A PROGRAMMING LANGUAGE/1130

REVISED: 5/5/69 1130-03.3.001

A PROGRAMMING LANGUAGE/1130
REVISED: 5/5/69 1130-03.3.001

CONTRIBUTED PROGRAM LIBRARY

DISCLAIMER

This program and its documentation have been contributed to the Program Information Department by an IBM employee and are provided by the IBM Corporation as part of its service to customers. The program and its documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of this program and the user of the program is expected to make the final evaluation as to the usefulness of the program in his own environment. There is no committed maintenance for the program.

Questions concerning the use of the program should be directed to the author or other designated party. Any changes to the program will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

Program Contribution Form

Type III (IBM Employee)

IBM Corporation
Program Information Department (PID)
40 Saw Mill River Road
Hawthorne, New York 10532, U.S.A.
Attention: Program Control Desk

1 PROGRAM ORDER NUMBER (TO BE FILLED IN BY PID): 1130 03.3.001

2 SYSTEM TYPE (MACHINE): 1130

3 SEARCH KEY: MATHEMATICAL AND BUSI- NESS CONVERSATIONAL INTER- ACTIVE ARRAY - PROCESSING PROGRAMMING LANGUAGE (APL)

4 AUTHORS' NAME(S) (IF DIFFERENT THEN SUBMITTER'S):
R.S. Carberry, A. G. Nemeth
L.M. Breed, C. H. Brenner
S.M. Raucher

5 SUBMITTER'S NAME (DIRECT TECHNICAL INQUIRIES TO): S.M. Raucher

6 SUBMITTER'S ADDRESS:
IBM
11141 Georgia Avenue
Wheaton, Maryland 20902

7 TITLE OF PROGRAM: A Programming Language/1130

8 PRIM.SUB. CODE: 0 3.3

9 SECONDARY SUBJECT CODES: 1 3.0 | 1 7.0 | 4 1.0 | 4 5.0

10 OPERATING OR MONITOR SYSTEM REQUIRED: NONE

11 TYPE OF SUBMITTAL PLEASE CHECK (✓):
1 INITIAL PROGRAM ABSTRACT
2 INITIAL SUBMISSION OF PROGRAM
3 RESUBMITTAL OF UNANNOUNCED PROG.
4 CORRECTION TO PROGRAM NUMBER: x 1130.03.3.001

12 DATE OF SUBMITTAL:

13 ABSTRACT (PLEASE LIMIT TO 150 WORDS. DESCRIBE PROGRAM AND PURPOSE. CLEARLY IDENTIFY MACHINE CONFIGURATION AND SOURCE LANGUAGE.)

APL is a conversational implementation of the Iverson notation, an extremely concise mathematical notation with simple but rigorous syntax. This concise attribute virtually eliminates the -program- step in the problem-solving chain of problem--algorithm--program--solution. All operators of the notation, editing capabilities, and the capability to save and retrieve work spaces are provided. The implementation allows data to be structured as scalars, vectors, and matrices with up to 255 elements in any dimension. Numerical values are accurate to six decimal digits, and identifiers are up to 6 alphabetic characters. Input may come from the console typewriter, card reader or a typewriter terminal. The program is independent of the IBM monitor and requires a dedicated disk cartridge. Facilities are provided to generate the system, assign and delete workspaces, and dump/restore individual work-spaces and their functions to cards.

PLEASE ATTACH ADDITIONAL PAGES IF NECESSARY → TOTAL PAGES ATTACHED: 1

120-1424-3

PERMISSION TO PUBLISH: "I HEREBY GIVE IBM PERMISSION TO RE-PRINT, REPRODUCE AND DISTRIBUTE THIS PROGRAM TO ANYONE."

14 Stephen M. Raucher 4/11/69
SIGNATURE OF SUBMITTER AND DATE

Minimum configuration is 1131-2B and 1442 or 2501. A 2741 terminal and requisite RPQ are highly desirable. APL/1130 is written in 1130 Assembler Language.

TABLE OF CONTENTS

DECK KEY

For 1442 loading

Load card:	APLIPL
Privileged user load card:	APLIPLPR
IPL sector deck:	First card unsequenced Next 8 cards 1442LD00-1442LD07 Next 5 cards IPL00000-IPL00004
APL system deck:	First card unsequenced Next 8 cards 1442LD00-1442LD07 Next 438 cards APL00000-APL00437
Empty Directories deck:	First card unsequenced Next 8 cards 1442LD00-1442LD07 Next 9 cards DIR00000-DIR00008

For 2501 loading

Identical to the decks above, except that the first 9 cards of each deck are to be removed and replaced by:

First card unsequenced
Next 14 cards 2501LD00-2501LD13

(Three copies of this deck are included for user insertion- see OPERATING INSTRUCTIONS page 13.2)

TOTAL NUMBER OF CARDS 526

TAPE KEY (OPTIONAL MATERIAL)

1. The tape supplied as,optional material, contains card images, blocked 20 cards per block. The tape is unlabeled. There is 1 file on the tape, preceeded by a single tape mark. Two tape marks follow the end of the file.*
2. Data stored includes a complete 1130 job stream (including all required control cards necessary to assemble the system under Monitor II).
3. The tape was created by the System 360 DEBE program. It may be dumped back to cards by the same program or any appropriate tape to card utility program.

* OPTIONAL MATERIAL WILL BE FORWARDED ONLY WHEN SPECIFICALLY REQUESTED.

USER INFORMATION KEY

The following items of interest may be found where indicated:

Purpose: See Preface

Advantages: See Preface

Restrictions and range: See page 56

Precision: See page 56

Program requirements: See Operating Instructions

System configuration: See page 20

Timing: Response to trivial operations is within 2 seconds. Other responses are proportional to the amount of computation required.

Program modification aids: None provided.

Input and output description: See page 24 and page 45

Sample problem: See Appendix A, page 93

PREFACE

APL\1130 is a single-user implementation on a smaller machine of the APL\360 conversational terminal system that has been operating within IBM since the Fall of 1966. Conceived as an experimental system for exploring certain aspects of computer science, its purpose required that it operate under a realistic load in an environment that was not artificially constrained. To this end, the members of the IBM Research staff, and others, were encouraged to learn the APL language and use the APL system. It was not anticipated that it would have the impact that it did:

> In twenty months of regularly scheduled operation, clocking well over 100,000 terminal-hours of use, the availability of APL\360 has materially changed the computing habits of the Research organization.
>
> Heavy users of batch processing have turned to APL for on-line development of their algorithms.
>
> Many laboratory data reduction chores formerly done by batch operation or desk calculator are now executed in a timely way at local terminals, using locally written, stored APL programs.
>
> Automatic collection of experimental data has, in many instances, been put on-line to APL.
>
> Routine correspondence and technical papers are prepared at terminals, with the help of text-handling programs written in APL.
>
> Many professional staff members who formerly were indifferent to, or actively resisted, the use of computers, have become steady users of the APL system.

In addition to IBM Research personnel, use of the system was offered to other locations within the IBM Company, and it was also used experimentally in elementary and secondary schools and in universities. The general findings may be summarized as follows:

> Although APL is easy for a beginner to learn, non-programmers (as well as programmers) often develop an interest in sophisticated use of the language, because of its analytical power and mathematical structure.

> The primitive array operations of APL make it a good language for scientific problems, text handling, and general data processing, because arrays are fundamental to all of these applications.
>
> The use of a powerful, readily accessible computational facility can materially change the quality and orientation of an academic course.
>
> Other things being equal, acceptance of conversational computing as a general mode of operation is strongly dependent upon its reliability and availability -- regularly scheduled hours are essential, and the more the better, including nights and weekends.

APL terminal systems are characterized by the following:

> Simple, uniform rules of syntax
>
> Use of common symbols for the 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
>
> Fast response
>
> A library structure built around workspaces that hold both programs and data
>
> An immediate-execution mode completely free of irrelevant keywords
>
> A comprehensive, integrated set of system commands for managing workspaces and libraries, and for other essential functions
>
> Three levels of security; account numbers, workspaces, and programs can be individually locked against use or display
>
> Visual fidelity between hard copy and transmitted entries, which ensures reproducibility of results
>
> Succinct diagnostic reports

New release. This is the second release of APL\1130. It embodies a number of new features which are described in detail in the APL\1130 User's Manual. They include:

1. Six-character identifiers.

2. Labels.

3. Extended function editing.

4. Card control commands for reading and punching cards.

5. Provision for connecting a remote 2741 terminal.

6. A dynamic method of localizing names as described in the section on Homonyms.

7. A new command)*SIV* to display the state indicator together with the list of local variables for each halted function.

8. A system of locks and keys for security of account numbers and workspaces.

9. The signum function (denoted by ×) has been added.

10. The following changes in notation have been made:

New notation	Replaces
)*ERASE F*	∇*F*∇
)*SI*	)Δ
*T*Δ*F*	Δ*F*

11. A set of COPY commands permits the copying of all or part of one workspace into another.

Instructional use. APL\1130 is well suited for use by secondary schools for instructional purposes. In immediate execution mode, APL\1130 executes each line as the student types it and replies before he types the following line. This give-and-take between student and computer enhances student involvement and augments the educational process. Function definition mode enables a student to build and store functions for later use which become a framework for the subject matter.

APL\1130 is suitable not only for student problem-solving in science and mathematics courses, but also for drill and exercise in a programmed-teaching mode. The system provides a formal method of function definition and a method of saving data and programs from session to session. APL also facilitates individual student experimentation; the student may draw from his own experience and library of functions or from a library of commonly used programs maintained by an instructor.

Under normal circumstances, it is difficult for a teacher to be able to provide adequate instruction for all the levels of student achievement represented in one classroom. APL\1130 can help accommodate the individual differences present in several ways. First, the computer system can provide drill and exercise for those students who require more practice in problems than the instructor can provide. Secondly, APL\1130 provides unlimited opportunity for individual experimentation and exploration for the above-average student; a gifted child no longer need be inhibited by the requirements of his classmates. Finally, APL\1130 can be used by teachers to prepare numerical examples which would otherwise require tedious time-consuming hand calculations.

In addition, the following features of APL\1130 are of particular significance in instructional use:

1. Since APL is simple and concise and is closely related to the ordinary notation of algebra, a student's attention can be devoted almost entirely to solving the problem, rather than being diverted by irrelevant details of a computer.

2. Basic APL operators are single characters. Economy of symbols is gained by using each symbol as both a monadic and dyadic operator, just as a minus sign is used in ordinary arithmetic.

3. APL provides standard operators to perform common functions which in other languages must be programmed. For example, polynomial evaluation, random number generation, summation, matrix multiplication and permutation may each be done using a few APL characters.

4. APL handles collections of data as easily as individual items. For example, the volumes of five different cylinders may be calculated simultaneously by the same function used to calculate the volume of one.

5. Numbers used in APL are normally in familiar decimal form. Powers of 10 can be written using exponential or scientific notation, i.e., 1400000 can be written 1.4E6 or .14E7. Therefore, representation of very large or very small numbers is facilitated.

However, operators are provided to facilitate use of number systems other than base 10. This facility conforms to the modern mathematics curriculum used in many schools.

6. APL avoids the use of complex function hierarchy rules. The order of execution of an expression is determined first by the placement of parentheses and otherwise by one simple rule -- each operator works on everything to its right. For example, in the expression 17 ÷ 2 + 3, the ÷ has as its denominator 2 + 3. The entire expression may be read as 17 divided by the sum 2 + 3.

7. Correction of errors discovered before a statement is entered into the computer is simple. The system provides "visual fidelity" acting upon exactly what appears on the printed page, regardless of the order of entry. The printed record therefore makes clear exactly what was acted upon by the computer. This is important for later review and explanation by a teacher.

8. Error messages are concise and indicate to the student not only the type of error he has commited, but also where the error was made. Resumption of work from the point of error is simple.

9. APL\1130 provides a simple means for function definition. Special purpose operations needed for a particular problem may be easily defined by the student or by the teacher for use by the entire class.

10. APL\1130 provides an extensive library capability; each user can maintain a personal library where he may store and later retrieve programs, and also has the opportunity to use functions defined in other libraries.

11. APL\1130 incorporates card control commands which provide for punching APL functions and data on cards and for reading APL commands and statements from cards so punched, or from cards prepared on a separate keypunch.

OPERATING INSTRUCTIONS

This manual contains information required by those who manage and maintain an APL\1130 system. It is not needed by those who simply intend to <u>use</u> APL\1130. Part 1 of the User's Manual should also be consulted.

GENERATING THE SYSTEM

APL\1130 is written in 1130 Assembler Language and runs on a dedicated disk. It is independent of any other 1130 programming systems. To generate an APL\1130 system on a disk pack:

1. Initialize the disk, using either the Disk Pack Initialization Routine (DPIR) distributed with Version 1 of the Disk Monitor System or the Disk Cartridge Initialization Program (DCIP) distributed with Version 2.

2. Select the appropriate card decks for loading from the 1442 or 2501.

3. Place the IPL Sector deck in the card reader. Press START on the card reader and IMMEDIATE STOP, RESET and PROGRAM LOAD at the console. After the last card has left the hopper, press START on the card reader and PROGRAM START at the console so that the last card will be processed. Do not follow the deck with blank cards. Perform the last card procedure <u>exactly</u> as described.

4. Repeat step 3 with the APL System deck.

5. Repeat step 3 with the Empty Directories deck.

The APL system should now be ready for the first Initial Program Load, as described in the next section.

The APL System deck can be reloaded by itself if the system is destroyed or when a new release of APL\1130 is made. To do so, just repeat step 4. User enrollments and saved workspaces will be intact.

INITIAL PROGRAM LOAD

When the 1130 system is first turned on or when an APL disk has just been placed in the disk drive, an Initial Program Load procedure is needed to make the APL system operative. Two Initial Program Load cards are provided for this, one labeled APLIPL (for general use) and one labeled APLIPLPR (for "privileged" use - this card allows the use of the system maintenance commands by the first person who signs on).

The Initial Program Load Procedure is given in Part 1 of the User's Manual.

When an APL disk has been freshly generated, the first Initial Program Load should be performed with the privileged load card and the operator should sign on using the number 0. This allows him to assign the first users to the system. The number 0 need not be used after the first sign-on; additional users can be assigned at any time by a privileged user.

USE OF CONSOLE SWITCHES

Console switches 0 and 1 select the input device from which the next sign-on will be accepted. They may be set at any time and take effect at the next sign-on attempt, as follows:

Switch 0 down, switch 1 down: The 1131 is the APL terminal device. Typed input will be accepted from the console keyboard.

Switch 0 down, switch 1 up: The 1131 is the APL terminal device. Card input will be accepted from the card reader, as if a *)CARD* command had been obeyed previously.

Switch 0 up: The 2741 is the APL terminal device. Typed input will be accepted from the 2741. Switch 1 is ignored, and card input will be accepted only upon execution of a *)CARD* command.

SYSTEM MAINTENANCE COMMANDS

The system maintenance commands are intended for the use of those responsible for maintaining the APL\1130 system; they are <u>not</u> for general use. For that reason, they are available only to a privileged user, i.e., the first user to sign on after a "privileged" IPL. They allow for the assigning of workspaces to users, the removal of users who will no longer use the system, and the printing of information about the current state of the APL system.

)ASSIGN N USERNAME [LOCK]

Assign user N (N is between 1 and 65535) another workspace (which may be his first) and call him *USERNAME*. The lock (a colon followed by a password; see discussion of locks and keys in the User's Manual) is optional. For example, *)ASSIGN 3141 JSMITH* or *)ASSIGN 3141 JSMITH:LC1*

<u>Trouble reports</u>:
SYSTEM FULL - All disk space has been assigned.
INCORRECT COMMAND

)EXPUNGE N

Remove user number N from the system, along with all of his workspaces, freeing the space to be assigned to other users.

<u>Trouble reports</u>:
NUMBER NOT IN SYSTEM - An attempt has been made to expunge a non-existent account number.
INCORRECT COMMAND

)PEOPLE

List all the users in the system, giving for each user his username, his account number, and the number of times he has signed on.

<u>Trouble reports</u>:
INCORRECT COMMAND

)SPACES

List all the workspaces in the system, classified under the users to whom they belong.

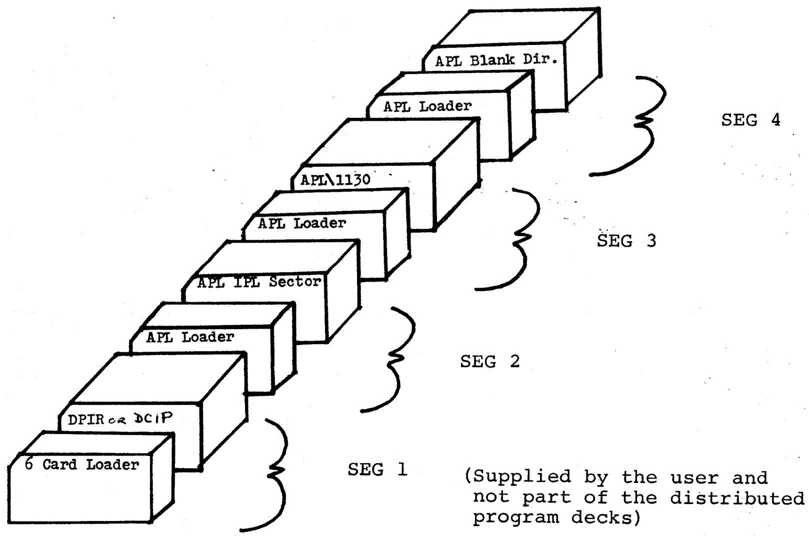
Trouble reports:
INCORRECT COMMAND

)DROP WSID

This command has the same effect as the non-privileged *)DROP* command described in the User's Manual, except that the *WSID* may contain any account number, and that using it during a privileged sign-on decreases by one the number of workspaces allotted to the account number (unless it has only one workspace).

Sample Typewriter Output

)0	
SIGNED ON	Only when new system generated
APL\1130	is user number 0 used
)ASSIGN 100 TEACHER	User no. 100 named Teacher assigned
)ASSIGN 100 TEACHER	to system with two workspaces.
)ASSIGN 1 STUDENTA	Users nos. 1 and 2 named Student
)ASSIGN 2 STUDENTB	A & B assigned to system with one
)OFF	workspace each.
SIGNED OFF	After every sign off system in
	student mode.
)100	User no. 100 named Teacher signs on.
TEACHER SIGNED ON	
APL\1130	Performs some work
)SAVE WORK	Saves workspace
WORK SAVED	
)OFF	
SIGNED OFF	User named Teacher signs off.
)3	User no. 3 tries to sign on, but
NUMBER NOT IN SYSTEM	has not been assigned to system.
)1	User no. 1 signs on
STUDENTA SIGNED ON	
APL\1130	Performs work
)SAVE SOMEWHERE	Saves workspace Somewhere.
SOMEWHERE SAVED	
)CLEAR	Loads clean workspace
	Performs more work
)SAVE THIS	Tries to save This, but has already
WS RATION EXCEEDED	used the one space he was assigned.
)LIB	Lists his workspaces.
SOMEWHERE	
)DROP SOMEWHERE	Drops previously saved Somewhere.
SOMEWHERE DROPPED	
)LIB	
	Lists again - there are no more
)SAVE THIS	saved workspaces. Now can save This.
THIS SAVED	
)LIB	
THIS	Lists saved workspaces. This is saved.
)OFF	Signs off. System now waiting
SIGNED OFF	for another student to sign on.



APL SYSTEM LOAD

Notes:

1. SEG 1-SEG 4 are loaded separately (see operating Instructions).

2. SEG 1 is only required to clear the disk. It may be omitted in rebuilding a pack.

3. SEG 4 (APL BLANK DIRECTORIES) may be omitted in rebuilding a pack where:

   a. former workspaces and/or functions are to be preserved.

   b. former directories are known to be unchanged.

APL\1130: User's Manual

A. D. Falkoff

K. E. Iverson

ACKNOWLEDGEMENTS

The present APL\1130 Manual is based on the APL\360 Manual, just as the APL\1130 system is based on the APL\360 system. For assistance in preparing the necessary modifications, the authors are indebted to Messrs. S. M. Raucher[*], D. J. Hills[†], C. A. Weidman[†], D. Oldacre[#], and L. M. Breed.

The original APL\1130 system was implemented by Messrs. R. S. Carberry[**], P. S. Abrams[††], L. M. Breed, C. H. Brenner, and A. G. Nemeth. Specifications for the present revision were drawn up by Messrs. Raucher, Breed, and Oldacre; the revisions were implemented by Mr. D. Oldacre and Mr. E. B. Iverson[#], with the assistance of Messrs. R. Kerley[##] and L. M. Breed.

A special acknowledgement is due to John L. Lawrence, who provided important support and encouragement during the early development of APL implementation, and who pioneered the application of APL in computer-related instruction.

* GEM Region, IBM Corporation, Washington, D. C.
† Amherst College, Amherst, Massachusetts
\# I. P. Sharp Associates, Toronto, Canada
** Hartford GEM Office, IBM Corporation
†† Stanford University, Stanford, California
\## IBM Corporation, White Plains, New York

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

PART 1

GAINING ACCESS

The APL\1130 system employs the APL language described in Part 3 and the system commands described in Part 2. It is operated either from the 1131 console or from an auxiliary 2741 terminal. This part of the manual describes the procedures for beginning and ending a work session.

The following physical equipment is required for the operation of APL\1130:

An 1131 CPU with 8K or more of core storage and a built-in disk drive.

A 2501 card reader or 1442 card read/punch wired for Initial Program Load.

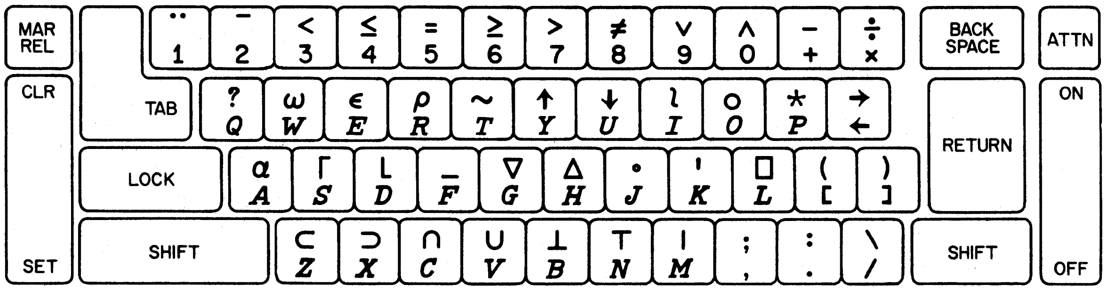
An APL typing element #1167988.

Recommended, but not required, is a 2741 terminal and an IBM 1130 Communications Channel Adapter (RPQ W16427) with data-phone or IBM 4-wire modem, and 2741 interrupt feature. With the 2501 reader a 1442 punch is required for the)*PCH* and)*PCHS* commands but is not otherwise needed.

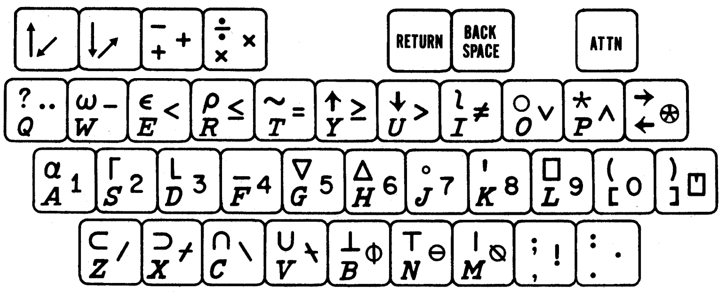
THE APL CHARACTER SET

The APL character set is shown in two arrangements in Figure 1.1; the first is appropriate to the 2741 terminal, and the second to the 1131 console. On the 2741 terminal, the numerals, alphabetic characters, and punctuation marks appear in their usual places, although the alphabet is used in only a single case: letters print as upper-case italics, but are produced only when the keyboard is in lower-case position (i.e., not shifted).

The special characters, most of which are produced with the keyboard shifted, generally have some mnemonic connection with their alphabetic or numeric correspondents.



2741 APL KEYBOARD



1131 APL CONSOLE KEYBOARD

APL KEYBOARD STICKERS

Keyboard	Part Number
2741	X20-1783-0
1131	X20-1784-0

Figure 1.1

This may be appearance (ω over W), Greek-Roman equivalence (ρ over R), sequence (< ≤ = ≥ > ≠ over 3 4 5 6 7 8), or some -- possibly farfetched -- relationship between the APL function represented by the symbol and the letter (⋆ over P for power, ' over *K* for "kwote", and ⌈ over *S* for ceiling).

Because of the smaller number of keys available, the 1131 console uses a three-case system, also shown in Figure 1.1. The currently active case is indicated by the left and right halves of the ACCUMULATOR and ACCUMULATOR EXTENSION lights in the manner shown in Figure 1.2.

The two keys at the upper left control the cases, the arrows on the keys indicating the transitions (from the active case to the new active case) initiated by striking them. If the shift is in either upper or middle case but not locked (as indicated by the lights), it remains in that case for the next character only, and then returns to lower case.

If the shift is in either the upper or middle case but not locked, it may be locked in that state by a further depression of the shift key which brought it to that state. If the shift is locked in upper or middle case, it remains in that state until returned to lower case by the action of the appropriate shift key or by a carrier return. The 1131 Alpha/Numeric shift must remain in Alpha shift at all times.

For the console printer, the APL character set is provided by printing element #1167988. A particular 2741 terminal may use either the #1167988 element (for terminals using PTTC/BCD printing elements) or the #1167987 element (for terminals using Selectric printing elements).

However, any printing element may be used with the APL system, since the encoded characters generated by the keyboard and transmitted to the computer are independent of the particular element mounted on the terminal. Subject to programmed intervention, the transmitted information will always be interpreted according to the APL keyboard characters.

Non-APL printing elements are frequently useful in conjunction with special-purpose APL programs designed to exploit their character sets. Also, any element that matches the keyboard encoding (Selectric or PTTC/BCD) of the terminal can be used for straightforward numerical work, since letters and digits print properly with such elements. The visual interpretation of complex APL expressions is, of course, awkward with any but an APL printing element.

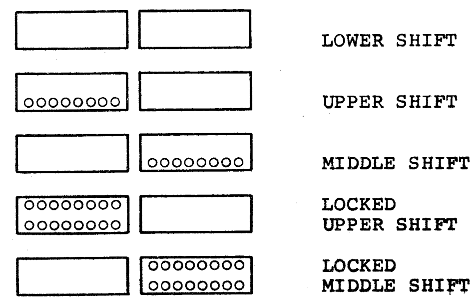


Figure 1.2: SHIFT CONDITION SIGNALS FOR 1130 KEYBOARD

STARTING THE SYSTEM

To start the APL system, turn the 1130 on, load the APL disk, mount the APL typing element (#1167988), and perform an Initial Program Load as follows:

1. Set console entry switch 0 down if the first sign-on is to come from the console keyboard, up if it is to come from the 2741 keyboard. Set all other console switches down.

2. Place the APLIPL card (APL initial program load) in the hopper.

3. Press IMMEDIATE STOP, RESET, and PROGRAM LOAD at the console.

The system will then be ready for the first sign-on.

STARTING AND ENDING A WORK SESSION

This section describes the procedures for beginning and ending a work session; more detail is given in Part 2.

The following procedures apply to operations from either the console keyboard or from a 2741 terminal. In the case of the 2741, a connection must first be established in the manner described in the succeeding section.

Communication with the computer is carried on by means of entries from the console keyboard (or auxiliary 2741 terminal), which alternately locks and unlocks as each entry is made and the computer completes its work. The general procedure is to type an instruction or command and strike the carrier return to indicate the end of the message.

Each user is assigned an account number. This number is used in the sign-on that begins a work session and serves to partially identify any work that the user may store in the system.

To begin a session, enter a right parenthesis followed by the appropriate account number, followed by a carrier return. For example:

)581

The account number may include a key (i.e., a colon and a password). For example:

)581:*ABC*

If the foregoing is properly executed and if the account number has previously been entered into the system (as described in the operator's manual), the computer will respond by typing the name associated with the account number, followed by *SIGNED ON* and the name of the system, e.g.:

SMITH SIGNED ON

A P L \ 1 1 3 0

If the sign-on is incorrect, one of the following trouble reports will be given:

INCORRECT SIGN-ON means that the form of the command was faulty.

NUMBER NOT IN SYSTEM means either exactly what it says or that the number has a lock associated with it and the wrong key was used.

INCORRECT COMMAND means that some account number is already signed on. Sign off, then execute the foregoing again.

To end a terminal session, enter the command:

)*OFF*

The computer will respond by typing:

SIGNED OFF

In the remainder of this manual the need for carrier return will not be explicitly mentioned, since it is required for every entry.

Mistakes. Before the carrier return that completes an entry, errors in typing can be corrected as follows: backspace to the point of error and then depress the linefeed button (marked ATTN on 2741 terminals and INT REQ on the 1131 console). This will have the effect of erasing everything to the right of, and including, the position of the carrier. The corrected text can be continued from that point on the new line.

If the keyboard of a 2741 terminal (either direct-connected or telephone-connected) simply unlocks after the carrier return in an attempted sign-on, repeat the sign-on procedure.

CONNECTING A 2741 TERMINAL

The directions that follow assume the use of a dial-up connection with a dataset. Instructions for the use of acoustic couplers should be obtained from their suppliers. Where terminals are connected to the computer by leased lines or private wires, instructions on dialing procedure (EC2) are irrelevant, but local sources of information should be consulted for equivalent procedures.

ACTION	NOTES
EC1. Set up terminal: Insert paper, mount an APL printing element, connect terminal to power source, and set switches as follows:	
LCL/COM — COM Power — ON	The power switch is at the right of the keyboard; the LCL/COM switch is on the left side of the terminal stand, toward the rear.
Test to see if the keyboard is locked by trying the shift key. If the key is operable, press the carrier return and test again.	If the keyboard does not lock after a carrier return, check the switches and try once more. If the switches are set properly and the keyboard remains unlocked, the terminal is faulty.
EC2. Dial computer: Set the telephone pushbutton switch to TALK and follow ordinary dialing procedure. After two rings, at most, the telephone will respond with a steady, high-pitched tone.	The telephone number is given in Table 1.3. If the line is busy, call the APL Operator to inquire about the schedule.

123 456-7890 Insert the access telephone number here. An assistance number should be included.
APL Operator: 123 456-7899

Table 1.3: TELEPHONE NUMBERS

ACTION	NOTES
Promptly set the pushbutton switch to DATA by holding the DATA button down firmly for a moment and then releasing. Cradle the handset.	The DATA button should light, and will remain lit as long as the terminal is connected to the computer. If it does not light, check the power connection to the dataset. If it lights, but quickly goes out, check the power connection to the terminal, the cable connection to the dataset, and the switch settings on the terminal. Then retry from EC1.
Response: The keyboard will unlock, indicating that the computer is ready to accept an entry from the terminal.	

Transmission errors. There are occasional transient failures in the communication between a terminal and the central computer. If the failure occurs during the transmission from the terminal, the system will respond by typing *RESEND*. The last entry from the keyboard should then be repeated.

Failures in the other direction are usually evidenced by the appearance of a spurious character, whose presence in the printed output is obvious in most contexts. However, there is no absolutely certain way of detecting such a failure.

LIMITED USE OF THE SYSTEM

No system commands other than the sign-on and sign-off described here are required in order to make use of Part 3, and the reading of Part 2 may therefore be deferred if only casual or restricted use is to be made of the system.

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 upon the (arbitrary) interpretation placed upon them. System commands in the APL\1130 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.

In this Part the structure of the APL\1130 system is described, and various notions essential to the understanding of system commands are introduced. Finally, the complete set of system commands is described in detail.

WORKSPACES AND LIBRARIES

Workspaces. The common organizational unit in the APL\1130 system is the workspace. When in use, a workspace is said to be active, and it occupies a block of working storage in the central computer. The size of the block, which is preset at a fixed value for a given system, determines the combined working and storage capacity of each workspace in that 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 present when the system is signed on. All transactions with the system are mediated by it. 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, and information on the progress of program execution is maintained in the state indicator of the active workspace.

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

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

NAMES

Names of workspaces, functions and variables may be formed of any sequence of alphabetic (*A* to *Z*) and numeric (0 to 9) characters that starts with an alphabetic and contains no blank. However, workspace names may not exceed 11 characters in length, and other names may not exceed six.

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. Also, since workspaces themselves are never the subject of APL operations, but only of system commands, it is possible for a workspace to have the same name as an object it holds. 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 pendent, 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 name of the workspace, when the workspace is stored. In order to activate a locked workspace or copy any information it contains, a colon and the password must again be used, as a key, in conjunction with the workspace name. Listings of workspace names never give the keys, and do not overtly indicate the existence of a lock.

Account numbers can be similarly protected by locks and keys, to avoid their unauthorized use and maintain the integrity of a user's private library.

Passwords for locks and keys may be formed of any sequence of alphabetic and numeric characters up to six characters long, without blanks. In use as either a lock or key, a password follows the number or name it is protecting, from which it is set off by a colon.

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. Since the keyboard is locked during printing or computing, the signal must be generated by means other than one of the standard keys.

The attention signal is generated by depressing the appropriate key once, firmly. On IBM 2741 terminals this key is usually of a distinctive color, and is marked ATTN; it is marked INT REQ on the 1131 console keyboard. (The same key is used for linefeed when the keyboard is not locked.)

Following an attention signal the keyboard will unlock, and the type carrier will return to the normal position for input (six 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.

Except for card control commands, the execution of system commands, once entered, cannot be interrupted.

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.

All system commands can be executed when the terminal is in the execution mode, in which APL operations are executed forthwith upon entry. However, in definition mode, in which sequences of operations are being composed into functions for later execution, commands which call for storing a copy of the workspace, or which might otherwise interfere with the definition process itself, are forbidden. (The two terminal modes are treated more fully in Part 3.)

Classification of commands. System commands are conveniently grouped into five classes with regard to their effect upon the state of the system:

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

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

3. Library control commands affect the state of the libraries.

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

5. Card control commands allow the reading of APL statements from cards rather than the keyboard, and the punching of variables and functions into cards for later use.

Reference and Purpose COMMAND FORM[1,2,3]	NORMAL RESPONSE	TROUBLE REPORTS[6]
TC1 Sign on designated user and start a work session.		
)NUMBER [KEY]	USER *SIGNED ON*; SYSTEM	1 8
TC2 End work session.		
)*OFF* [LOCK]	*SIGNED OFF*	11
WC1 Activate a clear workspace.		
)*CLEAR*		11
WC2 Activate a copy of a stored workspace.		
)*LOAD* WSID [KEY]	WSID *LOADED*	9 11
WC3 Copy global objects from a stored workspace.		
)*COPY* WSID [KEY] NAMES		2 4 7 9 10 11
WC3a Copy all global objects from a stored workspace.		
)*COPY* WSID [KEY]		2 7 9 10 11
WC4 Copy global objects from a stored workspace, protecting active ws.		
)*PCOPY* WSID [KEY] NAMES		2 4 7 9 10 11
WC4a Copy all global objects from a stored workspace, protecting active ws.		
)*PCOPY* WSID [KEY]		2 7 9 10 11
WC5 Erase global objects.		
)*ERASE* NAME[S]		3 11
LC1 Store a copy of the active workspace.		
)*SAVE* WSID [LOCK]	WSID *SAVED*	5 6 7 11
LC2 Erase a stored workspace.		
)*DROP* WSID	WSID *DROPPED*	9 11
IQ1 List names of defined functions.		
)*FNS*	FUNCTION NAMES	11
IQ2 List names of global variables.		
)*VARS*	VARIABLE NAMES	11
IQ3 List halted functions (state indicator).		
)*SI*	SEQUENCE OF HALTED FUNCTIONS	11
IQ4 List halted functions with local names (augmented state indicator).		
)*SIV*	SEQUENCE OF HALTED FUNCTIONS WITH LOCAL NAMES	11
IQ5 Give identification of active workspace.		
)*WSID*	WSID	11
IQ6 List names of stored workspaces.		
)*LIB* [NUMBER]	NAMES OF STORED WORKSPACES	8 11
CD1 Initiate reading of cards.[4]		
)*CARD* [[*NO*]*EDIT*] [[*NO*]*DISP*]		11
CD1a Continue reading cards but set edit and display options.[4]		
)*CARD* [[*NO*]*EDIT*] [[*NO*]*DISP*]		11
CD2 Stop reading cards and return control to the keyboard.[4]		
)*CARD END*		11
CD2a Run cards through to a stop[5] and return control to the keyboard.[4]		
)*CARD END*		11
CD3 Punch cards for global objects in the active workspace.		
)*PCH* NAMES		4 7 11
CD3a Punch cards for all global objects in the active workspace.		
)*PCH*		7 11
CD4 Punch cards for global objects in a stored workspace.		
)*PCHS* WSID [KEY] NAMES		4 7 9 11
CD4a Punch cards for all global objects in a stored workspace.		
)*PCHS* WSID [KEY]		7 9 11

Notes:
1. Items in brackets are optional.
2. KEY or LOCK: a password preceded by a colon.
3. WSID: library number and workspace name, or workspace name alone.
4. CD1 and CD2a are from the keyboard; CD1a and CD2 are from cards.
5. A stop is an attention signal, or a card with)*CARD END* or)*OFF*.
6. Trouble report forms:

1 *INCORRECT SIGN ON*	7 *NOT WITH OPEN DEFN*
2 *NOT COPIED* NAMES	8 *NUMBER NOT IN SYSTEM*
3 *NOT ERASED* NAMES	9 *WS NOT FOUND*
4 *NOT FOUND* NAMES	10 *WS FULL*
5 *NOT SAVED WS IS* WSID	11 *INCORRECT COMMAND*
6 *NOT SAVED WS QUOTA USED UP*	

Table 2.1: SYSTEM COMMANDS

The text that follows is based upon this classification, although it will be seen that one of the library control commands may sometimes affect the state of the active workspace.

Normal responses and trouble reports. Any entry starting with a right parenthesis will be interpreted by the system as an attempt to execute a system command. When the command is successfully executed, the normal response, if any, will be printed. This response is given in the description of the action for each command.

If, for any reason, a command cannot be completely executed, an appropriate trouble report will be printed. The most common report is *INCORRECT COMMAND*. This means that the command was incomplete, mis-spelled, used a wrong modifier, or was otherwise malformed. The corrective action in every case is to enter a properly composed command. The meanings and corrective actions for other trouble reports are given in the notes accompanying the description of each command.

Summary. The purposes, forms, responses, and trouble reports for all system commands are summarized in Table 2.1.

In general, the elements of a command form 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.

TERMINAL CONTROL COMMANDS

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

ACTION	NOTES
TC1. Start a work session: This is the sign-on, described in Part 1.	See Part 1, Starting and Ending A Work Session.
TC2. End work session: Enter *)OFF* followed by a colon and a password, if desired.	
Effect: 1. The currently active workspace will vanish.	There is no effect on any stored workspace.
2. 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 ending command that contains a colon. An existing lock is removed if no password follows the colon. If a colon is not used, the existing lock, if any, remains in force.
Response: 1. *SIGNED OFF*	Trouble report: *INCORRECT COMMAND*

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; and remove unwanted objects from the active workspace. No command in this class affects any but the active workspace.

Application packages. 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; for example, standard statistical calculations, exercises for teaching a scholastic subject, complex arithmetic, business accounting, text editing, etc. The workspace-centered organization of APL\1130 lends itself to such packaging, because each collection moves as a coherent 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 transfer between 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 selection of objects (i.e., functions or variables) from it. When an entire workspace is copied, all the functions and global variables within it are subject to the operation, but its state indicator and local variables are left behind.

Detailed Description. The term workspace identification is used here to mean either a library number followed by a workspace name, or a workspace name alone. When a name is used alone, the reference is to the user's private library. A key is a colon followed by a password.

ACTION	NOTES
WC1. Activate a clear workspace: Enter)*CLEAR*.	This command is used to make a fresh start, discarding whatever is in the active workspace.
Effect: 1. A clear workspace will be activated, replacing the presently active workspace.	A clear workspace has no variables or defined functions. Its workspace identification does not match that of any stored workspace. (See section on library control).
Response: None.	Trouble report: *INCORRECT COMMAND*
WC2. Activate a copy of a stored workspace: Enter)*LOAD* followed by a space and a workspace identification (with the key, if required).	This command may be used to obtain the use of any workspace in the system whose identification (and password) is known.
Effect: 1. A copy of the designated workspace will be activated, replacing the presently active workspace.	
Response: 1. The workspace name, followed by *LOADED*, is printed.	Trouble reports: *WS NOT FOUND* means either that there is no stored workspace with the given identification, or that no key, or the wrong key, was used when one was required. *INCORRECT COMMAND*

WC3. Copy global objects from a stored workspace:
Enter *)COPY*
followed by a space and a workspace identification (with the key, if required), and the names of objects to be copied, separated by spaces.

A global object may be a function or global variable.

Effect:
1. A copy of each designated object will appear in the active workspace with global significance, replacing existing global homonyms.

Objects having the same name as a halted function in the active workspace cannot be copied.

Response: None.

Trouble reports:
NOT WITH OPEN DEFN
means that the terminal is in definition mode. Either close the definition by entering ∇, or defer the copy operation.

WS NOT FOUND
See WC2 for meaning.

NOT COPIED
followed by the names of objects not copied, means those names were the names of halted functions in the active workspace.

NOT FOUND
followed by the names of objects not found, means that those names were not the names of global objects in the designated workspace.

WS FULL
means that the active workspace could not contain all the material requested.

INCORRECT COMMAND

WC3a. Copy all global objects from a stored workspace:
Enter *)COPY*
followed by a space and a workspace identification (with the key, if required).

See note at WC3.

Effect:
1. A copy of all functions and global variables in the source workspace will appear in the active workspace with global significance, replacing existing global homonyms.

Local variables and the state indicator are not copied.

Response: None.

Trouble reports:
NOT WITH OPEN DEFN
NOT COPIED
WS FULL
See WC3 for meanings.

WS NOT FOUND
See WC2 for meaning.

INCORRECT COMMAND

WC4. Copy global objects from a stored workspace, protecting the active workspace:
Enter *)PCOPY*
followed by a space and a workspace identification (with the key, if required), and the names of objects to be copied, separated by spaces.

See note at WC3.

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

Response:
1. *NOT COPIED*, followed by the names of objects not copied, will be printed if appropriate.

Trouble reports:
NOT WITH OPEN DEFN
NOT FOUND
WS FULL
See WC3 for meanings.

WS NOT FOUND
See WC2 for meaning.

INCORRECT COMMAND

WC4a. Copy all global objects from a stored workspace, protecting the active workspace:
Enter)*PCOPY*
followed by a space and a workspace identification (with the key, if required).

See note at WC3.

Effect:
1. 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.

See note at WC3a, Effect 1.

Response:
1. *NOT COPIED*, followed by the names of objects not copied, will be printed if appropriate.

Trouble reports:
NOT WITH OPEN DEFN
WS FULL
See WC3 for meanings.

WS NOT FOUND
See WC2 for meaning.

INCORRECT COMMAND

WC5: Erase global objects:
Enter)*ERASE*
followed by a space and the names of objects to be deleted, separated by spaces.

Effect:
1. Named objects having global significance, other than pendent functions, will be expunged.

Names which do not refer to global objects are ignored.

RESPONSE: None.

Trouble report:
NOT ERASED
followed by the names of functions not erased, means those functions are pendent.

INCORRECT COMMAND

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 a 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 may destroy a stored workspace by replacing it with a copy of the active workspace.

Continuity of work. When a workspace is stored, an exact copy of the active workspace is made, including the state indicator and intermediate results from the 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, to suit class schedules; and mathematical experimentation or the exploration of system models can be done over long periods of time, at the investigator's convenience.

Workspace identification. A library number and a name, together, uniquely identify each stored workspace in the system. An active workspace is also identified by a library number and a name, 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 assumes the identification assigned to the stored copy.

3. A clear workspace activated by a clear command or a sign-on, is called *CLEAR WS*, which cannot be the name of a stored workspace.

The identification of active workspaces is used as a safeguard against the inadvertent replacement of a stored workspace by an unrelated one: an attempt to replace, by a copy of the active workspace, any stored workspace other than the one with the same identification, will be stopped.

Library and account numbers. A user's account number is also the number of his private library.

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, a user is prevented from affecting the state of another user's private library. He may, of course, activate a copy of any workspace stored in the system, if he knows the library number and name (and password, if required).

Storage allotment. A user of APL\1130 is assigned library space in terms of the maximum number of stored workspaces he may have at one time. The allotment for each user is determined by those responsible for the general management of a particular system, within the bounds of the physical resources of the system.

Purging a workspace. The sequence of commands, *)SAVE ABC123*, *)CLEAR*, *)COPY ABC123*, will purge the active workspace, clearing it of all but its functions and global variables. This often results in more usable space than can otherwise be realized.

Detailed Description. The term workspace identification will be used with the same significance as for the workspace control commands.

ACTION	NOTES
LC1. Store a copy of the active workspace: Enter *)SAVE* followed by a space and a workspace identification, with a colon and password, if desired.	
Effect: 1. A copy of the active workspace will be stored with the designated identification, and with the assigned lock, if a password was used.	A stored workspace with the same identification will be replaced. A lock on a stored workspace will not be retained if the command does not include a lock explicitly.
2. The active workspace will assume the workspace identification used in the command.	To this extent only, this command may affect the state of the active workspace.
Response: 1. The workspace name, followed by *SAVED*, will be printed.	Trouble reports: *NOT WITH OPEN DEFN* means that the terminal is in function definition mode. Either close the definition by entering ∇, or defer the save operation. *NOT SAVED WS QUOTA USED UP* means that the allotted number of stored workspaces has previously been reached. Unless this is increased, the workspace can be stored only by first dropping a workspace already stored. *NOT SAVED WS IS* followed by identification of the active workspace, means a stored workspace with the identification used in the command exists, but this identification does not match that of the active workspace. *INCORRECT COMMAND*

LC2. Erase a stored work space:
Enter)*DROP*
followed by a space and a workspace identification.

Since a key is not used, a locked workspace whose key has been lost can always be removed from the system.

Effect:
1. The designated stored workspace will be expunged.

This command has no effect on the active workspace, regardless of its identification.

Response:
1. The workspace name, followed by *DROPPED*, will be printed.

Trouble reports:
WS NOT FOUND
means that there is no stored workspace with the identification used in the command.

INCORRECT COMMAND

INQUIRY COMMANDS

Most of the commands in this class refer to the state of the active workspace. One command lists the names of workspaces stored in libraries.

Detailed Description.

ACTION | NOTES

IQ1. List names of defined functions:
Enter)*FNS*

Effect: None.

Response:
1. The names of defined functions in the active workspace will be printed.

Trouble report
INCORRECT COMMAND

IQ2. List names of global variables:
Enter)*VARS*

Effect: None.

Response:
1. The names of global variables in the active workspace will be printed.

Trouble report
INCORRECT COMMAND

IQ3. List halted functions:
Enter)*SI*

Effect: None.

Response:
1. The names of halted functions will be listed, most recent ones first. With each name will be given the line number on which execution stopped. Suspended functions will be distinguished from pendent functions by an asterisk.

This display is the **state indicator**; its significance and use is explained in Part 3.

Trouble report
INCORRECT COMMAND

IQ4. List halted functions with names of local variables:
Enter)*SIV*

Labels (see Part 3) are included among the local variables.

Effect: None.

Response:
1. The response will be the same as for IQ3, except that with each function listed there will appear a listing of its local variables.

Trouble report
INCORRECT COMMAND

IQ5. Give identification of active workspace:
Enter)*WSID*

Effect: None.

Response:
1. The identification of the active workspace will be printed.

Trouble report
INCORRECT COMMAND

I06. List names of stored workspaces:
Enter)*LIB*
followed, if necessary, by a library number.

A library number is not required for listings of the user's private library.

Effect: None.

Response:
1. The names of workspaces in the designated library will be printed. If no number was used, the account number associated with the terminal will be taken as the library number.

Trouble report
NUMBER NOT IN SYSTEM means exactly that.

INCORRECT COMMAND

CARD CONTROL COMMANDS

The commands in this class provide for the use of punched cards. They allow the user to:

1. Prepare APL statements and system commands on an 029 keypunch for future entry to APL\1130, using the)*CARD* and)*CARD END* commands.

2. Have the APL system punch copies of variables and functions into cards for later use, using the)*PCH* and)*PCHS* commands.

Preparing APL statements on cards. Statements for input to the APL\1130 system may be punched into cards in free format in columns 1-71. If a statement is too long to fit onto one card, any non-blank character can be punched in column 72 to indicate that the statement continues on the next card. Columns 73-80 are ignored when the cards are being read, and can be used for identification or sequence numbers. They are so used by the punch commands, which place the name of the variable or function in columns 73-78, and the sequence numbers for that name in columns 79 and 80.

A list of APL characters and their keypunch equivalents is given in Table 2.2. All the letters and digits and many of the APL operators (e.g., +, /, ?, =) are available as single characters on the keypunch. The symbols ←, [,], ÷, ×, and - are represented on the keypunch by substitute characters, e.g., ← is represented by #.

APL	029	Card Code[1]	APL	029	Card Code[1]
A-Z	A-Z		∨	@OR	
'	'	8-5	∧	@AND	
0-9	0-9		~	@NOT	
¯	- (Note 2)		⊥	@BASE	
.	.		⊤	@REP	
(	(	12-8-5	∊	@EPS	
)	)	11-8-5	⍳	@IOTA	
[	$		⍴	@RHO	
]	¢	12-8-2	,	,	
;	;	11-8-6	/	/	
←	#	8-3	⌿	@CCMP	
→	@GO		\	@REXP	
→	@GOTO		⍀	@CEXP	
+	+	12-8-6	↑	@TAKE	
-	¬	11-8-7	↓	@DROP	
×	&	12	⌽	@RROT	
÷	%	0-8-4	⌽	@RREV	
*	*	11-8-4	⊖	@CROT	
⍟	@LOG		⊖	@CREV	
?	?	0-8-7	⍉	@TRAN	
!	!	11-8-2	∆	@DELTA	
⌊	@MIN		∘	@NULL	
⌊	@FLOOR		⎕	@QUAD	
⌈	@MAX		⍞	@QQ	
⌈	@CEIL		∇	@DEFN	
|	|	12-8-7	⍫	@LOCK	
○	@TRIG		:	:	8-2
<	<	12-8-4	_	_	0-8-5
≤	@LEQ		⊂	@LH	
=	=	8-6	⊃	@RH	
≥	@GEQ		∩	@CAP	
>	>	0-8-6	∪	@CUP	
≠	@NEQ				

Note 1. APL\1130 accepts 029 keypunch codes. Card codes are given here only for those which are not directly available on 026 keypunches, or for which some keypunches use different codes.
Note 2. This is the negative sign, used in numeric constants.

Table 2.2: Mnemonics and card codes for the APL character set.

For other non-alphanumeric characters a system of mnemonic names is used. These mnemonics always consist of an at-sign (@) followed by a string of letters designed to suggest the name or use of the symbol. Thus the mnemonic for ρ is @RHO, the mnemonic for ⌈ can be either @MAX or @CEIL, and so on. Mnemonics may be used in APL statements in exactly the way the corresponding symbols would be used, except that a blank must be left at the end of the mnemonic. For instance, the statement *N←ρQ* could be punched in a card as N←@RHO Q.

Detailed description. The term stop is used here to mean the occurrence of a card punched with *)CARD END* or *)OFF*, or an attention signal, while cards are being read.

ACTION	NOTES
CD1. Initiate reading of cards. Enter *)CARD* followed, if desired, by one of each of the pairs: *EDIT* or *NOEDIT*, *DISP* or *NODISP*.	This command enables the system to accept APL statements and system commands from the card reader, rather than from the keyboard. All statements and commands except *)PCH* and *)PCHS* may be executed from both sources.
Effect: 1. The keyboard will lock. 2. The system will accept instructions from the card reader until a stop occurs or an error is encountered.	If *EDIT* has been specified, the keyboard unlocks immediately when an error occurs or an attention signal is given. If *NOEDIT* is specified, the cards are flushed up to a stop before the keyboard is unlocked. (This is the same as the effect of CD2a.) If *DISP* is specified, the card input will be displayed on the typewriter; if *NODISP* is specified, it will not. If the options are not explicitly chosen in the command, *EDIT* or *DISP* is assumed.
Response: None.	Trouble report *INCORRECT COMMAND*

CD1a. Continue card reading with specified edit and display options. Include in the card deck a card punched with *)CARD* followed, if desired, by one of each of the pairs: *EDIT* or *NOEDIT*, *DISP* or *NODISP*.	
Effect: 1. The edit or display options will be set appropriately.	
Response: None.	Trouble report *INCORRECT COMMAND*
CD2. Stop reading cards and return control to the keyboard: Include in the card deck a card punched with *)CARD END*.	
Effect: 1. Card reading stops. 2. The keyboard unlocks.	
Response: None.	Trouble report *INCORRECT COMMAND*
CD2a. Run cards through to stop and return control to the keyboard: Enter *)CARD END*	
Effect: 1. Cards are read but instructions are not executed until a stop occurs. 2. The keyboard unlocks.	If cards run out before a stop card occurs the keyboard remains locked. An attention signal can be used to return control to the keyboard.
Response: None.	Trouble report *INCORRECT COMMAND*

CD3. Punch cards for global objects in the active workspace:
Enter *)PCH*
followed by a space and one or more names of objects, separated by spaces.

Locked functions (see Part 3) cannot be punched.

Effect:
1. A copy of each named object is punched into cards.

The cards produced by a punch command can be read into the system at a later time with a card command. The effect will be to restore the functions and variables (in the active workspace) as they were when the cards were punched.

Punch commands encountered when the system is reading cards result in an *INCORRECT COMMAND* report.

Response: None.

Trouble reports
NOT WITH OPEN DEFN
means that the terminal is in definition mode. Either end the definition mode by entering ∇, or defer the punching operation.

NOT FOUND
followed by a list of names means those names did not refer to global objects in the workspace.

INCORRECT COMMAND

CD3a. Punch cards for all global objects in the active workspace:
Enter *)PCH*

See note at CD3.

Effect:
1. A copy of each function and global variable in the active workspace is punched into cards.

Response: None.

Trouble reports
NOT WITH OPEN DEFN
See CD3 for meaning.

INCORRECT COMMAND

CD4. Punch cards for global objects in a stored workspace.
Enter *)PCHS*
followed by a space and the workspace identification (with the key, if required), and one or more names of objects, separated by spaces.

See note at CD3.

Effect:
1. A copy of each named object is punched into cards.

Response: None.

Trouble reports
NOT WITH OPEN DEFN
NOT FOUND
See CD3 for meanings.

WS NOT FOUND
means either there is no stored workspace with the given identification, or the wrong key (or no key) was used when one was required.

INCORRECT COMMAND

CD4a. Punch cards for all global objects in a stored workspace:
Enter)*PCHS*
followed by a space and the workspace identification (with the key, if required).

Effect:
1. A copy of each function and global variable in the designated workspace is punched into cards.

Response: None.

See note at CD3.

Trouble reports
NOT WITH OPEN DEFN
See CD3 for meaning.

WS NOT FOUND
See CD4 for meaning.

INCORRECT COMMAND

PART 3

THE LANGUAGE

The APL\1130 Terminal System executes system commands or mathematical statements entered on a terminal typewriter. 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 their definitions on 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. If ever the system seems unusually slow to respond, execute an attention signal to interrupt execution and unlock the keyboard.

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

The primitive functions and the defined functions available in libraries can be used without knowledge of the means of defining functions. These means are treated in the four contiguous sections beginning with Defined Functions and ending with Homonyms. These sections may be skipped without loss of continuity.

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 to 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 printed. For example:

```
      3×4
12
```

Results typed by the system begin at the left margin whereas entries from the keyboard are automatically indented. The keyboard arrangement is shown in Figure 1.2.

Scalar and vector constants. All numbers entered via the keyboard or typed out 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 1.44E2 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 to be distinguished from the negation function which is denoted, as usual, by the minus sign -.

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, and a sequence of characters entered in quotes represents a vector whose successive components are the characters themselves. Such a vector is printed 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 abbreviation of *CANNOT* is entered as '*CAN''T*' and prints as *CAN'T*.

Names and Spaces. As noted in Part 2, the name of a variable or defined function may be any sequence of six or fewer letters or digits beginning with a letter and not containing a space.

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 3+4 may be entered with no spaces, but if *F* is a defined function, then the expression 3 *F* 4 must be entered with the indicated spaces. The exact number of spaces used in succession is of no importance and extra spaces may be used freely.

Overstriking and erasure. Backspacing serves only to position the carriage 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. ⍫ and ⌽), and

3. to position the carriage for erasure, which is effected by striking the linefeed (marked ATTN on IBM 2741 terminals and INT REQ on the 1131 console keyboard). The linefeed has the effect of erasing the character at the position of the carriage, and all characters to the right.

End of Statement. The end of a statement is indicated by striking the carriage return. The typed entry is then interpreted exactly as it appears on the page, regardless of the time sequence in which the 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*←(3⌈*Q*)÷*X*×*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*←(3⌈*Q*)÷(*X*×(*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 of Table 3.1 to be typed out. The error report will be followed by the offending statement with a caret typed under the point in the statement where the error was detected. If the caret lies to the right of a specification arrow, the specification has not yet been performed.

TYPE	Cause; CORRECTIVE ACTION
CHARACTER	Illegitimate overstrike.
DEFN	Misuse of ∇ or ⎕ symbols: 1. ∇ is in some position other than the first. 2. The function is pendent. DISPLAY THE STATE INDICATOR AND CLEAR AS REQUIRED. 3. Use of more of the header than ∇*F* to reopen the definition of *F*. 4. Improper request for a line edit or display. 5. The function is locked.
FN SPACE FULL	Function portion of workspace cannot hold more functions. ERASE FUNCTIONS NO LONGER REQUIRED.
DOMAIN	Arguments not in the domain of the function.
FUNCTION TOO LARGE	Too many lines, labels, or local variables in a function. REVISE OR SEGMENT THE FUNCTION.
ID	Identifier with more than six characters.
INDEX	Index value out of range.
LABEL	Definition of a label that is already an argument, explicit result, or label for the function.
LENGTH	Shapes not conformable.
LINE TOO LONG	Statement with more than 160 characters. SEGMENT THE STATEMENT.
RANK	Ranks not conformable.
SYNTAX	Invalid syntax; e.g., two variables juxtaposed; function used without appropriate arguments as dictated by its header; mismatched parentheses.
SYSTEM	Fault in internal operation of APL\1130.
VALUE	Use of a name that has not been assigned a value. ASSIGN A VALUE TO THE VARIABLE OR DEFINE THE FUNCTION.
WS FULL	The portion of the workspace for variables is filled (perhaps by temporary values produced in evaluating a compound expression). CLEAR THE STATE INDICATOR, ERASE NEEDLESS OBJECTS, OR REVISE CALCULATIONS TO USE LESS SPACE.

Table 3.1 ERROR REPORTS

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 a right arrow followed by a zero (→0), and then retry with an amended statement. The 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 3.2 and 3.8, 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*÷*Y* may be expressed as "*X* divided by *Y*", or "*X* over *Y*"). Thus, the expression ρ*M* yields the dimension of the array *M*, but the terms size or shape may be preferred both for their brevity and for the fact that 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" (ι) for "index generator"), probably to avoid unwanted connotations of any of the chosen names.

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 four ways: element-by-element, reduction, inner product, and outer product, as described in the section on Functions on Arrays. Mixed functions are discussed in a later section.

The scalar functions are summarized in Table 3.2. Each 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, this being primarily a matter of the representation in a particular medium, 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.

Precision of numbers. Integers less than 2 to the power 23 are carried with full precision; larger numbers and non-integers are carried to a precision of 6 to 7 decimal digits.

<table>
<tr><th colspan="2">Monadic form fB</th><th>f</th><th colspan="2">Dyadic form AfB</th></tr>
<tr><th>Definition or example</th><th>Name</th><th></th><th>Name</th><th>Definition or example</th></tr>
<tr><td>+B ↔ 0+B</td><td>Plus</td><td>+</td><td>Plus</td><td>2+3.2 ↔ 5.2</td></tr>
<tr><td>-B ↔ 0-B</td><td>Negative</td><td>-</td><td>Minus</td><td>2-3.2 ↔ ¯1.2</td></tr>
<tr><td>×B ↔ (B>0)-(B<0)</td><td>Signum</td><td>×</td><td>Times</td><td>2×3.2 ↔ 6.4</td></tr>
<tr><td>÷B ↔ 1÷B</td><td>Reciprocal</td><td>÷</td><td>Divide</td><td>2÷3.2 ↔ 0.625</td></tr>
<tr><td rowspan="2"><table><tr><th>B</th><th>⌈B</th><th>⌊B</th></tr><tr><td>3.14</td><td>4</td><td>3</td></tr><tr><td>¯3.14</td><td>¯3</td><td>¯4</td></tr></table></td><td>Ceiling</td><td>⌈</td><td>Maximum</td><td>3⌈7 ↔ 7</td></tr>
<tr><td>Floor</td><td>⌊</td><td>Minimum</td><td>3⌊7 ↔ 3</td></tr>
<tr><td>*B ↔ (2.71828..)*B</td><td>Exponential</td><td>*</td><td>Power</td><td>2*3 ↔ 8</td></tr>
<tr><td>⍟*N ↔ N ↔ *⍟N</td><td>Natural logarithm</td><td>⍟</td><td>Logarithm†</td><td>A⍟B ↔ Log B base A
A⍟B ↔ (⍟B)÷⍟A</td></tr>
<tr><td>|¯3.14 ↔ 3.14</td><td>Magnitude</td><td>|</td><td>Residue</td><td><table><tr><th>Case</th><th>A|B</th></tr><tr><td>A≠0</td><td>B-(|A)×⌊B÷|A</td></tr><tr><td>A=0,B≥0</td><td>B</td></tr><tr><td>A=0,B<0</td><td>Domain error</td></tr></table></td></tr>
<tr><td>!0 ↔ 1
!B ↔ B×!B-1</td><td>Factorial</td><td>!</td><td>Binomial coefficient</td><td>A!B ↔ (!B)÷(!A)×!B-A
2!5 ↔ 10 3!5 ↔ 10</td></tr>
<tr><td>?B ↔ Random choice from ⍳B</td><td>Roll</td><td>?</td><td>Deal†</td><td>A Mixed Function (See Table 3.8)</td></tr>
<tr><td>○B ↔ B×3.14159...</td><td>Pi times†</td><td>○</td><td>Circular†</td><td>See Table at left</td></tr>
<tr><td>~1 ↔ 0 ~0 ↔1</td><td>Not</td><td>~</td><td></td><td></td></tr>
<tr><td colspan="2" rowspan="2"><table><tr><th>(-A)○B</th><th>A</th><th>A○B</th></tr><tr><td>(1-B*2)*.5</td><td>0</td><td>(1-B*2)*.5</td></tr><tr><td>Arcsin B</td><td>1</td><td>Sine B</td></tr><tr><td>Arccos B</td><td>2</td><td>Cosine B</td></tr><tr><td>Arctan B</td><td>3</td><td>Tangent B</td></tr><tr><td>(¯1+B*2)*.5</td><td>4</td><td>(1+B*2)*.5</td></tr><tr><td>Arcsinh B</td><td>5</td><td>Sinh B</td></tr><tr><td>Arccosh B</td><td>6</td><td>Cosh B</td></tr><tr><td>Arctanh B</td><td>7</td><td>Tanh B</td></tr></table>Table of Dyadic ○ Functions</td><td>∧
∨
⍲
⍱</td><td>And
Or
Nand†
Nor†</td><td><table><tr><th>A</th><th>B</th><th>A∧B</th><th>A∨B</th><th>A⍲B</th><th>A⍱B</th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table></td></tr>
<tr><td><
≤
=
≥
>
≠</td><td>Less
Not greater
Equal
Not less
Greater
Not Equal</td><td>Relations
Result is 1 if the relation holds, 0 if it does not:
3≤7 ↔ 1
7≤3 ↔ 0</td></tr>
<tr><td colspan="5">†This function not available on APL\1130</td></tr>
</table>

Table 3.2: PRIMITIVE SCALAR FUNCTIONS

For functions such as floor and ceiling, and in comparisons, a "fuzz" of about $1E^{-}7$ is applied in order to avoid anomalous results that might otherwise be engendered by doing decimal arithmetic on a binary machine.

Two of the functions of Table 3.2, the relations ≠ and =, are defined on characters as well as on numbers.

Monadic and dyadic functions. Each of the functions defined in Table 3.2 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, ⌈*Y* denotes the monadic function ceiling applied to the single argument *Y*, and *X*⌈*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 of Table 3.2 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 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 on dividing *B* by *A*, and may be stated more generally as the smallest non-negative member of the set *B*-*N*×*A*, where *N* is any integer.

This formulation covers the case of a zero left argument as shown in Table 3.2. The conventional definition is extended in two further respects:

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

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

In APL\1130, the domain of the ? function is limited to positive integer arguments less than 32768.

The factorial function !*N* is defined in the usual way as the product of the first *N* positive integers. The function *A*!*B* (pronounced *A* out of *B*) is defined as (!*B*)÷(!*A*)×!*B*-*A* and 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 3.2 exhaust the nontrivial logical functions of two logical arguments.

Vectors. Each of the monadic functions of Table 3.2 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 one scalar (or a single element vector or matrix) 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 length 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
                   Empty vector prints as a blank
      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
```

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

```
      ∇SPHERE
[1]   SURF←4×3.14159×R×R
[2]   VOL←SURF×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 *SPHERE*) 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. In APL\1130, the number of statements in a function body may not exceed 50.

The act of defining a function neither executes nor checks for validity the statements in the body. After the definition of a function is completed, entering the name of the function causes the execution of the statements in the body. For example:

```
      ∇SPHERE                   Definition of the
[1]   SURF←4×3.14159×R×R        function SPHERE
[2]   VOL←SURF×R÷3
[3]   ∇
      R←2                       Specification and display
      R                         of the argument R
2
      SURF                      SURF has not been
VALUE ERROR                     assigned a value
      SURF
      ∧
      SPHERE                    Execution of SPHERE
      SURF                      SURF and VOL now have
50.2654                         values assigned by the
      VOL                       execution of SPHERE
33.5103
      R←1                       Use of SPHERE for
      SPHERE                    a new value of the
      SURF                      argument R
12.5664
      VOL
4.18879
```

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 executed next. 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 is vacuous and the normal sequence is followed.

The following examples illustrate various methods of branching used in three equivalent functions (*SUM*, *SUM*1, and *SUM*2) for determining *S* as the sum of the first *N* integers:

```
      ∇SUM
[1]   S←0
[2]   I←1
[3]   →4×I≤N            Branch to 4×1 (i.e., 4) or to 4×0 (out)
[4]   S←S+I
[5]   I←I+1
[6]   →3                Unconditional branch to 3
[7]   ∇
      N←1
      SUM
      S
1
      N←2
      SUM
      S
3
      N←5
      SUM
      S
15
      ∇SUM1             Equivalent to SUM
[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
      SUM1
      S
15
      ∇SUM2             Equivalent to SUM
[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*."

<u>Local and global variables</u>. A variable is normally <u>global</u> in the sense that its name has the same significance no matter what function or functions it may be used in. However, the iteration counter *I* occurring in the foregoing function *SUM* is of interest only during execution of the function; it is frequently convenient to make such a variable <u>local to a function</u> 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 following behavior of the function *SUM*3, which has a local variable *I*, with the behavior of the function *SUM*2 in which *I* is global:

```
      ∇SUM3;I                    ∇SUM2

[1]   S←0                  [1]   S←0
[2]   I←0                  [2]   I←0
[3]   S←S+I                [3]   S←S+I
[4]   I←I+1                [4]   I←I+1
[5]   →3×⍳I≤N              [5]   →3×⍳I≤N
[6]   ∇                    [6]   ∇
      I←20                       I←20
      N←5                        N←5
      SUM3                       SUM2
      S                          S
15                         15
      I                          I
20                         6
```

Since *I* is local to the function *SUM*3, execution of *SUM*3 has no effect on the variable *I* referred to before and after the use of *SUM*3.

However, if the variable *K* is local to a function *F* then any function *G* used within *F* may refer to the same variable *K*, unless the name *K* is further localized by being made local to *G*. For further treatment of this matter, see the section on Homonyms.

Explicit argument. A function definition of the form

```
      ∇SPH X
[1]   SUR←4×3.14159×X×X
[2]   ∇
```

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

```
      SPH 2
      SUR
50.2654
      SPH 1
      SUR
12.5664
```

Any explicit argument of a function is automatically made local to the function; if *E* is any expression, then the effect of *SPH E* is to assign to the local variable *X* the value of the expression *E* and then execute the body of the function *SPH*. 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 therefore appear in a compound expression such as *X*+÷*Z*. A function definition of the form

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

defines *SP* 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×SP 1
      Q
37.6991
      R←2
      (SP R)×R÷3
33.5103
```


The 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 header used to define each of these six types is shown in Table 3.3. 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 3.3: FORMS OF DEFINED FUNCTIONS

It is not obligatory either for the arguments of a defined function to be used within the body, or for the result variable to be specified. A function definition which does not assign a value to the result variable will engender a value error report when it is used within a compound expression. This behavior permits a function to be defined with a restricted domain, by testing the argument(s) and branching out in certain cases without specifying a result. For example:

```
      ∇Z←SQRT X
[1]   →0×⍳X<0
[2]   Z←X*.5∇
      Q←SQRT 16
      Q
4
      Q←SQRT ¯16
VALUE ERROR
      Q←SQRT ¯16
        ∧
```

Use of defined functions. A defined function may be used in the same ways that a primitive function may. In particular, it may be used within the definition of another function. For example, the function *HYP* determines the hypotenuse of a right triangle of sides *A* and *B* by using the square root function *SQRT*:

```
      ∇Z←SQRT X
[1]   Z←X*.5∇

      ∇H←A HYP B
[1]   H←SQRT (A*2)+B*2∇

      5 HYP 12
13
```

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

Recursive function definition. A function *F* may be used in the body of its own definition, in which case the function is said to be recursively defined. The following program *FAC* 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 *FAC N*-1, except for the case *N*=0 when it is determined (by statement 4) as 1:

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

Trace control. A trace is an automatic type-out 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 typed out 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∆P*. If one types *T∆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∆P*←0 will discontinue tracing of *P*. A complete trace of *P* is set up by entering *T∆P*←⍳*N*, where *N* is the number of statements in *P*.

MECHANICS OF FUNCTION DEFINITION

When a function definition is opened (by typing a ∇ followed by a header), the system automatically types successive statement numbers enclosed in brackets and accepts successive entries as the statements forming the body of the definition. The system is therefore said to be in definition mode, as opposed to the execution mode which prevails outside of function definition.

There are several devices which may be used during function definition to revise and display the function being defined. After function definition has been closed, there are convenient ways to re-open the definition so that these same devices may be used for further revision or display.

Labels. If a statement occurring in the body of a function definition is prefaced by a name and a colon, then at the end of the definition the name is assigned a value equal to the statement number. A name specified in this way is called a label. Labels are used to advantage in branches when it is expected that a function definition may be changed for one reason or another, since a label automatically assumes the new value of the statement number of its associated statement as statements are inserted or deleted. A label is a local name as if it had occurred in the header; unlike a local variable it cannot be respecified.

Revision. Any statement number (including one typed 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 N is zero, it refers to the header line of the function.

If a statement number is used again, the new text associated with it replaces the old statement. If any statement is empty -- that is, the bracketed statement number was immediately followed by both a linefeed and a carriage return (a carriage return alone is vacuous) -- it is deleted.

When the 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. Labels are assigned appropriate values.

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.

Display. During function definition, statement N can be displayed by overriding the line number with [N⎕]. After the display, the system awaits replacement of statement N. Typing [⎕] displays the entire function, including the header and the opening and closing ∇, and awaits entry of the next statement; typing [⎕N] displays all statements from N onward and awaits replacement of the last statement. Executing an attention signal will stop any display.

Line editing. During function definition, statement N can be modified by the following mechanism:

1. Type [N⎕M] where M is an integer.

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

3. A letter or 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 backspace and linefeed apply.

4. When the carriage is returned, statement N is re-displayed without the line number. Each character understruck by a / is deleted, each character understruck by a digit K is preceded by K added spaces, and each character understruck by a letter is preceded by 5×*R* spaces, where *R* is the position of the letter in the alphabet. Finally, the carriage moves to the first injected space 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; in particular, a completely blank sequence leaves the statement unchanged.

A new statement number (in brackets) can be entered in the space left for it 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.

Reopening function definition. If a function *R* is already defined, the definition mode for that function can be re-established by entering ∇*R* alone; the rest of the function header must not be entered. The system responds by typing [*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 typing [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. Also, ∇*R*[⎕]∇ causes the entire definition of *R* to be displayed, after which the system returns to execution mode.

Similar expressions involving display are also permissible, for example, ∇*R*[⎕3]∇ or ∇*R*[⎕] or ∇*R*[2⎕10].

Locked functions. If the symbol ⍫ (formed by a ∇ overstruck with a ~ and called del-tilde) is used instead of ∇ to open or close a function definition, the function becomes locked. A locked function cannot be revised or displayed in any way. Moreover, an error stop within the function will print only the function name and the type of error, not the statement. Finally, the trace control vector for a function cannot be changed after the function is locked.

Function locks are used to keep a function definition proprietary. For example, in an exercise in which a student is required to determine the behavior of a function by using it for a variety of arguments, locking a 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 Table 2.1). A variable also may be deleted by the erase command.

System command entered during function definition. A system command entered during function definition will not be accepted as a statement in the definition. Some commands, such as)*COPY*, will be rejected with the message *NOT WITH OPEN DEFN* (see Table 2.1); most will be executed immediately.

SUSPENDED FUNCTION EXECUTION

Suspension. The execution of a function *F* may be stopped before completion in two ways: by an error report, or by an attention signal. In either 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.

Whatever the reason for suspension, the statement or statement number displayed is the next one to have been executed. A branch to that statement number will cause normal continuation of the function execution, and a branch out (→0) will terminate execution of the function.

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

1. execute statements or system commands.

2. resume execution of the function at an arbitrary point *N* (by entering →*N*).

3. reopen the definition of any function which is not pendent. The term pendent is defined in the discussion of the state indicator below.

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 at statement 7 of the 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*. (No line number is printed for locked functions). The ⋆ appearing to the right of *H*[7] indicates that the function *H* is suspended; the functions *G* and *F* are said to be pendent.

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]
```

It is recommended that the state indicator be cleared before modifying a program that uses statement labels. Changing the values of statement labels (by adding or removing statements) in the function will not affect the label values for a suspended execution, and if execution of the suspended function is continued, branch instructions may result in branches to the wrong statements.

The state indicator can often be cleared by repeated entry of →0. If this does not work, it can be cleared by a sequence of commands of the following form:)*SAVE A*)*CLEAR*)*COPY A*

Trace control vectors may be set within functions. In particular, they may be set by expressions which initiate or discontinue tracing according to the values of certain variables.

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: when a function is executed, its local variables supersede, for the duration of the execution, other objects of the same names. A name may, therefore, be said to have one active referent and (possibly) several latent referents.

The complete set of referents of a name can be determined with the aid of the SIV list (state indicator with local variables), whose display is initiated by the command *)SIV*. The SIV list contains the information provided by the command *)SI*, augmented by the names of the variables local to each function (including labels). A sample display follows:

```
      )SIV
G[7]      * Z X I
F[4]        P J
Q[3]      * C X T
R[2]        P
G[3]        Z X I
```

If the SIV list is scanned downward, from the top, the first occurrence of a variable is the point at which its active referent was introduced; lower occurrences are the points at which currently latent referents were introduced; and if the name is not found at all, its referent is global, and should be sought for with the commands *)FNS* or *)VARS*.

As the state indicator is cleared by the continuation to completion of halted functions, latent referents become active in the sequence summarized, for the preceding SIV list, by the following diagram:

```
         Z X I P J C T A B
      G  ↓ ↓ ↓ | | | | | |
      F  | | | ↓ ↓ | | | |
      Q  | ↓ | | | ↓ ↓ | |
      R  | | | ↓ | | | | |
      G  ↓ ↓ ↓ | | | | | |
 Global  ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
```

The currently active referent of a name holds down to and including the execution of the function listed at the point of the first arrow, because of localization of the

name within that function. The first latent referent becomes active when that function is completed, and holds down to the next arrow; and so forth until the state indicator is completely cleared, at which point there are no longer any latent referents, and all active referents are global objects.

Function names. All function names are global. In the foregoing example, therefore, a function named *P* cannot be used within the function *R* or within any of the functions employed by *R*, since the local variable name *P* makes the function *P* inaccessible. However, even in such circumstances, the opening of function definition for such a function *P* is possible. (Moreover, as stated in Part 2, system commands concern global objects only, regardless of the current environment.)

This scheme of homonyms is easy to use and relatively free from pitfalls. It can, however, lead to seeming anomalies as indicated by the following example (shown to the authors by J.C.Shaw) of two pairs of functions which differ only in the name used for the argument:

```
      ∇Z←F X                  ∇Z←F X
[1]   Z←X+Y∇            [1]   Z←X+Y∇

      ∇Z←G Y                  ∇Z←G R
[1]   Z←F Y∇            [1]   Z←F R∇
      Y←3                     Y←3
      G 4                     G 4
8                       7
```

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

For example:

```
      1000 CPI 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:

```
      CI
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) occurring in such an exchange be accepted not as an ordinary entry (which would only evoke the response 1000, etc.), but as data to be used within the function *CI*. Facilities for this are provided in two ways, termed evaluated input, and character input.

The definition of the function *CI* is shown at the end of 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 printed to alert the user to the type of input expected, the paper is spaced up one line, and the keyboard unlocks. 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
```


An invalid entry in response to request for a quad input induces an appropriate error report, after which input is again awaited at the same point. A system command entered will be executed, after which (except in the case of one which replaces the active workspace) a valid expression will again be awaited. An empty input (i.e., a carriage return alone or spaces and a carriage return) is rejected and the system again prints the symbols ⎕: and awaits input.

Character input. The quote-quad symbol ⍞ (i.e., a quad overstruck with a quote) accepts character input: the keyboard unlocks at the left margin and data entered are 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 printed. Hence, ⎕←*X* is equivalent to the statement *X*. The longer form ⎕←*X* is useful when employing multiple specification. For example, ⎕←*Q*←*X**2 assigns to *Q* the value *X**2 and then prints the value of *X**2.

Heterogeneous output. A sequence of expressions separated by semi-colons will cause the values of the expressions to be printed, with no intervening carriage returns or spaces except those implicit in the display of the values.

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*←2 14 , then:

```
   'THE VALUE OF X IS ';X
THE VALUE OF X IS 2 14
```

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

```
     ∇CI;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. In APL\1130, arrays of rank greater than rank 2 cannot be used and no dimension of an array may exceed 256; thus, a vector may have no more than 255 elements and a matrix may have no more than 255 rows or columns.

This section treats the reshaping and indexing of arrays, and the form of array output. The following section treats the four 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 ρ*X* denotes its dimension. For example, if *X*←2 3 5 7 11, then ρ*X* is 5, and if *Y*←'*ABC*' , then ρ*Y* 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*.

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 ρ*A* is the vector 3 4.

Since ρ*A* contains one component for each coordinate of *A*, the expression ρρ*A* is the rank of *A*. Table 3.4 illustrates the values of ρ*A* and ρρ*A* for arrays of rank 0 (scalars) up to rank 2. In particular, the function ρ applied to a scalar yields an empty vector.

Type of Array	ρ*A*	ρρ*A*	ρρρ*A*
Scalar		0	1
Vector	*N*	1	1
Matrix	*M N*	2	1

Table 3.4: 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 ρ reshapes its right argument to the dimension specified by its left argument. If *M*←*D*ρ*V*, then *M* is an array of dimension *D* whose elements are the elements of *V*. For example, 2 3ρ1 2 3 4 5 6 is the matrix

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

If *N*, the total number of elements required in the array *D*ρ*V*, is equal to the dimension of the vector *V*, then the ravel of *D*ρ*V* is equal to *V*. If *N* is less than ρ*V*, then only the first *N* elements of *V* are used; if *N* is greater than ρ*V*, then the elements of *V* are repeated cyclically. For example, 2 3ρ1 2 is the matrix

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

and 3 3ρ1 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.

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 ι0 and '' and ρ applied to any scalar. An empty vector prints as 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 BASE N
[1]   Z←ι0
[2]   Z←(B|N),Z
[3]   N←⌊N÷B
[4]   →2×N>0∇
      10 BASE 1776
1  7  7  6
      8 BASE 1776
3  3  6  0
```

Empty arrays of higher rank can be useful in analogous ways in conjunction with the expansion 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]` induces an index error report.

In general, `⍴X[I]` is equal to `⍴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 3⍴3 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

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

then `M[2;3]` is the element `7` and `M[1 3; 2 3 4]` is the matrix

```
 2   3   4
10  11  12
```

In general, `⍴M[I;J]` is equal to `(⍴I),⍴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, and if `I` is a matrix and `J` is a scalar (or vice versa), `M[I;J]` is a matrix.

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

```
 2   1
 6   5
10   9
```

Permutations are an interesting use of indexing. A vector `P` whose elements are some permutation of its own indices is called a permutation of order `⍴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 `⍴P` is equal to `(⍴M)[1]`, then `M[P;]` permutes the column vectors of `M` (i.e., interchanges the rows of `M`) and is called a column permutation. Similarly, if `⍴P` equals `(⍴M)[2]`, then `M[;P]` is a row permutation 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 positions 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.

Array output. Character arrays print with no spaces between components in each row; other arrays print with at least one space. 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 four ways in which the scalar functions of Table 3.2 extend to arrays: element-by-element, reduction, inner product, and outer product. Reduction 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 of them comprises only a single element. The requirements for inner product are shown in Table 3.6.

Scalar functions. All of the scalar functions of Table 3.2 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 induces a length or rank error report) unless one or 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 functions 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*[⍴*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 3.5) if it exists; if *X* is a scalar or a 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
Maximum	⌈	¯1.7014*E*38		L	R
Minimum	⌊	1.7014*E*38		L	R
Residue	|	0		L	
Out of	!	1		L	
Or	∨	0		L	R
And	∧	1		L	R
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 3.5: IDENTITY ELEMENTS OF PRIMITIVE SCALAR DYADIC FUNCTIONS

For a matrix *M*, reduction can proceed along the first coordinate (denoted by f⌿*M*) or along the second coordinate (f/*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).

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

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

then +⌿*M* is 5 7 9 and +/*M* is 6 15.

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`. (See the function **drop** 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 various cases are shown in Table 3.6.

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

Table 3.6: INNER PRODUCTS FOR PRIMITIVE SCALAR DYADIC FUNCTIONS f AND g

Outer product. The outer product of two vectors `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`.

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
```

Definitions for various cases are shown in Table 3.7.

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

Table 3.7: OUTER PRODUCTS FOR PRIMITIVE SCALAR DYADIC FUNCTION g

MIXED FUNCTIONS

Introduction. The **scalar** functions listed in Table 3.2 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 3.8, 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.

Monadic transpose. The expression `⍉A` yields the array `A` with the coordinates interchanged. For a vector `V` and a matrix `M`, the following relations hold:

`⍉V` is equivalent to `V`
`⍉M` is the ordinary matrix transpose, that is, the rows of `M` are the columns of `⍉M`, and vice versa.

Name	Sign[1]	Definition or example[2]
Size	ρA	ρP ↔ 4 ρE ↔ 3 4 ρ5 ↔ ι0
Reshape	VρA	Reshape A to dimension V 3 4ρι12 ↔ E 12ρE ↔ ι12 0ρE ↔ ι0
Ravel	,A	,A ↔ (×/ρA)ρA ,E ↔ι12 ρ,5 ↔ 1
Catenate	V,V	P,ι2 ↔ 2 3 5 7 1 2 'T','HIS' ↔ 'THIS' P[2] ↔3 P[4 3 2 1] ↔7 5 3 2
Index	V[A] M[A;A]	E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ ABCD EFGH IJKL
Index generator	ιS	First S integers ι4 ↔ 1 2 3 4 ι0 ↔ an empty vector
Index of	VιA	Least index of A in V, or 1+ρV Pι3 ↔2 5 1 2 5 PιE ↔ 3 5 4 5 5 5 5 5 4 4ι4 ↔ 1
Take† Drop†	V↑A V↓A	Take or drop |V[I] first (V[I]≥0) or last (V[I]<0) elements of coordinate I 2 3↑X ↔ ABC EFG ¯2↑P ↔ 5 7
Grade up † Grade down †	⍋A ⍒A	The permutation which would order A (ascending or descending) ⍋3 5 3 2 ↔ 4 1 3 2 ⍒3 5 3 2 ↔ 2 1 3 4
Compress	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 2 3 4 ↔ 1 0 1⌿E 9 10 11 12
Expand	V\A	1 0 1\ι2 ↔ 1 0 2 1 0 1 1 1\X ↔ A BCD E FGH I JKL
Reverse	⌽A	⌽X ↔ DCBA HGFE LKJI ⌽P ↔ 7 5 3 2 ⊖X ↔ IJKL EFGH ABCD
Rotate	A⌽A	3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P 1 0 ¯1⌽X ↔ BCDA EFGH LIJK
Transpose	V⍉A† ⍉A	Coordinate I of A becomes coordinate V[I] of result 2 1⍉X ↔ AEI BFJ CGK DHL 1 1⍉E ↔ 1 6 11 Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∊A	ρW∊Y ↔ ρW P∊ι4 ↔ 1 1 0 0 E∊P ↔ 0 1 1 0 1 0 1 0 0 0 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal †	S?S	W?Y ↔ Random deal of W elements from ιY

Table 3.8: PRIMITIVE MIXED FUNCTIONS (see adjacent notes)

†. This function not available on APL\1130

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

2. Arrays used in examples:

P ↔ 2 3 5 7		1	2	3	4		*ABCD*
	E ↔	5	6	7	8	*X* ↔	*EFGH*
		9	10	11	12		*IJKL*

Notes to Table 3.8

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

If *X* is a matrix, rotation of each row may be specified in the form *Z*←*K*⌽*X*, rotation of each column in the form *Z*←*K*⊖*X*. For example, for row rotation, if ⍴*X* is 3 4 , then *K* must be a scalar or a vector of dimension 3 and *Z*[*I*;] is equal to *K*[*I*]⌽*X*[*I*;]. For column rotation, ⍴*K* must be a scalar or a vector of dimension 4, and *Z*[;*I*] is equal to *K*[*I*]⌽*X*[;*I*]. For example:

M				0 1 2 3⊖*M*				1 2 3⌽*M*			
1	2	3	4	1	6	11	4	2	3	4	1
5	6	7	8	5	10	3	8	7	8	5	6
9	10	11	12	9	2	7	12	12	9	10	11

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*].

If *A* is a matrix, ⊖*A* is like *A* except that the order of elements is reversed along the columns; in ⌽*A*, the order is reversed along the rows. For example:

A			⊖*A*			⌽*A*		
1	2	3	4	5	6	3	2	1
4	5	6	1	2	3	6	5	4

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-element vector) 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 not extended. The result in every case is a vector.

If *M* is a matrix, then *U*⌿*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*/*M* denotes row compression. The result in every case is a matrix.

Expand. Expansion is the converse of compression and is denoted by *U**X*. If *Y*←*U**X*, then *U*/*Y* is equal to *X* and (if *X* is a vector of numbers) (~*U*)/*Y* is an array of zeros. In other words, *U**X* expands *X* to the form indicated by *U*, the elements of *X* assuming the positions of the ones in *U*, and zeros filling in elsewhere. To be conformable, +/*U* must equal ⍴*X*.

If *X* is a vector of characters, then spaces are supplied rather than zeros, i.e., if *Y*←*U**X* then (~*U*)/*Y* is an array of the space character ' '. Again, for matrices, *U**M* denotes expansion along the last coordinate, and *U*⍀*M* denotes expansion along the first. See Table 3.8 for examples of expansion.

A scalar left argument is not extended.

Decode. The expression *R*⊥*X* denotes the value of the vector *X* evaluated in a number system with radices *R*[1],*R*[2],...,*R*[⍴*R*]. For example, if *R*←24 60 60 and *X*←1 2 3 is a vector of elapsed time in hours, minutes, and seconds, then *R*⊥*X* has the value 3723, and is the corresponding elapsed time in seconds. Similarly, 10 10 10 10 ⊥ 1 7 7 6 is equal to 1776, and 2 2 2 ⊥ 1 0 1 is equal to 5. Formally, *R*⊥*X* is equal to +/*W*×*X*, where *W* is the weighting vector determined as follows: *W*[⍴*W*] is equal to 1, and *W*[*I*-1] is equal to *R*[*I*]×*W*[*I*]. For example, if *R* is 24 60 60, then *W* is 3600 60 1.

The result is a scalar.

The arguments *R* and *X* must be of the same dimension, except that either may be a scalar (or one-element vector). For example, 10 ⊥ 1 7 7 6 is equal to 1776. The arguments are not restricted to integer values. If *X* is a scalar, then *X*⊥*C* is the value of a polynomial in *X* with coefficients *C*, arranged in order of descending powers of *X*.

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*⊤*N* denotes the representation of the scalar *N* in the base-*R* number system. Thus, if *Z*←*R*⊤*N*, then (×/*R*)|*N*-*R*⊥*Z* is equal to zero. For example, 2 2 2 2 ⊤ 5 is 0 1 0 1 and 2 2 2 ⊤ 5 is 1 0 1 and 2 2 ⊤ 5 is 0 1. The dimension of *R*⊤*N* is the dimension of *R*. The encode function is also called representation.

Index of. If *V* is a vector and *S* is a scalar, then *J*←*V*⍳*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 1+⍴*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'⍳'GAFFE'
7  1  6  6  5
```

If *X* is a numerical vector, then the expression *X*⍳⌈/*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, then ⌈/*X* is 13 and *X*⍳⌈/*X* is 4.

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

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

If the vector *U* represents the universal set in some finite universe of discourse, then *U*∊*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 ∊ is determined by the size of the left argument, whereas the size of the result of the dyadic function ⍳ 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 ⍴*V*, then *S*↑*V* takes the first *S* components of *V*. For example, if *V*←⍳7, then 3↑*V* is 1 2 3 and 0↑*V* is ⍳0, and 8↑*V* yields a domain error.

If *S* is chosen from the set -⍳⍴*V*, then *S*↑*V* takes the last |*S* elements of *V*. For example, ¯3↑*V* is 5 6 7.

If *A* is an array, then *W*↑*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 *I*th coordinate of *A*. For example, if *A* ← 3 4⍴⍳12, then 2 ¯3↑*A* is the matrix

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

The function drop (↓) is defined analogously, except that the indicated number of elements are dropped rather than taken. For example, ¯1 1↓*A* is the same matrix as the 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. The take and drop functions are similar to the transpose in that the left argument concerns the dimension vector of the right argument.

Grade up and down.# The function ⍋*V* produces the permutation which would order *V*, that is *V*[⍋*V*] is in ascending order. For example, if *V* is the vector 7 1 16 5 3 9, then ⍋*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 ⍋ is formed by overstriking | and Δ.

If *P* is a permutation vector, then ⍋*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, ⍋5 3 7 3 9 2 is the vector 6 2 4 1 3 5.

These functions not available on APL\1130

The right argument of ⍋ may be any array *A* of rank greater than zero, and the coordinate *J* along which the grading is to be applied may be indicated by the usual notation ⍋[*J*]*A*. The form ⍋*A* applies as usual to the last coordinate. The result of ⍋*A* is of the same dimension as *A*.

The <u>grade</u> <u>down</u> function ⍒ is the same as the function ⍋ except that the grading is determined in descending order. Because of the treatment of duplicate items, the expression ∧/(⍋*V*)=⌽⍒*V* has the value 1 if and only if the elements of the vector *V* are all distinct.

<u>Deal</u>.# The function *M*?*N* produces a vector of dimension *M* obtained by making *M* random selections, without replacement, from the population ⍳*N*. In particular, *N*?*N* yields a random permutation of order *N*. Both arguments are limited to scalars or one-element arrays.

<u>Comments</u>.# The lamp symbol ⍝, formed by overstriking ∩ and ∘, signifies that what follows it is a comment, for illumination only and not to be executed; it may occur only as the first character in a statement, but may be used in defined functions.

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
```

sets *X* and *Y* to 1 and *Z* to 0.

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*.

This function not available on APL\1130

In the expression *Z*←(*A*+*B*)×(*C*+*D*) it is immaterial whether the left or the right argument of the × is evaluated first, and hence no order is specified. The principle of no specified order in such cases is also applied when the expressions include specification. Since the order here is sometimes material, there is no guarantee which of two or more possible results will be produced.

Suppose, for example, that *A* is assigned the value 5 and the expression *Z*←(*A*←3)×*A* is then executed. If the left argument of × is executed first, then *A* is assigned the value 3, the right argument then has the new value 3 and *Z* is finally assigned the value 9. If, on the other hand, the right argument is evaluated first it has the value 5 initially assigned to *A*, the value 3 is then assigned to *A* and multiplied by the 5 to yield a value of 15 to be assigned to *Z*.

Appendix A

SAMPLE TERMINAL SESSION

```
      )682
HILLS SIGNED ON

            A P L \ 1 1 3 0

                                  FUNDAMENTALS

      3×4                         Entry automatically indented
12                                Response not indented
      X←3×4                       X is assigned value of
                                       the expression
      X
12                                Value of X typed out
      Y←¯5                        Negative sign for negative
                                       constants
      X+Y
7
      144E¯2                      Exponential form of constant
1.44
      P←1 2 3 4                   Four-element vector
      P×P                         Functions apply element by element
1  4  9  16
      P×Y                         Scalar applies to all elements
¯5  ¯10  ¯15  ¯20
      Q←'CATS'                    Character constant (4-element
      Q                                vector)
CATS
      YZ←5                        Multi-character names
      YZ1←5
      YZ+YZ1
10
      3+4×5+6                     Correction by backspace
         v                             and linefeed
         +5+6
18
      X←3
      Y←4
      (X×Y)+4
16
      X×Y+4                       Executed from right to left
24
```



```
      X Y                         Entry of invalid expression
SYNTAX ERROR                      Shows type of error committed
      X Y                         Retypes invalid statement with
      ^                                caret where execution stopped
      XY                          Multi-character name (not X×Y)
VALUE ERROR
      XY                          XY had not been assigned a value
      ^
                                  SCALAR FUNCTIONS

      4×3⌈5.1                     Dyadic maximum
20.4
      (4×3)⌈5.1
12
      4×⌈5.1                      Monadic ceiling
24
      X←⍳5                        Index generator function
      X
1  2  3  4  5
      ⍳0                          Empty vector
                                       prints as a blank line
      Y←5-X                       All scalar functions extend
      Y                                to vectors
4  3  2  1  0
      X⌈Y
4  3  3  4  5
      X≤Y                         Relations produce
1  1  0  0  0                          logical (0 1) results
```

DEFINED FUNCTIONS

```
        ∇Z←X F Y                     Header (2 args and result)
[1]     Z←((X*2)+Y*2)*.5             Function body
[2]     ∇                            Close of definition
        3 F 4                        Execution of dyadic function F
5
        P←7
        Q←(P+1)F P-1                 Use of F with expressions
        Q                                 as arguments
10
        4×3 F 4
20
        ∇B←G A                       G is the signum function
[1]     B←(A>0)-A<0                  A and B are local variables
[2]     ∇
        G 4
1
        G ¯6
¯1
        X←¯6
        G X
¯1
        ∇H A                         Like G but has no explicit result
[1]     P←(A>0)-A<0                  P is a global variable
[2]     ∇
        H¯6
        P
¯1
        Y←H¯6                        H has no explicit result
VALUE ERROR                               and hence produces a value
        Y←H ¯6                            error when used to right
         ∧                                of assignment
        ∇Z←FAC N;I                   FAC is the factorial function
[1]     Z←1
[2]     I←0
[3]     L1:I←I+1                     L1 becomes 3 at close of defn
[4]     →0×⍳I>N                      Branch to 0 (out) or to next
[5]     Z←Z×I
[6]     →L1                          Branch to L1 (that is, 3)
[7]     ∇
        FAC 3
6
        FAC 5
120
        T∆FAC←3 5                    Set trace on lines 3 and 5 of FAC
        X←FAC 3
FAC[3] 1                             Trace of FAC
FAC[5] 1
FAC[3] 2
FAC[5] 2
FAC[3] 3
FAC[5] 6
FAC[3] 4
        T∆FAC←0                      Reset trace control
```


MECHANICS OF FUNCTION DEFINITION

```
        ∇G←M GCD N                   Greatest common divisor
[1]     G←N                               function based on the
[2]     M←M|N                             Euclidean algorithm
[3]     →4×M≠0
[4]     [1]G←M                       Correction of line 1
[2]     [4]N←G                       Resume with line 4
[5]     [1⎕]                         Display line 1
[1]     G←M
[1]     [⎕]                          Display entire GCD Function
      ∇ G←M GCD N
[1]     G←M
[2]     M←M|N
[3]     →4×M≠0
[4]     N←G
      ∇                              Part of display. Does not close
[5]     →1                           Enter line 5
[6]     ∇                            Close of definition
        36 GCD 44                    Use of GCD
4                                    4 is GCD of 36 and 44
        ∇GCD                         Reopen def (Use ∇ and name only)
[6]     [4.1]M,N                     Insert between 4 and 5
[4.2]   [⎕]                          Display entire function
      ∇ G←M GCD N
[1]     G←M
[2]     M←M|N
[3]     →4×M≠0
[4]     N←G
[4.1]   M,N                          Line number not changed until
[5]     →1                                close of defn
      ∇                              End of display
[6]     ∇                            Close of definition
        36 GCD 44
8  36                                Iterations printed by
4  8                                      line 5 (was line 4.1)
4                                    Final result
        ∇GCD[⎕]∇                     Reopen, display, and close GCD
      ∇ G←M GCD N
[1]     G←M
[2]     M←M|N
[3]     →4×M≠0
[4]     N←G
[5]     M,N                          Line numbers have been
[6]     →1                                reassigned as integers
      ∇                              Part of display
        ∇GCD                         Reopen definition of GCD
[7]     [5]                          Delete line 5 by linefeed
          ∧
[6]     ∇                            Close definition
```

```
      ∇Z←ABC X                    A function to show line editing
[1]   Z←(33×Q+(R×5)-6             A line to be corrected
[2]   [1⎕9]                       Initiate edit of line 1
[1]   Z←(33×Q+(R×5)-6             Types line, stops ball under 9
         /   1 /1                 Slash deletes, digit inserts spaces
      Z←(3×Q)+(T×5)-6             Ball stops at first new
[2]   ∇                                space. Then enter )  T
      FAC 5                       FAC still defined
120
      )ERASE FAC                  Erase function FAC
      FAC 5                       Function FAC no longer exists
SYNTAX ERROR
      FAC 5
      ∧
      ∇Z←BIN N                    An (erroneous) function for
[1]   LA:Z←(Z,0)+0,Z                   binomial coefficients
[2]   →LA×N≥⍴Z∇
      BIN 3
VALUE ERROR
BIN[1] LA:Z←(Z,0)+0,Z             Suspended execution
                    ∧
      Z←1                         Assign value to Z
      →1                          Resume execution
1 3 3 1                           Binomial coefficients of order 3
      BIN 4
VALUE ERROR                       Same error (local variable Z
BIN[1] LA:Z←(Z,0)+0,Z                  does not retain its value)
                    ∧
      )SI                         Display state indicator
BIN[1]   *                        Suspended on line 1 of BIN
      →0                          Clear state indicator
      ∇BIN[.1]Z←1∇                Insert line to initialize Z
      BIN 4                       Execute revised function
1 4 6 4 1
      ∇BIN[⎕]∇                    Display revised function
    ∇ Z←BIN N                          and close definition
[1]   Z←1
[2]   LA:Z←(Z,0)+0,Z
[3]   →LA×N≥⍴Z
    ∇
```


INPUT AND OUTPUT

```
      ∇MULT     N;Y;X             A multiplication drill
[1]   Y←?N                        ⍴N random integers
[2]   Y                           Print the random factors
[3]   X←⎕                         Keyboard input
[4]   →0×⍳X='S'                   Stop if entry is the letter S
[5]   →⍳X=×/Y                     Repeat if entry is correct product
[6]   'WRONG, TRY AGAIN'          Prints if preceding branch fails
[7]   →3∇                         Branch to 3 for retry
      MULT     12 12              Drill for pairs in range 1 to 12
2  10
⎕:                                Indicates that keyboard entry
      37                               is awaited
WRONG, TRY AGAIN
⎕:
      20
6  7
⎕:
      'S'                         Entry of letter S stops drill
      ∇Z←ENTER                    Example of character (⍞) input
[1]   Z←''                        Make Z an empty vector
[2]   D←⍴Z                        D is the length of Z
[3]   Z←Z,⍞                       Append character keyboard entry
[4]   →2×D≠⍴Z                     Branch to 2 if length increased
[5]   ∇                                (i.e., entry was not empty)
      Q←ENTER
THIS IS ALL                       Keyboard
 CHARACTER INPUT                       entries
                                  Empty input to terminate
      Q                           Display Q
THIS IS ALL CHARACTER INPUT
      N←5
      'NOTE: ⍳';N;' IS ';⍳N       Mixed output statement
NOTE:⍳5 IS 1  2  3  4  5
```

RECTANGULAR ARRAYS

```
      P←2 3 5 7
      ⍴P                          Dimension of P
4
      T←'OH MY'                   Character vector
      ⍴T
5
      P,P                         Catenation
2  3  5  7  2  3  5  7
      T,T
OH MYOH MY
```

```
      M←2 3⍴2 3 5 7 11 13          Reshape to produce a 2×3 matrix
      M                            Display of an array of rank >1
                                        is preceded by a blank line
2  3  5
7 11 13
      2 4⍴T                        A 2×4 matrix of characters

OH M
YOH
      6⍴M                          A matrix reshaped to a vector
2 3 5 7 11 13
      ,M                           Elements in row-major order
2 3 5 7 11 13
      P←,M
      P[3]                         Indexing (third element of P)
5
      P[1 3 5]                     A vector index
2 5 11
      P[⍳3]                        The first three elements of P
2 3 5
      P[⍴P]                        Last element of P
13
      M[1;2]                       Element in row 1, column 2 of M
3
      M[1;]                        Row 1 of M
2 3 5
      M[1 1;3 2]                   Rows 1 and 1, columns 3 2

 5 3
 5 3
      A←'ABCDEFGHIJKLMNOPQ'        The alphabet to Q
      A[M]                         A matrix index produces
                                        a matrix result
BCE
GKM
      A[M[1 1;3 2]]

EC
EC
      M[1;]←15 3 12                Respecifying the first row of M
      M

 15  3 12
  7 11 13
```



```
      Q←3 1 5 2 4 6                A permutation vector
      P[Q]                         Permutation of P
5 2 11 3 7 13
      Q[Q]                         A new permutation
5 3 4 1 2 6
```

FUNCTIONS ON ARRAYS

```
      V←?3⍴9                       Vector of 3 random integers (1-9)
      M←?3 3⍴9                     Random 3 by 3 matrix
      N←?3 3⍴9                     Random 3 by 3 matrix
      V
2 1 7
      M

 7 9 4
 5 8 1
 1 5 7
      N
 1 4 1
 4 7 6
 9 8 5
      M+N                          Sum (element-by-element)

  8 13  5
  9 15  7
 10 13 12
```

```
      M⌈N
 7  9  4
 5  8  6
 9  8  7
```
Maximum

```
      M≤N
0 0 0
0 0 1
1 1 0
```
Comparison

```
      +/V
10
```
Sum-reduction of V

```
      ×/V
14
```
Product-reduction

```
      +⌿M
13  22  12
```
Sum over first coordinate of M (down columns)

```
      +/M
20  14  13
```
Sum over second coordinate of M (over rows)

```
      ⌈/M
9  8  7
```
Maximum over last coordinate

```
      M+.×N
  79  123   81
  46   84   58
  84   95   66
```
Ordinary matrix (+.× inner) product

```
      M+.≤N
 1  1  1
 1  1  1
 2  3  2
```
An inner product

```
      M+.×V
51  25  56
```
+.× inner product with vector right argument


```
      V
2  1  7
      V∘.×⍳5
  2   4   6   8  10
  1   2   3   4   5
  7  14  21  28  35
```
Outer product (times)

```
      V∘.≤⍳9
  0   1   1   1   1   1   1   1   1
  1   1   1   1   1   1   1   1   1
  0   0   0   0   0   0   1   1   1
```
Outer product

MIXED FUNCTIONS

```
      Q←?10⍴5
      Q
1  4  3  4  5  4  2  1  4  2
```
A random 10 element vector (range 1 to 5)

```
      +⌿Q∘.=⍳5
2  2  1  4  1
```
Ith element of result is number of occurences of the value I in Q

```
      ⍉M
  7   5   1
  9   8   5
  4   1   7
```
Ordinary transpose of *M*

```
      Q
1  4  3  4  5  4  2  1  4  2
      3⌽Q                          Rotate to left by 3 places
4  5  4  2  1  4  2  1  4  3
      ¯3⌽Q                         Rotate to right by 3 places
1  4  2  1  4  3  4  5  4  2
      0 1 2⊖M                      Rotate columns by
                                        different amounts
   7     8     7
   5     5     4
   1     9     1
      ¯2⌽M                         Rotation of rows all
                                        by 2 to right
   9     4     7
   8     1     5
   5     7     1
      1 2 3⌽M                      Rotation of rows

   9     4     7
   1     5     8
   1     5     7
      ⌽Q                           Reversal of Q
2  4  1  2  4  5  4  3  4  1

      ⊖M                           Reversal of M along
                                        first coordinate
   1     5     7
   5     8     1
   7     9     4
      ⌽M                           Reversal along last coordinate

   4     9     7
   1     8     5
   7     5     1
```



```
      U←Q>4
      U
0  0  0  0  1  0  0  0  0  0
      U/Q                          Compression of Q by logical
5                                       vector U
      (~U)/Q                       Compression by not U
1  4  3  4  4  2  1  4  2
      +/U/Q
5
      1 0 1⌿M                      Compression along first
                                        coordinate of M
  7  9  4
  1  5  7
      1 0 1/M                      Compression along last
                                        coordinate
  7  4
  5  1
  1  7
      (,M>5)/,M                    ,M is  7 9 4 5 8 1 1 5 7
7  9  8  7                         All elements of M which exceed 5
      V←1 0 1 0 1
      V\⍳3                         Expansion of iota 3
1  0  2  0  3
      V\M                          Expansion of rows of M

  7  0  9  0  4
  5  0  8  0  1
  1  0  5  0  7
      V\'ABC'                      Expansion of literal vector
A B C                                   inserts spaces
      10⊥1 7 7 6                   Base 10 value of vector 1 7 7 6
1776
      8⊥1 7 7 6                    Base 8 value of 1 7 7 6
1022
      (4⍴10)⊤1776                  4 digit base 10 representation
1  7  7  6                              of number 1776
      (3⍴10)⊤1776                  3 digit base 10 representation
7  7  6                                 of 1776
      10 10⊤1776
7  6
      10⊤1776
6
      24 60 60⊥1 3 25              Mixed base value of 1 3 25
3805                                    (time radix)
      24 60 60⊤3805                Representation of number 3805
1  3  25                                in time radix
      2⊥1 0 1 1 0                  Base 2 value
22
```

```
      P
2  3  5  7  11  13
      P⍳7
4
      P⍳6
7
      P⍳4 5 6 7
7  3  7  4
      Q←5 1 3 2 4
      R←Q⍳⍳⍴Q
      R
2  4  3  5  1
      Q[R]
1  2  3  4  5
      A←'ABCDEFGHIJKLMNOPQ'
      A←A,'RSTUVWXYZ'
      A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      A⍳'C'
3
      J←A⍳'CAT'
      J
3  1  20
      A[J]
CAT
```

Index of 7 in vector P
7 is 4th element of P
6 does not occur in P, hence result is 1+⍴P

A permutation vector
R is the permutation inverse to Q

A is the alphabet

Rank of letter C in alphabet is 3


```
      M←3 5⍴'THREESHORTWORDS'
      M
THREE
SHORT
WORDS
      J←A⍳M
      J
 20   8  18   5   5
 19   8  15  18  20
 23  15  18   4  19
      A[J]
THREE
SHORT
WORDS
```

A matrix of characters

Ranking of M produces a matrix

Indexing by a matrix produces a matrix

```
      U←A∊'NOW IS THE TIME'
      '01'[1+U]
00001001100011100011001000
      U/A
EHIMNOSTW
      (⍳8)∊3 7 5
0  0  1  0  1  0  1  0
```

Membership

BIBLIOGRAPHY

Berry, P.C., _APL\1130 Primer_, IBM Corporation, 1968, Form No. C20-1697-0.

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.

Pakin, S., _APL\360 Reference Manual_, Science Research Associates, 1967.

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

Hellerman, H., _Digital Computer System Principles_, McGraw-Hill, 1967.