*I*;*J*], and if *Q*←f*N*, then *Q*[*I*;*J*] is equal to f*N*[*I*;*J*].

If *M* and *N* are not of the same size, then *M*f*N* is undefined (and yields a length or rank error report) unless one or the other of *M* and *N* is a scalar or one-element array, in which case the single element is applied to each element of the other argument. In particular, a scalar versus an empty array produces an empty array.

An expression or function definition which employs only scalar function and scalar constants extends to arrays like a scalar function.

Reduction

The sum-reduction of a vector *X* is denoted by +/*X* and defined as the sum of all components of *X*. More generally, for any scalar dyadic function f, the expression f/*X* is equivalent to *X*[1]f*X*[2]f...f*X*[ρ*X*], where evaluation is from rightmost to leftmost as usual. A user defined function cannot be used in reduction.

If *X* is a vector of dimension zero, then f/*X* yields the identity element of the function f (listed in Table 5) if it exists; if *X* is a scalar or vector of dimension 1, then f/*X* yields the value of the single element of *X*.

The result of reducing any vector or scalar is a scalar.

Dyadic Function		Identity Element		Left-Right
Times	×	1		L R
Plus	+	0		L R
Divide	÷	1		R
Minus	-	0		R
Power	*	1		R
Logarithm	⍟	1		None
Maximum	⌈	¯1.7014...*E*38		L R
Minimum	⌊	1.7014...*E*38		L R
Residue	|	0		L
Circle	○	0		None
Out of	!	1		L
Or	∨	0		L R
And	∧	1		L R
Nor	⍱	0		None
Nand	⍲	0		None
Equal	=	1	Apply	L R
Not equal	≠	0	for	L R
Greater	>	0	logical	R
Not less	≥	1	arguments	R
Less	<	0	only	L
Not greater	≤	1		L

Table 5: IDENTITY ELEMENTS OF PRIMITIVE SCALAR DYADIC FUNCTIONS

For a matrix *M*, reduction can proceed along the first coordinate (denoted by f/[1]*M*) or along the second coordinate (f/[2]*M*). The result in either case is a vector; in general, reduction applied to any non-scalar array *A* produces a result of rank one less than the rank of *A* (hence the term reduction). The numbering of coordinates follows the index origin.

Since +/[1]*M* scans over the row index of *M*, it sums each column vector of *M*, and +/[2]*M* sums each row vector of *M*. For example, if *M* is the matrix

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

then +/[1]*M* is 5 7 9 and +/[2]*M* is 6 15.

In reducing along the last coordinate of an array, the coordinate indicator may be elided -- thus, +/*M* denotes summing over each of the rows of *M* and +/*V* denotes summing over the last (and only) coordinate of the vector *V*.

Scan

Generally, for any scalar dyadic function f, the expression f\X yields a result Q where ρQ is equal to ρX.

If Q←f\X is an expression where X is a vector, then Q[I] is equivalent to f/X[⍳I] where I is in the set ⍳ρX. For example, if V is the vector 1 3 5 7, then +\V will yield the result 1 4 9 16.

For a matrix M, scan can proceed along the first coordinate (denoted by f\[1]M) or along the second coordinate (f\[2]M or f\M). For example, if M is the matrix

```
64  16   4
 8   4   2
```

then +\[1]M is the matrix

```
64  16   4
 8   4   2
```

and +\M (or +\[2]M) is the matrix

```
64   4  16
 8   2   4
```

Inner product

The familiar matrix product is denoted by C←A+.×B. If A and B are matrices, then C is a matrix such that C[I;J] is equal to +/A[I;]×B[;J]. A similar definition applies to Af.gB where f and g are any of the standard scalar dyadic functions.

If A is a vector and B is a matrix, then C is a vector such that C[J] is equal to +/A×B[;J]. If B is a vector and A is a matrix, then C is a vector such that C[I] is equal to +/A[I;]×B. If both A and B are vectors, then A+.×B is the scalar +/A×B.

The last dimension of the pre-multiplier A must equal the first dimension of the post-multiplier B, except that if either argument is a scalar, it is extended in the usual way. For non-scalar arguments, the dimension of the result is equal to (¯1↓ρA),1↓ρB. (<u>See</u> the function <u>drop</u> in the section on Mixed Functions.) In other words, the dimension of the result is equal to (ρA),ρB except for the two inner dimensions (¯1↑ρA and 1↑ρB), which must agree and which are eliminated by the reduction over them. Definitions for the various cases are shown in Table 6.

Outer product

The outer product of two arrays X and Y with respect to a standard scalar dyadic function g is denoted by X∘.gY and yields an array of dimension (ρX),ρY, formed by applying g to every pair of components of X and Y, providing the rank of the result is not greater than 2. <u>See</u> Table 7 for definitions of various cases.

ρA	ρB	ρAf.gB	Conformability requirements	Definition Z←Af.gB
	V			Z←f/AgB
U				Z←f/AgB
U	V		U=V	Z←f/AgB
	V W	W		Z←f/AgB
T U		T		Z[I]←f/AgB[;I]
U	V W	W	U=V	Z[I]←f/A[I;]gB
T U	V	T	U=V	Z[I]←f/AgB[;I]
T U	V W	T W	U=V	Z[I]←f/A[I;]gB
				Z[I;J]←f/A[I;]gB[;J]

Table 6: INNER PRODUCTS FOR PRIMITIVE SCALAR DYADIC FUNCTIONS f AND g

If X and Y are vectors and Z←X∘.gY, then Z[I;J] is equal to X[I]gY[J]. For example:

```
      X←⍳3
      Y←⍳4
      X∘.×Y

1  2  3  4
2  4  6  8
3  6  9 12
      X∘.≥Y

1 0 0 0
1 1 0 0
1 1 1 0
```

ρA	ρB	ρA∘.gB	Definition Z←A∘.gB
	V	V	Z←AgB
U		U	Z[I]←AgB[I]
U	V	U V	Z[I]←A[I]gB
	V W	V W	Z[I;J]←A[I]gB[J]
T U		T U	Z[I;J]←AgB[I;J]
			Z[I;J]←A[I;J]gB

Table 7: OUTER PRODUCTS FOR PRIMITIVE SCALAR DYADIC FUNCTION g

MIXED FUNCTIONS

Introduction

The scalar functions listed in Table 2 each take a scalar argument (or arguments) and yield a scalar result; each is also extended element by element to arrays. The mixed functions of Table 8, on the other hand, may be defined on vector arguments to yield a scalar result or a vector result, or may be defined on scalar arguments to yield a vector result. In extending these definitions to arrays of higher rank, it may therefore be necessary to specify over which coordinate of an array the mixed function is to be applied. The expression [*J*] following a function symbol indicates that the function is applied to the *J*th coordinate. If the expression is elided, the function applies to the last coordinate of the argument array. These conventions agree with those used earlier in reduction. The numbering of coordinates follows the index origin.

Transpose

The expression ⍉*A* yields the array *A* with the last two coordinates interchanged. For a scalar *S*, vector *V*, and matrix *M*, the following relations hold:

⍉*S* is equivalent to *S*
⍉*V* is equivalent to *V*
⍉*M* is equivalent to ordinary matrix transpose.

Rotate

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

If the rank of *X* is 2, then the coordinate *J* along which rotation is to be performed may be specified by the form *Z*←*K*⌽[*J*]*X*. Moreover, the dimension of *K* must equal the remaining dimension of *X*, and each vector along the *J*th coordinate of *X* is rotated as specified by the corresponding element of *K*. For example, if ⍴*X* is 3 4 and *J* is 2, then *K* must be of dimension 3 and *Z*[*I*;] is equal to *K*[*I*]⌽*X*[*I*;]. If *J* is 1, then ⍴*K* must be 4, and *Z*[;*I*] is equal to *K*[*I*]⌽*X*[;*I*]. A scalar *K* is extended in the usual manner. The following are examples of rotate:

M

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

0 1 2 3⌽[1]*M*

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

1 2 3⌽[2]*M*

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

Reverse

If *X* is a vector and *R*←⌽*X*, then *R* is equal to *X* except that the elements appear in reverse order. Formally, *R* is equal to *X*[1+(⍴*X*)-⍳⍴*X*]. In 0-origin indexing, the appropriate expression is *X*[¯1+(⍴*X*)-⍳⍴*X*].

If *A* is any array, *J* is a scalar or one-element array, and *R*←⌽[*J*]*A*, then *R* is an array like *A* except that the order of the elements is reversed along the *J*th coordinate. For example:

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

The expression ⌽*A* denotes reversal along the last coordinate of *A*.

Compress

The expression *U*/*X* denotes compression of *X* by *U*. If *U* is a logical vector (comprising elements having only the values 0 or 1) and *X* is a vector of the same dimension, then *U*/*X* produces a vector result of +/*U* elements chosen from those elements of *X* corresponding to non-zero elements of *U*. For example, if *X*←2 3 5 7 11 and *U*←1 0 1 1 0 then *U*/*X* is 2 5 7 and (~*U*)/*X* is 3 11.

To be conformable, the dimensions of the arguments must agree, except that a scalar (or one-component array) left argument is extended to apply to all elements of the right argument. Hence 1/*X* is equal to *X* and 0/*X* is an empty vector. A scalar right argument is extended. The result in every case is a vector.

If *M* is a matrix, then *U*/[1]*M* denotes compression along the first coordinate, that is , the compression operates on each column vector and therefore deletes certain rows. It is called column compression. Similarly, *U*/[2]*M* (or simply *U*/*M*) denote row compression. The result in every case is a matrix. As in reduction, *U*/*M* denotes compression along the last coordinate. For example:

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

Mesh

Mesh is denoted by *U**X* where *U* is a logical (in the set 0 1) scalar or vector, and where *X* is an arbitrary array. A scalar left argument is not extended, but is treated as a one-component vector. If *X* is not a matrix, then ⍴*U**X* is equal to ⍴*U*. If *X* is a matrix, *U*\[*J*]*X* denotes mesh along the *J*th coordinate (*U**X* denotes mesh along the last coordinate), and the *J*th dimension of the result is ⍴*U*; the other result dimension is the dimension of the non-meshed coordinate of *X*.

Let *P* be the number of ones in *U* (*P*←+/*U*) and let *Q* be the number of zeros in *U* (*Q*←+/~*U*). Also, let *K* be the mesh identity element such that if the right argument *X* of *U**X* is a numeric array, then *K* is a 0 (*K*←0), and if *X* is a literal array, then *K* is a blank (*K*←' '). In particular, in the expression *U**V* let *V* be a vector partitioned into two subvectors *Y* and *Z* by the following rules. If:

1.	0=ρ*V*	*Y* ↔ *P*ρ*K*	*Z* ↔ *Q*ρ*K*
2.	*P*≥ρ*V*	*Y* ↔ *P*ρ*V*	*Z* ↔ *Q*ρ*K*
3.	*P*<ρ*V*	*Y* ↔ *P*ρ*V*	*Z* ↔ *Q*ρ*P*↓*V*

Then, each 1 in *U* selects from *Y* (the first substring), and each 0 in *U* selects from *Z* (the second substring). For example:

```
  1 0 1 1 0\⍳0                Case 1: Y ↔ 0 0 0    Z ↔ 0 0
0  0  0  0  0

  1 0 1 1 0\2 3               Case 2: Y ↔ 2 3 2    Z ↔ 0 0
2  0  3  2  0

  1 0 1 1 0\1 2 3 4 5 6 7     Case 3: Y ↔ 1 2 3    Z ↔ 4 5
1  4  2  3  5

  1 0 1 1 0\1 2 3 4           Case 3: Y ↔ 1 2 3    Z ↔ 4 4
1  4  2  3  4
```

If the right argument *M* is a matrix, then the following equivalences hold:

(*U**M*)[*I*;] ↔ (*U*\[2]*M*)[*I*;] ↔ *U**M*[*I*;]

(*U*\[1]*M*)[;*I*] ↔ *U*\[1]*M*[;*I*] ↔ *U**M*[;*I*]

For example:

```
  M                0 1 1 1 0\M

CAD()                (CAD)
BAT()                (BAT)
END()                (END)
```

All argument lengths are conformable for mesh.

When *P*=ρ*V*, mesh is the converse of compression:

```
  1 0 1 1 0 0 1 0 1/1 0 1 1 0 0 1 0 1\⍳5
1 2 3 4 5
```

Prefix

Prefix produces a logical array (elements are only 0 and 1) from the expression $S\alpha X$ where S must be a scalar and X may be a scalar or a vector. A one-component vector is treated as a scalar. Prefix is defined as $S\alpha X \leftrightarrow X\circ.\geq\iota S$. For example:

```
      5α3
1 1 1 0 0
      5α⍳5

1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 1
```

Suffix

Suffix is formally defined as the reversal of prefix. The expression $S\omega X$ is equivalent to $\phi S\alpha X$.

Decode

The expression $R\perp X$ denotes the value of the array X evaluated in a number system with radices $R[1],R[2],\ldots,R[\rho R]$. For example, if $R\leftarrow$24 60 60 and $X\leftarrow$1 2 3 is a vector of elapsed time in hours, minutes, and seconds, then $R\perp X$ has the value 3723, and is the corresponding elapsed time is seconds. In the same manner, 10 10 10 10⊥1 7 7 6 is equal to 1776, and 2 2 2⊥1 0 1 is equal to 5. Formally, $R\perp X \leftrightarrow +/X\times\phi\times\backslash\phi 1\downarrow R,1$. If X is a scalar, the result is a scalar; otherwise, $\rho R\perp X \leftrightarrow {}^{-}1\downarrow\rho X$.

The arguments R and X are conformable if ρR is equal to ${}^{-}1\downarrow\rho X$; scalar arguments are extended in the usual way. If X is a scalar, then $X\perp C$ is the value of the polynomial in X with coefficients C, arranged in order of decending powers of X. For example, the polynomial $(X*3)+(3\times X*2)-7$ may be evaluated for a scalar X by the expression $X\perp$1 3 0 ¯7. The arguments are not restricted to integer values.

For a matrix right argument, $(R\perp M)[I] \leftrightarrow R\perp M[I;]$; decode is not subscriptable. The following is an example of decode:

```
      M←2 3⍴⍳6
      M
1 2 3
4 5 6
      10⊥M
123 456
```

The decode function is commonly applied in work with fixed-base number systems and is often called the base value function.

Encode

The encode function $R\tau X$ denotes the representation of X in the base-R number system; encode is the converse of decode. For example, 2 2 2 2τ5 is 0 1 0 1 and 2 2 2τ5 is 1 0 1 and 2 2τ5 is 0 1. If X is a negative number, then $R\tau X$ is the base-R complement representation of $|X$; for example, $(8\rho 2)\tau^{-}5$ is 1 1 1 1 1 0 1 1.

The dimension of $R\tau X$ is $(\rho X),\rho R$, except that a one-component right argument is treated as a scalar. For a vector right argument the result is a matrix and $(R\tau V)[I;] \leftrightarrow R\tau V[I]$; encode is not subscriptable. The following is an example of encode:

```
      10 10 10τ123 456 ¯1

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

The encode function may also be called representation.

Index of

If V is a vector and S is a scalar, then $J\leftarrow V\iota S$ yields the position of the earliest occurrence of S in V. If S does not equal any element of V, then J has the value $(\iota 1)+\rho V$. Clearly, this value depends, as does any result of this function, on the index origin, and is one greater than the largest permissable index of V.

If S is a vector, then J is a vector such that $J[I]$ is the index in V of $S[I]$. For example:

```
      'ABCDEFGH'ι'GAFFE'
  7  1  6  6  5
```

If X is a numerical vector, then the expression $X\iota\lceil/X$ yields the index of the (first) maximum element in X. For example, if X is the vector 8 3 5 13 2 7 9, the $\lceil/X$ is 13 and $X\iota\lceil/X$ is 4.

The result in every case has the same dimensions as the righthand argument of ι. For example, if $Z\leftarrow V\iota S$, and S is a matrix, then $Z[I;J]$ is equal to $V\iota S[I;J]$.

Membership

The function $X\epsilon Y$ yields a logical array of the same dimension as X. Any particular element of $X\epsilon Y$ has the value 1 if the corresponding element of X belongs to Y, that is, if it occurs as some element of Y. For example, $(\iota 7)\epsilon 3\ 5$ is equal to 0 0 1 0 1 0 0 and 'ABCDEFGH'ε'COFFEE' equals 0 0 1 0 1 1 0 0.

If the vector U represents the universal set in some finite universe of discourse, then $U\epsilon A$ is the characteristic of the set A, and the membership function is therefore also called the characteristic function.

The size of the result of the function ϵ is determined by the size of the left argument, whereas the size of the result of the dyadic function ι is determined by the size of the right argument. However, the left arguments of both frequently play the role of specifying the universe of discourse.

Take and Drop

If V is a vector and S is a scalar between 0 and ρV, then $S\uparrow V$ takes the first S components of V. For example, if $V\leftarrow\iota 7$, then $3\uparrow V$ is 1 2 3 and $0\uparrow V$ is $\iota 0$, and $8\uparrow V$ yields a domain error.

If S is chosen from the set $-\iota\rho V$, then $S\uparrow V$ takes the last $|S$ elements of V. For example, $\bar{}3\uparrow V$ is 5 6 7.

If N is a scalar, then $S\uparrow N$ is valid only if S is an empty vector ($\iota 0$). The result of $(\iota 0)\uparrow N$ is N.

If A is an array, then $W\uparrow A$ is valid only if W has one element for each dimension of A, and $W[I]$ determines what is to be taken along the Ith coordinate of A. For example, if $A\leftarrow 3\ 4\rho\iota 12$, then $2\ \bar{}3\uparrow A$ is the matrix

```
2  3  4
6  7  8
```

The function drop ($\downarrow$) is defined analogously, except that the indicated number of elements are dropped rather than taken. For example, $\bar{}1\ 1\downarrow A$ is the same matrix as the one displayed in the preceding paragraph.

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

Grade up and down

The function $\grave{A}V$ produces the permutation which would order V, that is $V[\grave{A}V]$ is in ascending order. For example, if V is the vector 7 1 16 5 3 9, then $\grave{A}V$ is the vector 2 5 4 1 6 3, since 2 is the index of the first in rank, 5 is the index of the second in rank, and so on. The symbol $\grave{A}$ is formed by overstriking | and Δ.

If P is a permutation vector, then $\grave{A}P$ is the permutation inverse to P. If a vector D contains duplicate elements, then the ranking among any set of equal elements is determined by their positions in D. For example, $\grave{A}5\ 3\ 7\ 3\ 9\ 2$ is the vector 6 2 4 1 3 5.

The right argument of grade up or grade down is only valid if it is a vector.

The grade down function $\forall$ is the same as the function $\grave{A}$ except that the grading is determined in descending order. Because of the treatment of duplicate items, the expression $\wedge/(\grave{A}V)=\phi\forall V$ has the value 1 if and only if the elements of the vector V are all distinct.

Deal

The function *M?N* produces, for a scalar *N*, a vector of dimension *M* obtained by making *M* random selections, without replacement, from the population ⍳*N*; therefore, *M* must be in the set 0,⍳*N*. In particular, *N?N* yields a random permutation of order *N*. The left argument is limited to a scalar or one-component array; the right argument may be a scalar or vector.

If *V* is a vector, then (*S?V*)[*I*;] ↔ *S?V*[*I*]. For example:

```
      5?5 6 7

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

Comments

The lamp symbol ⍝, formed by overstriking ∩ and ∘, signifies that what follows it is a comment, for illumination only and not to be executed. The lamp symbol may occur only as the first character in a statement, but may be used in defined functions. Comments may not be entered during evaluated (⍎) input.

MULTIPLE SPECIFICATION

Specification (←) may (like any other function) occur repeatedly in a single statement. For example, the execution of the statement *Z*←*X*×*A*←3 will assign to *A* the value 3, then multiply this assigned value of *A* by *X* and assign the resulting value to *Z*.

Multiple specification is useful for initializing variables. For example, *X*←*Y*←1+*Z*←0 assigns 0 to *Z* and 1 to both *X* and *Y*.

A branch may occur in a statement together with one or more specifications, provided that the branch is the last operation to be executed (i.e., the leftmost). For example, the statement →*S*×⍳*N*>*I*←*I*+1 first augments *I*, and then branches to statement *S* if *N* exceeds the new value of *I*.

Monadic form *f B*			*f*	Dyadic form *A f B*		
Definition or example	Syntax	Name		Name	Syntax	Definition or example

`ρA ↔ ⍳0`
`ρB ↔ 4`
`ρC ↔ 3 4`
	`ρM`	Size	ρ	Reshape	`VρM`	`5ρC ↔ 'ABCDE'`
						`2 3ρB ↔ 2 3 1`
						`4 2 3`

`,M ↔ (×/ρM)ρM`
`,C ↔ 12ρC`
`ρ,A ↔ 1`
	`,M`	Ravel	,	Catenate	`V,V`	`4 1,A ↔ 4 1 3`
						`0,⍳2 ↔ 0 1 2`
						`ρA,⍳0 ↔ 1`

`⍳4 ↔ 1 2 3 4`
`⍳0 ↔ an empty vector`
`6-⍳5 ↔ 5 4 3 2 1`
	`⍳S`	Index generator	⍳	Index of	`V⍳M`	`B⍳3 7 ↔ 2 5`
						`B⍳B ↔ 1 2 3 4`
						`L⍳L ↔ 1 2 2 1`

`⌽V ↔ V[(ρV)+1-⍳ρV]`
`⌽B ↔ 4 1 3 2`
`⌽C ↔ DCBA  ⌽[1]C ↔ IJKL`
`HGFE  EFGH`
`LKJI  ABCD`
	`⌽M`	Reverse	⌽	Rotate	`V⌽M`	`S⌽V ↔ V[1+(ρV)|S-1-⍳ρV]`
						`3⌽B ↔ 4 2 3 1 ↔ ¯1⌽B`
						`¯1 0 1⌽C ↔ DABC`
						`EFGH`
						`JKLI`

Scalar function See Table 2	`?M`	Roll	?	Deal	`S?V`	`S?V ↔` Random deal of `S` elements from `⍳V`. If `V` is a vector: `(S?V)[I;] ↔ S?V[I]`

`⍉A ↔ A` `⍉B ↔ B`
`⍉C ↔ AEI`
`BFJ`
`CGK`
`(⍉M)[I;J] ↔ M[J;I]` `DHL`
	`⍉M`	Transpose	⍉			

`B[⍋B] ↔ 1 2 3 4` `⍋'CDAEAC' ↔ 3 5 1 6 2 4`	`⍋V`	Grade up	⍋			

`V[⍒V] ↔ ⌽V[⍋V]`
`B[⍒B] ↔ 4 3 2 1`
`⍒'CDAEAC' ↔ 4 2 1 6 3 5`
	`⍒V`	Grade down	⍒			

DEFINITIONS USED IN EXAMPLES

`A←3`
`B←2 3 1 4`
`C←3 4ρ'ABCDEFGHIJKL'`
`L←1 0 0 1`
S ↔ a scalar.
V ↔ a scalar or vector.
V ↔ a vector only.
M ↔ a scalar, vector, or matrix.
M ↔ a matrix only.
I ↔ a scalar integer.

Table 8: PRIMITIVE MIXED FUNCTIONS

Monadic form fB			f	Dyadic form AfB		
Definition or example	Syntax	Name		Name	Syntax	Definition or example
NOTES 1. The functions ιS, $\underline{V}\iota M$, $\underline{V}[M]$, $\underline{M}[M;M]$, $?M$, $S?V$, $\Grade{\underline{V}}$, $\Gradedown{\underline{V}}$, and operator subscripting are index origin dependent. 2. Elision of any index in subscripting selects all along that coordinate. 3. The functions ϕM, $V\phi M$, V/M, $V\backslash M$, $S?V$, $S\alpha V$, $S\omega V$, $V\tau V$, and $V\perp M$ apply along the last coordinate of the indicated matrix. The functions $\phi[I]M$, $V\phi[I]M$, $V/[I]M$, and $V\backslash[I]M$ indicate that the function is to be applied along the Ith coordinate. The functions $S?V$, $S\alpha V$, $S\omega V$, $V\tau V$, and $V\perp M$ can not be subscripted. 4. The functions $\Grade{\underline{V}}$ and $\Gradedown{\underline{V}}$ sort literal arguments according to an EBCDIC collating sequence. Thus, if X is a matrix of names, the expression $X[\Grade{X}[;I];]$ will sort X by the Ith letter. 5. The left argument of $V\rho M$ is restricted to two or less elements; arrays of rank three or higher are not permitted. 6. The right argument of $,M$ must not have more than 255 elements; vectors of lengths longer than 255 are not permitted.						
			[	Index	$\underline{V}[M]$ $\underline{M}[M;M]$	$B[2] \leftrightarrow 3$ $B[] \leftrightarrow B$ $C[2;] \leftrightarrow C[2;\iota 4] \leftrightarrow$ 'EFGH' $C[2;2\ 3\rho B] \leftrightarrow FGE$ HFG
			↑	Take	$V\uparrow M$	$3\uparrow B \leftrightarrow 2\ 3\ 1$ $2\ 2\uparrow C \leftrightarrow AB$ $\bar{3}\uparrow B \leftrightarrow 3\ 1\ 4$ EF
			↓	Drop	$V\downarrow M$	$2\downarrow B \leftrightarrow 1\ 4$ $1\ \bar{1}\downarrow C \leftrightarrow EFG$ $\bar{2}\downarrow B \leftrightarrow 2\ 3$ IJK
			/	Compress	V/M	$L/B \leftrightarrow 2\ 4$ $L/A \leftrightarrow 3\ 3$ $1\ 0\ 1/[1]C \leftrightarrow ABCD$ $0/B \leftrightarrow \iota 0$ $IJKL$
			\	Mesh	$V\backslash M$	$L\backslash\iota 4 \leftrightarrow 1\ 3\ 4\ 2$ $L\backslash$'AB-' $\leftrightarrow$ 'A--B' $L\backslash L/B \leftrightarrow 2\ 0\ 0\ 4$
			∈	Membership	$M\in M$	'QUICK'$\in C \leftrightarrow 0\ 0\ 1\ 1\ 1$ $C\in$'QUICK' $\leftrightarrow 0\ 0\ 1\ 0$ $0\ 0\ 0\ 0$ $1\ 0\ 1\ 0$
			τ	Encode	$V\tau V$	$2\ 2\ 2\tau 1\ 6\ 2 \leftrightarrow 0\ 0\ 1$ $2\ 3 \leftrightarrow 10\ 10\tau\underline{1}23$ $1\ 1\ 0$ $9\ 9 \leftrightarrow 10\ 10\tau\bar{1}$ $0\ 1\ 0$
			⊥	Decode	$V\perp M$	$B \leftrightarrow 2\perp 2\ 2\ 2\tau B$ $9 \leftrightarrow 2\perp L$ $(4\rho 60)\tau 761167 \leftrightarrow 3\ 31\ 26\ 7$
			α	Prefix	$S\alpha V$	$S\alpha I \leftrightarrow I\geq\iota S$ $S\alpha\underline{V} \leftrightarrow \underline{V}\circ.\geq\iota S$ $6\alpha 2 \leftrightarrow 1\ 1\ 0\ 0\ 0\ 0$
			ω	Suffix	$S\omega V$	$S\omega V \leftrightarrow \phi S\alpha V$ $3\omega 1 \leftrightarrow 0\ 0\ 1$

Table 8 PRIMITIVE MIXED FUNCTIONS (continued)

SYSTEM DEPENDENT FUNCTIONS

There are three main types of information about the state of the system which are of value to the user:

1. General information common to all users, such as the date, time of day, and the port numbers of all signed-on terminals.

2. Information specific to the particular work session, such as the time of sign-on, the central computer time used, the total input wait time, and the input device type.

3. Information specific to the active workspace, such as the amount of storage available, the condition of the state indicator, and the number of significant digits to be displayed during output.

The functions I-beam and Domino provide the user with the facility to examine system information and modify workspace or work session parameters.

I-Beam

The function ⌶S fetches system information; the result is selected by the right argument, which must be a one-component array. The ⌶ is formed by overstriking ⊤ and ⊥. Table 9 is a summary of the I-beam function. ⌶9 and ⌶27 yield vectors; all other results are scalar. Times are all in units of one-sixtieth of a second, the date is given as a six digit integer in which the successive digit pairs specify the month, day, and year, and the available storage is given in bytes.

The byte is a unit of storage equal to eight binary digits. A variable requires four bytes of overhead, plus four bytes per element. A defined function requires seven bytes of overhead, one byte for each local variable, one byte for each line in the function, plus one byte for each character in the body of the function; the total is then raised to the next highest multiple of ten.

In designing an algorithm for a particular purpose, it frequently happens that one may trade time for space; that is, an algorithm which requires little computer time may require more storage space for intermediate results, and an algorithm which requires little storage may be less efficient in terms of time. Hence, the information provided by the functions ⌶21 (central computer time used) and ⌶22 (available storage for arrays and function execution) may be helpful in designing algorithms. Moreover, since the functions ⌶21 and ⌶22 can, like all of the I-beam functions, be used within a defined function, they can be used to make the execution depend upon the space available or the computer time used.

Input wait time is defined as the total accumulated time since sign-on during which the keyboard has been unlocked. The associated function (⌶19) may be used in conjunction with ⍞ or ⎕ to determine the amount of time taken by a student in responding to a question. The following is an example of the use of ⌶19:

```
      ∇Q;N;A;B;T
[1]   N←0
[2]   A←¯1+?10
[3]   B←¯1+?10
[4]   T←⌶19
[5]   A;' × ';B;' = ?'
[6]   →(□=A×B)/9
[7]   'WRONG, TRY AGAIN.'
[8]   →4
[9]   'CORRECT!    TIME= ';⌊.5+((⌶19)-T)÷60;' SECONDS.'
[10]  →(5>N←N+1)/2
[11]  'EXERCISE COMPLETED.'∇
      Q
4 × 2 = ?
□:
  8
CORRECT!    TIME= 2 SECONDS.
2 × 3 = ?
□:
  7
WRONG, TRY AGAIN.
2 × 3 = ?
□:
  6
CORRECT!    TIME= 1 SECONDS.
8 × 9 = ?
□:
  72
CORRECT!    TIME= 3 SECONDS.
4 × 5 = ?
□:
  20
CORRECT!    TIME= 3 SECONDS.
0 × 7 = ?
□:
  0
CORRECT!    TIME= 1 SECONDS.
EXERCISE COMPLETED.
```

Domino

The function ⌹V allows dynamic modification of workspace and work session parameters such as output line width. Domino does not return a result, and must be the last executed (i.e. leftmost) operator in an APL expression. The arguments and actions of Domino are summarized in Table 10.

Generally, the related I-beam and Domino functions differ by 10; for example, ⌶11 gives the input device type, and ⌹1 changes devices. Expressions which combine related I-beam and Domino functions are often used in defined functions to assign a new value to a workspace parameter while saving the old value. For example, the expression ⌹6,20+0×Q←⌶16 saves the current output line width value in Q and resets it to 20. Since ⌹⍳0 is ignored (i.e. no action results), the actions of Domino may be made conditional: the expression ⌹⍳⌶11 switches the input/output device to the CRT at that station if one exists.

I-BEAM RESULT

I3	Requests light-pen input. Returns the screen row and column coordinates of the response. Returns ¯1 ¯1 if input timed-out, or if the input device was a typewriter.
I4	Returns which devices are operational at this terminal. 1 CRT 2 Typewriter 3 CRT and typewriter 5 Film projector and CRT 6 Film projector and typewriter 7 Film projector, typewriter, and CRT
I5	Current sense switch setting on the IBM 1800 console (always 0 on the IBM 1130).
I6	The current console data entry switch setting.
I7	The number of bytes available for function storage. The maximum is 5110.
I8	Port number: 0 thru 31.
I9	The vector of port numbers of active terminals.
I10	The user's sign on number (account number) returned as an integer.
I11	The user's terminal input device type: 0 CRT 1 Typewriter
I12	The current index origin: 0 or 1
I13	The current random number seed.
I14	The next CRT row upon which output will occur: 0 thru 14
I15	The current film frame number. If the film projector is not operational, then the result of I15 is meaningless. Displayable frames are in the range 1 thru 1022. 0000 Film is at reverse overrun indicator. 1023 Film is at forward overrun indicator (end).
I16	Current maximum output line-width setting.
I17	Time-out Indicator: 0 Last input did not time-out. 1 Last input timed-out.
I18	Returns current time-out setting for [] and I3 inputs.
I19	Cumulative input wait time (latency) in 60ths of a second.
I20	Current time of day in 60ths of a second from midnight.
I21	Elapsed CPU time from sign-on in 60ths of a second.
I22	The number of bytes currently available in the data workspace. The maximum is 6400. Each array (temporary or defined) uses 4×1+(number of elements) bytes.
I23	The number of users currently signed on the system.
I24	User's sign on time in 60ths of a second.
I25	Today's date as an integer in the form MMDDYY.
I26	The current line number of the function being executed.
I27	The vector of the line numbers of pendent or suspeded functions from inner to outer.
I28	The number of pendent or suspended functions.
I29	The current significant digit setting for numeric output.

Table 9: THE I-BEAM FUNCTION

DOMINO RESULT

⌹0 Erases the CRT screen and positions the cursor at the top of the screen. ⌹0 is ignored if the current input/output device is a typewriter.

⌹1 Switches input/output control to the other terminal device if it has been configured and is operational; otherwise, ⌹1 is ignored.

⌹2,*N* Sets the index origin (see)*ORIGIN* command) to *N* where *N*∈0 1.

⌹3,*N* Sets the random number seed to *N* where *N*∈0,32767.

⌹4,*N* Sets the CRT row to *N* if the CRT is the current input/output device; otherwise, ⌹4 is ignored. *N*∈0,⍳14.

⌹5,*N* Positions the film at frame *N* if the Film Projector has been configured and is operational; otherwise, ⌹5 is ignored. If *N*∈⍳1022 position film and open shutter; if *N*=0 rewind the film to the reverse overrun indicator and leave the shutter closed.

⌹6,*N* Sets the current maximum output line-width (see)*WIDTH* command) to *N* where *N*∈19+⍳101. If the CRT is the current input/output device then the width will be set to 40⌊*N*.

⌹7,*N* Stops execution for *N* 60ths of a second where *N*∈0,⍳32767.

⌹8 Stops execution until a key is pressed. If the current input device is a typewriter, then ⌹8 is ignored. The pause time associated with ⌹8 is not counted in the ⎕I9 accumulation.

⌹9,*N* sets the maximum significant digits (see)*DIGITS* command) for numerical output to *N* where *N*∈⍳16.

⌹10,*N* Sets time-out for ⍞ and light pen inputs to *N* 60ths of a second, where *N*∈0,⍳32767.

NOTES: 1. In each of the above, *N* must be a scalar or one element array.

2. DOMINO must be the last executed operation (i.e., the leftmost) of an APL expression.

Table 10: THE MONADIC DOMINO FUNCTION

THE PLOT FUNCTION

The expression *A*⍞*B* results in a plot of the data contained in the vector right argument *B*. *B* is composed of the vector of Y-axis data catentated to the vector of X-axis data.

B ↔ *X*,*Y* and (⍴*X*) ↔ ⍴*Y*

The plotted points are bounded by the vector left argument *A*. *A* must be a 4-component vector defined as:

A[1] ↔ The minimum X-data to be plotted.
A[2] ↔ The maximum X-data to be plotted.
A[3] ↔ The minimum Y-data to be plotted.
A[4] ↔ The maximum Y-data to be plotted.

Points falling outside of the specified *A* values will not be plotted. The data is plotted on a 25 by 31 grid using ∘ to mark the points plotted. Axis markings are output to 3 significant digits and the X and Y scale factors are displayed at the bottom of the plot. Plots on a CRT or typewriter are identical. After a plot on a CRT, no further execution occurs until a key has been pressed. This protects the plot from being inadvertently destroyed by subsequent output. The following is an example of a plot:

```
      X←100 200 300 400 500 600 700 800
      Y←2806 1403 935 702 561 468 401 351
      100 800 350 2810⍞X,Y

2.81┼ ∘
    |
    |
2.19┼
    |
    |
1.58┼
    |     ∘
    |
0.96┼           ∘
    |
    |                 ∘
    |                    ∘
    |                       ∘
0.35┼                          ∘   ∘
      ┼-----┼-----┼-----┼-----┼-----┼-
    1.00  2.40  3.80  5.20  6.60  8.00
  X←X×10* 02          Y←Y×10* 03
```

The scale factors multiplied by the values on each axis give the X-data and Y-data. In this example the X-data ranged from 100 to 800 and Y-data ranged from 350 to 2810 approximately.

$\underline{V}$ is a useful function for plotting data. $\underline{V}$ tests X and Y for plot argument legality, extends scalar arguments, computes the maxima and minima for the plot left argument, and plots Y $\underline{vs}$ X. $\underline{V}$ is defined as follows. Note line 6.

```
    ∇Y ⍺ V X;⍺;V
[1]   →((2=(ρρX),ρρY),(0≠(0\0ρX),0\0ρY),(1=(ρ,X),ρ,Y),(ρ,X)≠ρ,Y)/
        10 10 11 11 8 9 12
[2]   →(≠/⍺←(⌊/X),⌈/X)/4
[3]   X←⍺+ .4 .6×X+⍺=0
[4]   →(≠/V←(⌊/Y),⌈/Y)/6
[5]   Y←V+ .5 .5×Y+V=0
[6]   (⍺,V)⊞X,Y
[7]   →0
[8]   →2,X←(ρY)ρX
[9]   →2,Y←(ρX)ρY
[10]  →0,ρ⎕←'MATRIX ARGUMENTS ILLEGAL'
[11]  →0,ρ⎕←'LITERAL ARGUMENTS ILLEGAL'
[12]  'UNEQUAL LENGTH VECTORS ILLEGAL'
    ∇
```

The plot example from the previous page can be obtained by executing:

```
    X←100×⍳8
    Y←2806 1403 935 702 561 468 401 351
    Y V X
```

APL\1500 FILE SYSTEM

The *APL*\1500 File System provides the following capabilities:

1. The dynamic manipulation of larger data bases than can be contained in a workspace.

2. The retention of input or computational results under program control.

3. The interaction of a group of users via the modification of a common data base.

This, student responses can be saved for subsequent analyses without recourse to a multitude of system commands; the large volume of data associated with information retrieval systems can be conveniently manipulated; and, asynchronous hardware operations can be simulated.

Files

A file is a collection of items which are arbitrary APL arrays. Items are identified by item numbers which are positive integers less than 256. Files reside on auxiliary (disk) storage and are identified by 1 to 6 character file names. File operations are used to store, retrieve, and erase file items, determine which file items are defined, and request temporary exclusive use of a file. Up to 4608 bytes of information may be stored in a file. The creation and assignment of files is controlled by the *APL*\1500 System Operator.

A user can have up to 8 private files which can be listed by the command *)FILES* (see Part 2). A user can access any of his private files, as well as any public file; however, certain file operations may not be permitted with a public file. Private files may belong to any number of users. A file which is accessible by more than 1 user is called a shared file; all public files are shared files.

File operations

All file operations are performed by the primitive function ⌷ (upshifted *J*), which may be either monadic or dyadic according to the desired operation. File operations may, like any other function, be embedded within APL expressions. The only exceptions are the erase, hold and release operations which do not return a result, and, therefore, must be the last executed functions in an APL expression.

The following is a list of valid file operations:

1. Select the file to be used for subsequent file operations.

2. Ascertain which items are defined in a file.

3. Read an item from a file.

4. Create or Replace an item in a file.

5. Erase an item from a file.

6. Hold a file for temporary exclusive use.

7. Release a previous hold on a file.

These operations and their uses are explained in the following section and summarized in Table 11. The explanation of possible error reports is summarized in Table 12. Examples of the use of each operation may be found in Appendix A, Sample Terminal Session.

Select

Before any operation can be performed on the items in a file, the user must indicate which file is to be used. If *V* is a character vector which represents a file name, then the expression •*V* selects the file specified by *V* for use with subsequent file operations. The file name may be padded on the right with blanks providing that the 6 character limit is not exceeded. The select operation returns the number of unused bytes remaining in the named file. Only one file can be selected at a time, but files may be selected as often as required. Each file remains selected until another select operation is performed. System commands have no effect upon file selection.

If *V* is an empty character vector (*V*←'') then the operation •*V* cancels any previous file selection; a 0 is returned.

NOTE: Since the select operation is the slowest of the file operations, care should be taken to select a file only when it is necessary to do so.

Ascertain

The operation •0 is used to find out which items are defined in a file. The result is the vector of defined item numbers in ascending order.

Read

The read operation is used to bring a copy of a file item into the active workspace. The syntax is •*I* where *I* is the item number of a defined item (i.e. *I*∈•0). The result is a replica of item *I*, which may be used within more complex APL expressions. The read operation is analogous to using the name of a defined variable. For example, an attempt to read a non-existent item yields a *VALUE ERROR* report.

Create and Replace

The create operation is used to add an item to a file. The replace operation is used to modify an existing file item. The syntax of both is $I \bullet A$ where I is an item number and A is an arbitrary APL array. If item I is defined ($I \in \bullet 0$) the operation is a replace; if item I is undefined ($\sim I \in \bullet 0$) then the operation is a create. Subsequent to this operation, item I will be defined as a copy of the array A.

Create and replace are analogous to specifying the value of a variable. In particular, if either operation is the last executed function in an APL expression, no value is returned or displayed. If, however, a create or replace operation is embedded within a more complex APL expression, then the right argument is returned as the result. Thus, the statement `1 2•A` would specify both item 1 and item 2 as A, but would not display any result.

Erase

If I is an item number, then the expression $\bullet - I$ will erase item I from the selected file. If item I was not defined before the operation, no special response is made. Since the erase operation does not return a result, it must be the left-most (last executed) function in an APL statement.

Hold and Release

Applications using shared files may occasionally require temporary exclusive access to a file. This facility is provided by the hold and release operations. The syntax of both is $0 \bullet H$ where $H \leftarrow 1$ for hold and $H \leftarrow 0$ for release. Since neither operation returns a result, hold and release must be the last executed functions in an APL statement.

The file hold operation posts a request to have temporary exclusive use of the currently selected file. If this file is already held by another user, the execution of this user's program will be suspended until the file is released. A suspended hold operation may be interrupted by an attention signal, in which case the hold request is revoked. The file hold and release operations do not affect the execution of any other file operations.

For example, the following function appends the user's account number, the time, the date, and a input message as two items to the selected file.

```
    ∇M;A;I
[1]  A←⍞                                  Request input
[2]  0•1                                  Place a hold on the file
[3]  (I←1+¯1↑0,•0)•(⍕10),(⍕20),⍕25         Append number,time,date
[4]  (I+1)•A                              Append input message
[5]  0•0                                  Release the file
    ∇
```

Note that if more than one user were to execute the above function simultaneously, only one user at a time would be permitted to execute line 3 and line 4. The remaining users would be temporarily halted at line 2 until line 5 is executed.

TABLE 11

SUMMARY OF FILE OPERATIONS

OPERATION	SYNTAX	RESTRICTIONS	ACTION TAKEN	RESULTS RETURNED
SELECT	∘*F*	*F* must be a character vector which is the name of a file that this user may access.	The specified file is selected for use with all subsequent file operations.	Returns the number of unused bytes in the selected file.
	∘''	None.	Cancels any previous file selection.	Returns 0.
ASCERTAIN	∘0	A file for which this operation is permitted must be selected.	None.	Returns the vector of defined item numbers.
READ	∘*I*	A file for which this operation is permitted must be selected. *I* must be the number of a defined item (*I*∈∘0).	None.	Returns a replica of the specified item.
CREATE/ REPLACE	*I*∘*A*	A file for which this operation is permitted must be selected. *I* must be a valid item number.	A replica of *A* will appear in the selected file as item *I*.	Returns a replica of *A* if this is not the last-executed operation in the line.
ERASE	∘-*I*	A file for which this operation is permitted must be selected. *I* must be a valid item number.	Item *I* will not appear in the selected file.	None.
HOLD/ RELEASE	0∘*H*	A file for which this operation is permitted must be selected. *H*=1 for hold; *H*=0 for release.	*H*=1: A request for temporary exclusive use of this file will be posted. *H*=0: Any hold on this file will be canceled.	None.

TABLE 12

INTERPRETATION OF ERROR REPORTS
FOR FILE OPERATIONS

FILE OPERATION	DOMAIN	LENGTH	PACK	RANK	SYNTAX	VALUE	WS FULL
SELECT	2	5	6	7			11
ASCERTAIN	1		6	8			11
READ	1,3		6	8		1Ø	11
CREATE/REPLACE	1,3,4		6	8			12
ERASE	1,3		6	8	9		
HOLD/RELEASE	1,4		6	8	9		

DOMAIN ERROR

1. A file for which this operation is permitted has not been selected.
2. Invalid file name, or the required file does not exist.
3. The item number is not a positive integer ≤255.
4. The item number or operation indicator is not numeric.

LENGTH ERROR

5. The file name must be ≤6 characters.

PACK ERROR

6. The disk pack containing the selected file is not mounted and ready.

RANK ERROR

7. The file name is not a scalar or vector.
8. The item number or operation indicator is not a scalar.

SYNTAX ERROR

9. An erase or hold/release operation is not the last executed function in a statement.

VALUE ERROR

10. The required item is not defined in the selected file.

WS FULL ERROR

11. The active WS can not contain the result of this operation.
12. The selected file can not contain the given array.

```
  X←3×4                          Assigns value of expression to X

  X                              Value of X typed out
12

  3×4                            Entry automatically indented
12                               Response not indented

  Y←¯5                           Negative sign for constants

  Y-X
¯17

  144E¯2                         Exponential form of constant
1.44

  P←1 2 3 4                      Four-element vector
  P×P                            Function applied element by element
1  4  9  16

  P×Y                            Scalar applies to all elements
¯5  ¯10  ¯15  ¯20

  Q←'CATS'                       Character constant (4-element vector)

  Q
CATS

  X←3
  Y←4
  (X×Y)+4
16
  X×Y+4                          Execution from right to left
24

  X¯Y                            Entry of invalid expression
SYNTAX ERROR                     Shows type of error committed
  X¯Y                            Retypes invalid expression with
   ^                             caret where execution stopped

  1|2.6                          Residue function
0.6

  3≤7                            Less than or equal function
1

  7≤3                            Greater than or equal function
0
```

```
   1 2 3 4×4 3 2 1
4  6  6  4
```

Multiplication function

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

Addition function

```
   1 2 3 4⌈2
2  2  3  4
```

Maximum function

```
   ⍳4
1  2  3  4
```

Index generator

```
   ⍳5
1  2  3  4  5
```

```
   ⍳0
```

Empty vector prints as a blank line

```
   6-⍳6
5  4  3  2  1  0
```

```
   2×⍳0
```

Scalar applies to all (i.e. 0) elements of ⍳0, resulting in an empty vector

```
   2×⍳6
2  4  6  8  10  12
```

```
   ∇S
[1]   S←4×3.14159×R×R
[2]   V←S×R*3
[3]   ∇
```

Function header
Function body

Close of definition

```
   R←2
   S
```

Execution of function

```
   S
50.2654
```

Display of values calculated in function

```
   V
33.5103
```

```
    ∇Q;I
[1]   S←0
[2]   I←0
[3]   S←S+I
[4]   I←I+1
[5]   →3×⍳I≤N
[6]   ∇
```

Local variables established in header

Branch to line 3 (as long as condition $I \leq N$ is met)

```
    N←5
    Q
    S
15
    I
VALUE ERROR
    ^
```

Execution of function

Local variable has no value after function is executed

```
    )ERASE S
    ∇Z←S X
[1]   Z←4×3.14159×X×X
[2]   ∇
```

Erases definition
Function header--explicit result, one argument

```
    S 3
113.097
```

```
    Q←3×S 1
    Q
37.6991
    R←2
    (S R)×R+3
33.5103
```

Use of defined function in expression

```
    ∇Z←F N;I
[1]   Z←1
[2]   I←0
[3]   I←I+1
[4]   →0×⍳I>N
[5]   Z←Z×I
[6]    →3
[7]   ∇
```

F is the factorial function

```
    F 3
6

    F 5
120
```

```
   T∆E←3 5                    Sets trace on lines 3 and 5
   X←E 3                      Execution of function
E[3]  1                       Trace of function
E[5]  1
E[3]  2
E[5]  2
E[3]  3
E[5]  6
E[3]  4


   T∆E←0                      Terminates trace


   ∇G←M G N                   Explicit function with two arguments
[1]   G←N
[2]   M←M|N
[3]   →4×M≠0
[4]   [1]G←M                  Change line 1
[2]   [4]N←G                  Override line 2 with line 4
[5]   [1□]                    Display line 1
[1]    G←N                    Old line 1 retained until close
[1]   ∇                       of definition


   ∇G[□]                      Display function definition and
   ∇G←M G N                   stay in definition mode
[1]    G←M
[2]    M←M|N
[3]    →4×M≠0
[4]    N←G
     ∇
[5] →1                        Add line 5
[6] ∇                         Close definition


   36 G 44                    Execution of function
4


   ∇G                         Add line between lines 4 and 5
[6]   [4.1]M,N
[4.2] ∇


   36 G 44
8  36
4  8
4
```

```
    ∇G[□]∇                          Display of function
    ∇G←M G N
[1]   G←M
[2]   M←M|N
[3]   →4×M≠0
[4]   N←G
[5]   M,N
[6]   →1
    ∇

    ∇G
[7]  [5]                            Deletes line 5
[6]  ∇                              Close definition

    ∇Z←B N                          An (erroneous) function for
[1]  Z←(Z,0)+0,Z                    binomial coefficients
[2]  →1×N≥ρZ∇

    B 3
VALUE ERROR                         Suspended function
B[1]  Z←(Z,0)+0,Z
                ^

    Z←1                             Assign value to Z
    →1                              Resume execution
1  3  3  1                         Binomial coefficients of order 3

  B 4
VALUE ERROR                         Same error (local variable Z
B[1]  Z←(Z,0)+0,Z                   does not retain its value)
                ^

  ∇B[.1]Z←1∇                        Cannot edit function in definition
SUSPENSION                          mode
  )SI                               Display state indicator
B[1]  *

  )PURGE                            Clear state indicator

  )SI

  ∇B[.1]Z←1∇                        Insert line
  ∇B[□]∇                            Display revised text
   ∇Z←B N
[1]   Z←1
[2]   Z←(Z,0)+0,Z
[3]   →1×N≥ρZ                      Branching error because of
    ∇                               insertion
```

```
  ∇B[3]→2×N≥ρZ∇
```

Change line 3

```
  ∇B[□]∇
  ∇Z←B N
[1]    Z←1
[2]    Z←(Z,0)+0,Z
[3]    →2×N≥ρZ
       ∇

  B 4
1  4  6  4  1
```

```
  ∇D;A;I;Y
DEFN ERROR
```

A function D is already defined

```
  ∇I;A;I;Y
[1]  'ENTER CAPITAL AMOUNT IN DOLLARS'
[2]  A←□
[3]  'ENTER INTEREST IN PERCENT'
[4]  I←□
[5]  'ENTER PERIOD IN YEARS'
[6]  Y←□
[7]  'RESULT IS ';A×(1+.01×I)*Y∇
```

A conversational function to compute value of an amount *A* invested at interest *B* for a period of *Y* years.

Request for input
Heterogeneous output

```
  I
ENTER CAPITAL AMOUNT IN DOLLARS
□:
  1000
ENTER INTEREST IN PERCENT
□:
  4.75
ENTER PERIOD IN YEARS
□:
  10
RESULT IS 1590.52
```

Waits for input from keyboard

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

Catenation

```
  A←3 4ρ2×ι12
  A
 2   4   6   8
10  12  14  16
18  20  22  24
```

Reshape

```
      6 2ρA                                  Reshape of matrix A

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


      ,A                                     Ravel of A
 2  4  6  8  10  12  14  16  18  20  22  24


      3 3ρ1 0 0 0                            Identity matrix

     1 0 0
     0 1 0
     0 0 1


      A←'ABCDEFG'
      M←4 3ρ3 1 4 2 1 4 4 1 2 4 1 4
      M

      3  1  4
      2  1  4
      4  1  2
      4  1  4

      A[M]                                   Indexing

CAD
BAD
DAB
DAD


      M[1 3 4;1 3]

      3  4
      4  2
      4  4


      X←2 3 5 7 11
      X[1 3]←6 8                             Indexed variable on left of
      X                                      specification arrow
6  3  8  7  11
```

```
M
1   2   3
4   5   6

  +/[1]M
5  7  9                    Column reduction

  +/[2]M                   Row reduction
6  15


  +/M                      Sum reduction
6  15


  +\[1]M                   Column scan

  1  2  3
  5  7  9

  +\[2]M                   Row scan

  1  3  6
  4  9 15


  A←2 3⍴1 5 7 3 4 2
  A

  1  5  7
  3  4  2

  A+.×100 10 1             Inner Product (+.× is ordinary
157  342                   matrix product)


  X←⍳3
  Y←⍳4

  X∘.×Y                    Outer Product

   1   2   3   4
   2   4   6   8
   3   6   9  12


⎕←M←3 4⍴⍳12

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

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


0 1 2 3⌽[1]M                          Column rotation

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

1 2 3⌽[2]M                            Row rotation

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


⌽[1]M                                 Column reversal

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


⌽[2]M                                 Row reversal

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


1 0 1/[1]M                            Column compression

 1   2   3   4
 9  10  11  12


1 1 0 1/[2]M                          Row compression

 1   2   4
 5   6   8
 9  10  12


 1 0 1 1 0\⍳0                         Mesh
0  0  0  0  0

 1 1 1 0 1 1 0 1 1 1 1\'442308446-'
442-30-8446

 1 0 1 1 0\1 2 3 4 5 6 7
1  4  2  3  5
```

```
    1 0 1 1 0 1\⍳4
1   0   2   3   0   4


  M←3 5⍴'CAD()BAT()END()'
  M

CAD()
BAT()
END()

  0 1 1 1 0/M

  (CAD)
  (BAT)
  (END)


  5∘.⍳5                                  Prefix

  1  0  0  0  0
  1  1  0  0  0
  1  1  1  0  0
  1  1  1  1  0
  1  1  1  1  1


  M←2 3⍴⍳6
  M

  1  2  3
  4  5  6


  10⊥M                                   Decode
123  456

  10 10 10⊤123 456 ¯1                    Encode

  1  2  3
  4  5  6
  9  9  9


  'ABCDEFGJ'⍳'COFFEE'                    Index of
3  9  6  6  5  5

  A←3 4 ⍴⍳12
```

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

    2 ¯2↑A                          Take

    3   4
    7   8


    2 ¯2↓A                          Drop

 9   10

    ⍋A←45 23 78 ¯45 71 55           Upgrade
  4  2  1  6  5  3


    A[⍒A]                           Descending Sort
¯45  23  45  55  71  78


    5?5 6 7                         Deal

    5  1  2  4  3
    6  2  5  1  4
    4  2  5  7  6


   )FILES                           List files enabled for your workspace
FILE1

    ⎕NTIE 0 'FILE1'                 Open or select a file number
4608                                Number of bytes left in file

    ⎕NINFO 0                        Ascertain list of defined items
                                    None defined

   10⎕NPUT 2 5⍴⍳10                  Create item 10 with 2 by 5 matrix
                                    of the integers 1-10

   ⎕NINFO 0                         Ascertain list of defined items
10

   ⎕NGET 10                         Read item 10 from the file
 1  2  3  4  5
 6  7  8  9 10


   10⎕NREPLACE 3 3⍴9                Replace item 10 with new matrix
```

```
    ⍞10                                  Read item 10
 1  2  3
 4  5  6
 7  8  9

3⍞'THIS FILE ITEM IS BEING CREATED'      Create item 3 with character vector

3⍞'ITEM 3 IS NOW REPLACED'               Replace item 3

   ⍞0                                    Ascertain
3  10

   ⍞¯10                                 Erase item 10

   ⍞10                                   Read item 10
VALUE ERROR                              No value since it was erased.
   ⍞10
   ∧

   ⍞0                                    Ascertain
 3

   2⍞2  4  6  8                          Create item 2

   A←⍞2                                  Specify A as item 2

   A
2  4  6  8


      ∇R F                               Function to display all items
[1] →2××⍴F←,⍞0⍞F                         in right argument
[2]  ⍞1↑F
[3] →2××⍴F←1↓F ∇


   R 'FILE1'                             Execute function R

2  4  6  8
ITEM 3 IS NOW REPLACED

⍝THIS ENDS THE EXAMPLES IN THE TERMINAL SESSION          Comment
```

BIBLIOGRAPHY

Berry, P.C., APL\360 Primer, IBM Corporation, 1968.

Berry, P.C., APL\1130 Primer, IBM Corporation, 1968.

Breed, L.M., and R.H. Lathwell, "The Implementation of APL\360", ACM Symposium on Experimental Systems for Applied Mathematics, Academic Press, 1968.

Falkoff, A.D., and K.E. Iverson,"The APL\360 Terminal System", ACM Symposium on Experimental Systems for Applied Mathematics, Academic Press, 1968.

Falkoff, A.D., K.E. Iverson, and E.H. Sussenguth, "A Formal Description of System/360", IBM Systems Journal, Volume 3, Number 3, 1964.

Iverson, K.E., A Programming Language, Wiley, 1962.

Iverson, K.E., Elementary Functions: an algorithmic treatment, Science Research Associates, 1966.

Iverson, K.E., "The Role of Computers in Teaching", Queen's Papers in Pure and Applied Mathematics, Volume 13, Queen's University, Kingston, Canada, 1968.

Lathwell, R.H., APL\360: Operator's Manual, IBM Corporation, 1968.

Lathwell, R.H., APL\360: System Generation and Library Maintenance, IBM Corporation, 1968.

Pakin, S., APL\360 Reference Manual, Science Research Associates, 1968.

Rose, A.J., Videotaped APL Course, IBM Corporation, 1968.

Smillie, K.W., Statpack 1: An APL Statistical Package, Publication No. 9, Department of Computing Science, University of Alberta, Edmonton, Canada, 1968.