pdp11

APL-11 Programmer's Reference Manual

Order No. AA-5076B-TC

digital

January 1980

This document describes Version 2 of APL-11.

APL-11 Programmer's Reference Manual

Order No. AA-5076B-TC

SUPERSESSION/UPDATE INFORMATION:	This is a revision.
SOFTWARE VERSION TO OPERATING SYSTEMS AND VERSION:	APL-11 V2 • RT11 V4 • RSX-11M V3.2 • RSX-11M-PLUS V1.0 APL-11 V1 • RSTS/E V7

digital equipment corporation · maynard. massachusetts

Second Printing: January 1980

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

PREFACE

APL was first defined by K. E. Iverson in *A Programming Language* (Wiley, 1962) and has since been developed in collaboration with A. D. Falkoff and L. M. Breed. *APL* has been adapted into a conversational programming system and has been implemented on a variety of computers.

APL is a very concise programming language especially suitable for handling numeric and character array-structured data. Despite its mathematically concise and consistent format, *APL* is intended to be used as a general data-processing language as well as a mathematician's tool. The language is flexible enough to solve problems in text-handling and commercial data processing as concisely and as easily as it can be used to solve problems in numerical mathematics and statistics.

APL allows user-defined functions to be expressed with the same syntax as that used to express primitive language functions. This provides the user with an efficient and simple means of expanding the capabilities of the language to handle the requirements of any application area.

Areas of current application include scientific data reduction and analysis, simulation and forecasting, financial modeling, design engineering, electric circuit analysis, engineering analysis, inventory and payroll management, data base manipulation, reservation systems, automatic theorem proving, computer-assisted instruction (CAI), and student education (high school and college level) in programming and the structure of algorithmic processes. The applicability of *APL* as a complete conversational programming system is unlimited.

This manual presents an implementation of *APL*-11, a version of the *APL* language on the PDP-11 computer system. It should serve as a reference manual for all users of *APL*-11. This is not intended to be used as a language primer. The new *APL* user may want to refer to any of several good primers available for basic instruction in the language.

This manual is divided into six chapters and five appendixes. Information is structured as shown on the following page:

Chapter	Contents
1	*APL*-11 operating environment *APL* terminals and character set Starting, ending, and interrupting the *APL* session Keyboard editing procedures
2	*APL* language introduction and background information Primitive scalar and mixed functions
3	Defining, editing, and executing user-defined functions
4	*APL* system variables and I-beam functions
5	*APL* system commands
6	*APL*-11 file system

Appendixes A through D provide summary information on *APL* functions, system commands, I-beams and system variables, and error messages respectively. Appendix E describes procedures for installing *APL* in the RT-11, RSTS/E, RSX-11M, and IAS operating environments.

ACKNOWLEDGMENT

The *APL*-11 System, which runs under RSTS/E, RT-11, RSX-11M, or IAS, was originally developed by the Oregon Museum of Science and Industry.

CHAPTER 1

THE APL-11 OPERATING ENVIRONMENT

1.1 APL ON THE PDP-11

The *APL*-11 system has been implemented as a language interpreter on the PDP-11. It operates on a wide range of hardware processors and has been designed to run under any of four operating systems: RT-11, RSTS/E, RSX-11M, or IAS. This chapter introduces *APL*-11 and illustrates differences in initiating, terminating, and interrupting an *APL* session under these operating systems.

The *APL*-11 interpreter has been designed to be as flexible as possible to meet the needs of a variety of different users. Users can select from a range of system options. A customized version of the *APL*-11 system is distributed, reflecting the following installation-dependent characteristics:

- type of PDP-11 processor being used
- availability of floating-point hardware (FPP or FIS) on the PDP-11 processor
- operating system (RT-11, RSTS/E, RSX-11M, or IAS) under which *APL*-11 will run
- arithmetic precision desired (single-precision or double-precision)

If an installation's PDP-11 processor is not a PDP-11/70 or another processor that offers extended instructions (for example, the PDP-11/45 or a processor supporting EIS), the *APL* software simulates these extended instructions by generating a set of macros. Similarly, if the processor does not support the FPP hardware, the *APL* software simulates the capabilities of this hardware by assembling a software floating-point package with the *APL* interpreter.

The RT-11, RSTS/E, RSX-11M, and IAS operating systems provide *APL* users with most of the standard features of the PDP-11 single-user and time-sharing environments. *APL*-11 is configured for single-user access under RT-11 and RSTS/E, or as a task under RSX-11M and IAS; thus, the interpreter is heavily overlaid to increase the size of the user workspace.

A single-precision *APL*-11 system provides an accuracy of approximately seven digits, and a double-precision system offers an accuracy of about 16 digits. If a single-precision system is selected, floating-point numbers are four bytes long. If a double-precision system is selected, floating-point numbers are eight bytes long.

1.2 FILES IN THE APL SYSTEM

This section describes the special characteristics of workspaces and data files in the *APL*-11 environment.

1.2.1 APL Workspaces

An *APL* workspace is a part of the user's memory area that is used to store functions and variables defined by the *APL* user, and values and temporary results obtained while executing *APL* statements. The user's symbol table is stored in the workspace, along with the *state indicator*, an internal stack that may be accessed to determine the execution status of any defined function in the workspace. Workspaces can be cleared, named, erased, or saved on a secondary-storage device, and retrieved from that device at a later time.

When the user begins an *APL* session, a special workspace called the *clear workspace* is made available for his use. This workspace contains no defined functions or variables, has a clear symbol table and state indicator, and has no open files. The clear workspace has a variety of standard system values associated with it, including the following:

- index origin of 1
- output line length of 72 (the default terminal width is used for RSTS/E)
- six (single-precision) or ten (double-precision) significant digits
- comparison tolerance (fuzz) of *5E¯7* (single-precision) or 5*E¯15* (double-precision)

These values may be changed by the user during the current *APL* session. If a workspace is saved, the user-specified values are stored along with the workspace and will be in effect when the saved workspace is retrieved at a later time.

The workspace currently available to the user is known as the *active workspace*. All functions and variables defined during the current *APL* session are stored in this workspace. The active workspace may be stored as a file on a PDP-11 secondary-storage device, such as disk, floppy disk, DECtape, and magnetic tape. It may be saved in core-image format by the *)SAVE* system command (Section 5.2.3). The user may assign the active workspace a name by the *)WSID* system command (Section 5.2.2) and may override this name, if desired, when the workspace is saved.

Once an *APL* workspace has been saved on secondary storage, it may be deleted by the *)DROP* system command (Section 5.2.6) or retrieved to function as the active workspace once again. If the file has been saved, it must be retrieved by the *)LOAD* system command (Section 5.2.4). Data from the workspace may be copied into the current workspace by the *)COPY* and *PCOPY* system commands (Sections 5.3.6 and 5.3.7). The maximum size of an *APL* workspace depends upon the operating system and the amount of memory in the system.

1.2.2 APL Data Files

In the *APL*-11 system, data files may be stored on a variety of devices, including disk, floppy disk, DECtape, and magnetic tape. Two types of data files are supported by *APL*-11:

- ASCII sequential
- random access

ASCII sequential files are line-oriented sequential files that may be read and written by *APL*, by the MACRO Assembler, and by a variety of other language processors. ASCII sequential files may also be created and modified by RT-11, RSTS/E, RSX-11M, and IAS text editors.

Random-access files may be read and written in a non-sequential fashion. When accessing the file, the user identifies the particular byte or data value to be read or written and specifies the format of that value. Data may be specified as ASCII, byte, integer, *APL* character, single-precision floating-point, or double-precision floating-point quantities. Random-access mode allows the user to construct records containing values of several different data types. It also facilitates the use of random-access data files created by other language processors or systems.

The file operators and system commands implemented as part of the *APL*-11 file system are described in detail in Chapter 6.

1.3 APL HARDWARE

The user interacts with *APL* by means of a typewriter-like terminal. The *APL* language supports the use of a special character set, in which Greek letters and a variety of other special characters represent *APL* language operators. Examples of such special characters include ρ, ι, ⎕, ∇, and ϵ. Several terminals available to *APL*-11 users provide keyboards on which the full *APL* character set may be utilized. Standard ASCII terminals may also be used with *APL*. On ASCII terminals, the special *APL* symbols are represented by means of keyword mnemonics, described in Section 1.4. The user selects the *APL* or ASCII character set at the time that he begins the current *APL* session (see Section 1.5).

Table 1-1 lists the terminals supported by the PDP-11 computer system for use with *APL*-11. The second column of this table indicates whether or not the special *APL* character set is represented on the terminal keyboard. The third column lists the terminal designator that must be entered at the time that the *APL* session begins (see Section 1.5).

Table 1-1
APL Terminals

Terminal	Character Set	Designator
Any standard ASCII terminal without the *APL* character set	ASCII	TT
DECwriter II model LA37 with *APL* option	*APL*/ASCII	LA37
Tektronix Ⓡ 4013 or 4015 terminal	*APL*/ASCII	4013
Ⓡ Tektronix is a registered trademark of Tektronix, Inc.		

1.3.1 APL Terminals

The terminal keyboard illustrated below may be used in either ASCII or *APL* mode. It supports the full *APL* character set: all characters on this keyboard are received and interpreted by *APL*. Note that letters, numbers, and some of the special characters appear in the conventional keyboard positions. The letters print only in upper-case and are produced only when the keyboard is not shifted. The full *APL* character set is described in Table 1-2, included in Section 1.4.

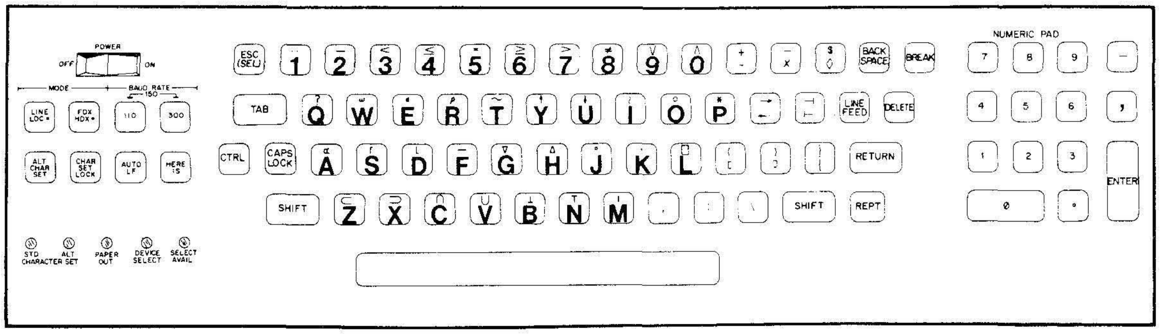

Figure 1-1 The APL Keyboard (LA37 Terminal)

1.3.2 ASCII Terminals

ASCII terminals do not support the use of the special *APL* character set illustrated in the *APL* keyboard shown in Section 1.3.1. If the user has an ASCII terminal or is operating in ASCII mode on an *APL* terminal, he must use keyboard mnemonics in place of the special *APL* symbols not available in ASCII. To represent the *APL* rho symbol (ρ), for example, the user enters the mnemonic .RO. The .GO mnemonic is equivalent to the *APL* right-arrow (→), and the .EP mnemonic is equivalent to the *APL* epsilon (ε). A summary of the mnemonic equivalents for all *APL* characters is provided in Table 1-2.

If the user has an ASCII terminal, but erroneously selects the *APL* character set by specifying an incorrect terminal designator, he can terminate the *APL* session by typing the *)OFF* system command, replacing the left parenthesis with a double quote character, as in the following:

```
    "off
```

Note that lower-case characters must be used.

NOTE

Because the keyword mnemonics are characterized by the presence of a period (.) as the first character, the period should not be used to separate the workspace filename and extension name in ASCII mode. The comma (,) should be used instead.

1.4 THE APL CHARACTER SET

Table 1-2 summarizes all characters used in the *APL* system. The first column lists the characters found on *APL* terminal keyboards. The second column provides a list of the corresponding characters available on ASCII terminals. The third column lists keyword mnemonics used to represent *APL* symbols not available on the ASCII keyboard. The fourth column supplies the names commonly associated with the *APL* characters, with upper-case letters indicating the origin of the mnemonic representation.

The second section of the table lists *APL* overstruck characters. These are characters constructed by overstriking two distinct characters on the terminal keyboard. For example, the logarithm symbol (⍟) is formed by overstriking the circle (○) with the exponentiation symbol (⋆). The grade up symbol (⍋) is formed by overstriking the delta symbol (Δ) with the straight line (|) used to represent residue or absolute value. To express an overstruck character on an *APL* terminal, the user types one character of the overstrike combination, then presses the backspace key, then types the other character of the combination. The order is not significant. On ASCII terminals, all overstruck characters are represented by alternate single-strike characters or by keyword mnemonics.

Table 1-2
APL Character Set

Single-Strike Characters			
APL Set	ASCII Set	Mnemonic	Name
+	+		add
A-Z	A-Z		alphabetics
∧	&	.AN	ANd
←	_		assignment
,	,		concatenate, comma
:	:		colon
.	.		decimal point
÷	%	.DV	DiVide
=	=		equal to
\	\		expand
*	*		exponentiate
>	>	.GT	Greater Than
[	[		left bracket
(	(		left parenthesis
<	<	.LT	Less Than
×	#		multiply
0-9	0-9		numerics
'	'		quote string
?	?		question (roll and deal)
/	/		reduce
]	]		right bracket
)	)		right parenthesis
;	;		semicolon
-	-		subtract
↑	^	.TK	TaKe
_		.US	UnderScore
|		.AB	residue (ABsolute value)
α		.AL	ALpha
⎕		.BX	quad (BoX)
⌈		.CE	CEiling (maximum)

Table 1-2 (Cont.)
APL Character Set

Single-Strike Characters			
APL Set	ASCII Set	Mnemonic	Name
↓		.DA	drop (Down Arrow)
¨		.DD	Dieresis
⊥		.DE	DEcode
∇		.DL	DeL
◊		.DM	DiaMond
∩		.DU	Down Union
⊤		.EN	ENcode
∊		.EP	EPsilon
⌊		.FL	FLoor
≥		.GE	Greater than or Equal
→		.GO	GO to (branch)
⍳		.IO	IOta
{		.LB	Left Brace
∆		.LD	delta (Lower Del)
≤		.LE	Less than or Equal
⊢		.LK	Left tacK
○		.LO	circle (Large O)
⊃		.LU	Left Union
≠		.NE	Not Equal to
¯		.NG	NeGation
~		.NT	NoT
⍵		.OM	OMega
∨		.OR	OR
}		.RB	Right Brace
⊣		.RK	Right tacK
⍴		.RO	RhO
⊂		.RU	Right Union
∘		.SO	jot (Small o)
∪		.UU	Up Union

Overstruck Characters			
APL Set	ASCII Set	Mnemonic	Name
⍝	"		lamp
!	!		factorial
$	$		Dollar Sign
⍒		.GD	Grade Down
⍋		.GU	Grade Up
⌶		.IB	I-Beam
⍟		.LG	LoGarithm
⍲		.NN	NaNd
⍱		.NR	NoR
⍀		.CB	Column expansion
⊖		.CR	Column Rotate
⌿		.CS	Column reduction
⌹		.DQ	Divide Quad
⍇		.IQ	Input Quad
⍈		.OQ	Output Quad
Ʊ		.OU	OUt
⍫		.PD	Protected Del
⍐		.PT	set file PoinTer
⍔		.QD	Quad Del
⍞		.QQ	Quote Quad
⊆		.SS	Subset
⊇		.CO	Contains

Table 1-2 (Cont.)
APL Character Set

Overstruck Characters			
APL Set	ASCII Set	Mnemonics	Name
⌽		.RV	ReVersal
⍉		.TR	TRanspose
⍎		.XQ	eXecute
⍕		.FM	ForMat
A̲-Z̲		.ZA-.ZZ	underscored alphabetics
⍙		.Z@	underscored del

1.5 INTERACTING WITH APL

This section describes how the *APL*-11 user establishes communication with *APL* and concludes or interrupts an *APL* session. It provides separate descriptions of operating procedures in the RT-11, RSTS/E, RSX-11M, and IAS operating systems.

1.5.1 RT-11 Operating Procedures

APL-11 runs as a single-user program in the RT-11 operating system. An example of invoking *APL*, performing a function, and ending an *APL* session is included at the end of this section.

To initiate interaction with *APL*, the user first establishes communication with RT-11. The system displays the period prompt character, and the user enters the following command:

```
.R APL
```

and presses the RETURN key. The RETURN key must be pressed at the conclusion of any monitor or *APL* statement to cause that statement to be transmitted. *APL* begins the session by asking about the user's terminal type:

```
TERMINAL..
```

and waiting for a response. If the user types *H* (for *HELP*), *APL* displays a description of the terminals currently supported by the system, and repeats the *"TERMINAL.."* prompt - for example:

```
RUN $APL
TERMINAL..H
GIVE THE APPROPRIATE RESPONSE FOR YOUR TERMINAL
RESPONSE  YOUR TERMINAL
LA36      LA36 WITH APL CHARACTER SET OPTION
4013      TEKTRONIX 4013
TT        ANY TERMINAL NOT HAVING APL FONT
TERMINAL..
```

The user selects the designator that is appropriate to his terminal type, and enters it as shown below:

```
TERMINAL..  TT
```

After receiving a valid terminal designator, *APL* responds with a sign-on greeting -- for example:

```
    WELCOME TO APL/11
```

It then supplies a clear workspace for use during the current *APL* session and displays the message:

```
    CLEAR WS
```

The system indents six spaces to indicate that it is ready to accept user input. *APL* output results at the left margin, but automatically indents six spaces before unlocking the keyboard and allowing any user text to be entered. The system thus clearly differentiates between system and user entries.

Under RT-11, an *APL* session may be concluded by means of either of the system commands shown below.

Command	Effect
)OFF	Ends the session, exits from *APL*, and returns to RT-11 command level.
)RUN filename	Ends the session, exits from *APL*, and runs the program specified as an argument in the *)RUN* command.

These system commands are described in greater detail in Sections 5.5.1 and 5.5.2. ɪ-beam 36 (Section 4.3.15) may also be used to terminate the *APL* session and return to RT-11 command level. With all of these commands and functions, *APL* automatically closes all open files before exiting from *APL*.

The currently active workspace will not be preserved if *)OFF*, *)RUN*, or ɪ-beam 36 is issued. If the user wants to save this workspace before terminating the *APL* session, he should store it on disk or on another secondary-storage device by issuing a *)SAVE* (Section 5.2.3) system command.

The user may interrupt *APL* without actually terminating the session and losing the active workspace. The following control characters are used in the RT-11 system to interrupt *APL*.

Character(s)	Circumstances	Effect
CTRL/C	*APL* is awaiting input	Echoes a ↑C character and stops execution, displaying "EXECUTION STOP". Indents six spaces and awaits new *APL* input.
CTRL/C, CTRL/C	*APL* is executing a function, evaluating an expression, or performing output.	Echoes two ↑C characters, aborts output, and stops execution, displaying "EXECUTION STOP" and the expression being evaluated. Indents six spaces and awaits new *APL* input.

Character(s)	Circumstances	Effect
CTRL/O	*APL* is performing output on the terminal.	Inhibits output until completion of current output or until another CTRL/O is typed. The first CTRL/O echoes a ↑O character, the second CTRL/O reenables output.

NOTE

> On an *APL* terminal, CTRL/C echoes as ≥∩ and CTRL/O echoes as ≥∘.

In the following example, user responses are underlined.

```
    RUN APL
    TERMINAL..I

    WELCOME TO APL/11 V1.0
    CLEAR WS
            .DL LOOPER
    [1]     LOOP: A_A+1
    [2]     .GO LOOP
    [3]     .DL
            A_1
            LOOPER
    ^C

    .RE

    RESTARTING APL
    LOOPER[1] A_A+1
              ^


            A
    973
            )OFF
```

If three or more CTRL/C characters cause control to be returned to RT-11 command level, the user may reenter *APL* without beginning a new session by issuing the REENTER monitor command. The current active workspace is preserved, so the symbol table and all defined functions and variables remain intact. In general, the user should close all files before interrupting the *APL* session in this way.

When *APL* is reentered, a restart message is displayed, and *APL* identifies the command or function line that was being evaluated when execution was suspended. An up-arrow (↑) or caret (∧) identifies the particular position in the line at which evaluation was interrupted, exactly as shown for the "EXECUTION STOP" case illustrated above.

CAUTION

> If the user issues any command that runs another program (e.g., PIP) while at RT-11 command level, the current workspace will be lost.

1.5.2 RSTS/E Operating Procedures

To initiate interaction with *APL*, the user first establishes communication with RSTS/E, entering his project-programmer number and password in the normal way. The system displays a sign-on greeting and enters the standard BASIC language environment. BASIC displays a "Ready" message to indicate that it is ready to accept input. The user enters the following command:

```
    run $apl
```

and presses the RETURN key. The RETURN key must be pressed at the conclusion of any monitor or *APL* statement to cause that statement to be transmitted. If the RSTS/E installation has established the CCL command, *APL*, the user simply types:

```
    apl
```

and presses the RETURN key. In either case, *APL* begins the session by asking about the user's terminal type:

```
    TERMINAL..
```

and waiting for a response. The user specifies the terminal being used during the current *APL* session, as illustrated in Section 1.5.1.

After receiving a valid terminal designator, *APL* responds with a sign-on greeting. As in the RT-11 environment, it supplies a clear workspace for use during the current *APL* session and displays the message:

```
    CLEAR WS
```

The system indents six spaces to indicate that it is ready to accept user input.

Under RSTS/E, an *APL* session may be concluded by means of one of the system commands shown below.

Command	Effect
)OFF	Ends the session, exits from *APL*, and returns to BASIC; BASIC displays the "Ready" message.
)RUN filename	Ends the session, exits from *APL*, and runs the program specified as an argument in the *)RUN* command.

ɪ-beam 36 may also be used to terminate the *APL* session and return to BASIC. With all of these commands and functions, *APL* automatically closes all open files before exiting from *APL*.

As in the RT-11 environment, *APL* does not automatically preserve the currently active workspace when *)OFF*, *)RUN*, or ɪ-beam 36 is issued.

If the user wants to save this workspace before terminating the *APL* session, he should store it on disk or on another secondary storage device by issuing a *)SAVE* system command.

In the RSTS/E operating system, the user may interrupt *APL* execution without actually exiting from *APL*. The following control characters are used to interrupt the *APL* session in the RSTS/E system.

Character(s)	Circumstances	Effect
CTRL/C	*APL* is executing a function, evaluating an expression, awaiting input, or performing output.	Echoes a ↑C character, displays an *EXECUTION STOP* message and the expression being evaluated, indents six spaces, and awaits new *APL* input.
CTRL/O	*APL* is performing output.	Inhibits output until completion of current output or until another CTRL/O is typed. The first CTRL/O echoes a ↑O character; the second CTRL/O reenables output.

NOTE

On an *APL* terminal, CTRL/C echoes as ≥∩ and CTRL/O echoes as ≥∘.

When *APL* is interrupted, it identifies the command or function line that was being evaluated when execution was suspended. An up-arrow (↑) or caret (∧) identifies the particular position in the line at which execution was interrupted, as illustrated in the following example. In this example, user responses are underlined. Note also that, because the terminal used in this example was an *APL* terminal, CTRL/C echoed as ≥∩, the equivalent of ↑C in the *APL* type face.

1.5.3 RSX-11M Operating Procedures

APL-11 runs as a task under the RSX-11M operating system. To initiate interaction with *APL*, the user first establishes communication with the RSX-11M Monitor Console Routine (MCR). MCR displays the angle bracket prompt character (>), and the user enters the following command:

```
>APL
```

and presses the RETURN key. The RETURN key must be pressed at the conclusion of any monitor or *APL* command to cause that statement to be transmitted. As with RT-11 and RSTS/E, *APL* begins the session by asking about the user's terminal type and awaiting a response (see Section 1.5.1 for examples of valid responses). *APL* then displays a sign-on greeting and supplies a clear workspace for use during the current *APL* session. This is shown in the example below.

```
>APL
TERMINAL..TT

WELCOME TO APL-11 V02-01
CLEAR WS
```

The system indents six spaces to indicate that it is ready to accept user input.

Under RSX-11M, an *APL* session may be concluded by means of the *)OFF* system command described in Section 5.5.1 I-beam *36* may also be used to terminate the *APL* session and return control to the Monitor Console Routine. If the user wants to save the currently active workspace before issuing *)OFF* or I-beam *36*, he should use the *)SAVE* system command.

1.5.4 IAS Operating Procedures

APL-11 runs as a task under the IAS operating system. To initiate interaction with *APL*, the user first establishes communication with the IAS Program Development System (PDS). PDS displays the PDS> prompt, and the user enters the following command:

```
PDS>APL
```

and presses the RETURN key. The RETURN key must be pressed at the conclusion of any monitor or *APL* command to cause that statement to be transmitted. As with the other operating systems, *APL* begins the session by asking about the users' terminal type and awaiting a response (see Section 1.5.1 for examples of valid responses). *APL* then displays a sign-on greeting and supplies a clear workspace for use during the current *APL* session. This is shown in the example below.

```
PDS>APL
TERMINAL..TT

WELCOME TO APL-11 V02-01
CLEAR WS
```

The system indents six spaces to indicate that it is ready to accept user input.

Under IAS, an *APL* session may be concluded by means of the *)OFF* system command, described in Section 5.5.1 I-beam *36* may also be used to terminate the *APL* session and return control to the Program Development System. If the user wants to save the currently active workspace before issuing *)OFF* or I-beam *36*, he should use the *)SAVE* system command.

1.6 KEYBOARD EDITING PROCEDURES

This section summarizes the procedures for entering and correcting *APL* text at an *APL* or ASCII terminal.

1.6.1 Immediate-Mode Editing Procedures

The characters in an *APL* input line may be typed in any order. The line may even be typed backwards by using the appropriate space and backspace characters. Regardless of how the line is entered, it is evaluated exactly as it appears on the terminal before the user presses the RETURN key; the order in which statement components are entered is not significant. If there are too many characters in the line, *APL* will display the following message:

LINE TOO LONG

The entire line will be ignored.

If the user has left spaces in the *APL* line, he may backspace to insert characters before he presses the RETURN key. Note that backspacing is a method for positioning the carriage; it does not cause characters to be erased or deleted.

The user may discover that he has mistyped one or more characters in an *APL* statement before he presses the RETURN key and causes that statement to be transmitted. Errors may be corrected by means of the special keyboard characters described below. If a new line is entered by the user immediately after a CTRL/C, CTRL/U, or RUBOUT character has been processed, the line will not be indented six spaces in the normal *APL* fashion, but will begin at the left margin.

Character	Meaning
RETURN	Terminates the input line and causes the *APL* statement to be transmitted.
CTRL/C	Interrupts *APL* function execution, expression evaluation, input, and output. Echoes ↑C on the terminal. In RT-11, returns control to command level; two CTRL/C characters must be entered if *APL* is performing input or output. In RSX-11M and IAS interrupts, *APL* execution but does not return control. On an *APL* terminal, echoes as ≥∩.

Character	Meaning
CTRL/U	Deletes the current input line and echoes ↑U on the terminal. Does not delete characters past the first carriage return/line feed combination encountered to the left of the CTRL/U. CTRL/U cannot be used to delete a multiple-line literal (see the description of the overstruck OU below). On an *APL* terminal, echoes as ≥↓.
RUBOUT	Deletes the last character from the current line and echoes the character on most terminals. Each succeeding RUBOUT typed by the user deletes and echoes another character up to the first carriage return/line feed combination encountered to the left of the RUBOUT. The DELETE key is used in place of RUBOUT on some *APL* terminals.
U (overstruck *OU*)	Causes an escape from an input loop or expression. Entered as mnemonic .OU on ASCII terminals. The *OU* sequence may be used to escape from a loop containing a quad or quote-quad input request. It may also be used to delete a multiple-line literal, which has been created by placing an odd number of quotes (usually one) on a line. When this occurs, subsequent lines are considered part of the literal until another line with an odd number of quotes is typed - for example:

```
      A←'THIS
IS A MULTIPLE
LINE LITERAL'
      A
THIS
IS A MULTIPLE
LINE LITERAL
```

If *APL* does not respond to an input line, the problem may be that the user has accidentally entered a multiple-line literal. To escape, the user should type a single quote on a line to terminate the literal or should use the *OU* sequence to cancel the literal.

Character	Meaning
CTRL/R	Retypes the current input line. This is often helpful in cases in which extensive editing has been performed on a line. CTRL/R does not alter the input line and may be used any number of times. On an *APL* terminal, echoes as ≥ .

The example included below illustrates the use of many of the special immediate-mode editing characters described above.

```
      A←÷42≥↓

      A←1÷4⊢4÷⊢+4++/⍳5
      A
20
```

1.6.2 Processing Character Errors

If the user transmits a line containing an invalid character - for example, an illegal overstrike in *APL* mode or an illegal mnemonic in ASCII mode - *APL* generates the following error message:

CHARACTER ERROR

The line in which the error occurred is ignored, and *APL* indents to allow the user to retype it.

CHAPTER 2

THE APL LANGUAGE

This chapter describes an implementation of the *APL* language and provides a detailed discussion of the features supported by *APL*-11.

2.1 OVERVIEW OF APL STATEMENTS

This section introduces the syntax rules that govern the construction of statements in the *APL* language. It summarizes statement components and discusses the types of *APL* statements and the evaluation of *APL* expressions.

2.1.1 Statement Execution Modes

APL statements may be executed in either of two modes:

- Immediate mode, in which statements and expressions are executed immediately, as entered by the user.

- Function-definition mode, in which the user may construct a program or function consisting of *APL* statements, may name and save the function, and may execute the function at a future time.

The syntax of the language itself is identical in both modes; however, a few special symbols have been defined for ease of editing in function-definition mode, and these are not generally relevant to immediate-mode execution. Most of the examples in this chapter illustrate immediate-mode execution of individual *APL* statements. Chapter 3 describes the preparation and editing of programs in function-definition mode and introduces the special *APL* symbols used in that mode.

In immediate mode interactions, *APL* clearly differentiates between system and user entries. When the user begins an *APL* session, the carriage automatically indents six spaces before allowing any text to be entered. The user enters a statement and presses the carriage return to indicate that the entry is complete. *APL* processes the statement and displays the result at the left margin of the next line. The system then begins a new line and automatically indents the customary six spaces before unlocking the keyboard for the user's next statement. System and user entries can thus be distinguished, as shown in the following interaction.

```
System   User

         A←2+3
         A
  5
         2+1
  3
         3+5
  8
         A-3
  2
         A←A+5
         A
  10
```

2.1.2 Statement Components

An *APL* statement may consist of the following components:

- identifiers (variables, label names, user-defined function names)
- constants
- symbols for *APL* primitive functions

The following subsections summarize the characteristics of *APL* identifiers, constants, and data structures. *APL* primitive functions and operators are described in Sections 2.6 through 2.8. Labels and user-defined functions are discussed in Chapter 3.

2.1.2.1 Identifiers - *APL identifiers* are used to name variables, user-defined functions, and labels within functions. An identifier may consist of any number of letters or digits; the first character of the sequence must be a letter, where a letter is defined as any character *A-Z*, *A̲-Z̲*, Δ or Δ̲. Only the first 31 characters of an identifier are significant, and embedded spaces are not allowed.

A variable must be assigned a value before it can be referenced, or a *VALUE ERROR* results. A discussion of specific types of *APL* variables and detailed information about function and label name construction are included in Chapter 3.

All *APL identifiers* are stored in the symbol table. The symbol table consists of 508 bytes in the clear workspace, and it expands dynamically, as needed, to the capacity of the workspace. Each identifier entry requires seven bytes, plus one byte for each character of the identifier, plus an optional fill byte, to bring the total to an even number of bytes.

2.1.2.2 Numeric Constants - *Numeric constants* are of two types: decimal and exponential. The decimal form may be entered with or without a decimal point. The exponential form consists of an integer or decimal quantity, followed by *E* and the power of ten by which the quantity is to be multiplied. All of the following numeric constants are valid representations of the same value.

```
        556 556.0 5560E¯1 5.56E2 55600E¯2
  556 556 556 556 556
```

When *APL* outputs a sequence of numeric constants, the system attempts to display the entire list in decimal form, as shown in the example above.

In *APL*, *negative numbers* are represented by a numeric constant, immediately preceded by a negative sign (¯). This sign is a distinct symbol (upper-case 2) and can be used only in negative numeric constants; it is not the same character as the minus sign (-) used to indicate a negative or minus function. On ASCII terminals, the negative sign is typed as .*NG*. Note that a space may not be included between the negative sign (¯ or .*NG*) and the number. It is displayed as the minus sign (-) on ASCII terminals, except in displays of user functions. Examples of using negative numbers and negative and minus functions are included below.

```
        ¯1
  ¯1
        2-3
  ¯1
        ¯2-¯3
  1
        ¯5E¯6-¯2E¯6
  ¯3E¯6
```

2.1.2.3 Data Structures - Numeric and character data can be structured in a variety of ways. The following data structures are supported by *APL*:

- scalars
- vectors
- matrices
- arrays of three or more dimensions

A *scalar* is a single numeric or character value. A numeric scalar is entered as shown in the first example below. Note in the second example that a character scalar must be enclosed in single quotes.

```
        A←5
        A
  5
        B←'C'
        B
  C
```

A *vector* is a 1-dimensional array or string consisting of any number of values. A numeric vector is entered as a list of values separated by at least one space - for example:

```
        A←1 2 3 4
        A
  1 2 3 4
```

Here *A* is defined as a vector whose elements are 1, 2, 3, and 4, stored in the order in which they were entered. Several other numeric vectors are created below.

```
      ¯1 2 3
¯1 2 3
      -1 2 3
¯1 ¯2 ¯3
```

Note that the first example generates a vector whose first element is ¯1; the second example applies the monadic negative operator (-) to the positive numeric vector 1 2 3.

A character or literal vector is entered as a string of character constants enclosed in single quotes; no spaces are inserted between entries in a character vector, because the space character is itself a legitimate literal value. An example of entering and examining a character vector is shown below.

```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

A single quote character may be represented in a character vector by means of two consecutive single quotes - for example:

```
      NAME←'MARTHA''S'
      NAME
MARTHA'S
      ''''''''
'''
```

Several lines of character data may be entered as one literal string, as shown below.

```
      A←'THIS IS A
MULTIPLE LINE
LITERAL'
      A
THIS IS A
MULTIPLE LINE
LITERAL
```

A *matrix* is a 2-dimensional array consisting of rows and columns. The user must enter values corresponding to each element of an array, but must also specify the shape of the array. The shape of an array is the number of dimensions which it has and the length of each of these dimensions. For example, a matrix may have six elements arranged as two rows and three columns, or three rows and two columns, as illustrated by arrays *A* and *B* below.

```
      A
1 2 3
4 5 6


      B
1 2
3 4
5 6
```

The primitive rho (ρ) function is used to specify the shape of a new array, to reshape an existing array, or to determine the shape of an existing array; it is described in detail in Sections 2.7.3 and 2.7.4. Following is an example of creating a simple matrix with the rho function.

```
        A←4 2⍴0 1 2 3 4 5 6 7
        A
0 1
2 3
4 5
6 7
```

Arrays of three dimensions and more are also supported by *APL*. An *APL* array may have as many as 16 dimensions; the only restriction is that the size of the array must not exceed the size of the user's workspace. When an array of more than two dimensions is displayed, a blank line is inserted between each dimension, as in the following example.

```
        2 2 2 4⍴'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEF'
ABCD
EFGH

IJKL
MNOP


QRST
UVWX

YZAB
CDEF
```

2.1.3 Significance of Spaces and Comments

Spaces are usually not significant in *APL*. They need not be included to separate primitive functions from constants or variables but they may be used in such statements if desired. In particular, on ASCII terminals the mnemonics for *APL* primitive functions need not be either preceded or followed by a space. The following pairs of expressions are equivalent.

```
A←B+1-C
A ← B + 1 - C


.TRB
.TR B
```

Spaces are also not required between a succession of primitive functions - for example:

```
A←+/B
```

Spaces must be included to separate the names of adjacent user-defined functions, constants, and variables. For example, they are required when entering a series of numeric constants as a vector. The spaces included in the following statements are necessary.

```
    2 TRIG 3
    B←3 4 5
    X←F 12
```

Spaces may not be included between a negative sign (¯ or *.NG*) and a numeric constant (Section 2.1.2.2).

Comments may be used freely in *APL*. Their use is particularly relevant in function-definition mode. Comments must appear on separate lines and may not be included on lines containing *APL* statements. The first character in a comment line must be a lamp (⍝) character, formed by overstriking the down union (∩) and jot (∘) characters. If an ASCll terminal is being used, the first character in a comment line must be a double-quote ("). Chapter 3 describes comment lines in greater detail (see Section 3.2.4) and illustrates their use in a variety of user-defined functions.

2.1.4 APL Statement Types

There are two general types of *APL* statements.

- branch statements
- assignment statements

Branch statements are used to restart a function and to transfer control from one part of a program to another. These statements are most relevant in the context of user-defined functions and are described in Chapter 3.

Assignment statements are used to assign one or more values to a variable or data structure. The general form of an assignment statement is illustrated in the following example:

```
    A←2+3
```

where the constant 3 is added to the constant 2 and the resulting value, 5, assigned to variable *A*. There may be multiple assignments or specifications in a single *APL* statement - for example:

```
    A←3+B←4+C←7
```

Here the value 7 is assigned to *C*, 11 to *B*, and 14 to *A*. The expression is evaluated according to the rule described in Section 2.1.5.

Multiple specifications are particularly useful in initializing data values, as illustrated below:

```
    A←B←C←D←0
```

The expression:

```
        2+3
    5
```

may also be considered an assignment statement; in this case, no explicit variable is available to receive the result, so the value of the computation is simply assigned to the terminal.

The result of an *APL* expression is displayed at the terminal, unless the leftmost operation on the line is an assignment or branch operation or unless the user is in function-definition mode.

2.1.5 Evaluation of APL Statements and Expressions

Unlike some languages, which perform multiplication and division before addition and subtraction, *APL* has no explicit *operator precedence*. *APL* statements and expressions are evaluated in strict right-to-left order, regardless of the particular functions in the statement. For example, the expression:

```
      3×4+5
27
```

evaluates to 27, using right-to-left evaluation, rather than 17, which would be the result if operator precedence were employed. All *APL* statements are executed as if they were parenthesized from right-to-left. Thus, the expression:

```
      3×4+5
27
```

is interpreted as:

```
      3×(4+5)
27
```

The user may control the order in which the individual operations in a statement are evaluated by explicitly parenthesizing the operations to be treated as a quantity. To cause the expression included above to evaluate to 17, not 27, the user enters the following:

```
      (3×4)+5
17
```

This expression evaluates to 17, because 5 is added to the quantity 3×4, not simply to 4.

2.2 FORMATTING APL NUMERIC OUTPUT

The *APL-11* system may be configured as either a single-precision or a double-precision system for the internal representation of floating-point numbers. The single-precision version of *APL-11* uses a precision of about seven decimal digits; the double-precision version uses a precision of about 16 digits.

The internal precision of numeric representation in *APL* is not subject to the user's run-time control. However, the user may specify both the desired precision of numbers to be displayed as output and the maximum length of the output line. The *)DIGITS* system command (Section 5.4.2) sets the output precision, and the *)WIDTH* system command (Section 5.4.3) sets the length of the line. The examples in this section illustrate the impact of both of these commands on the appearance of *APL* output. Vector and scalars are printed in a compact form; arrays and higher dimensional structures, however, are formatted for tabular output.

Before a numeric array is printed, it is scanned to determine the "best" output format. The columns of numeric arrays are aligned and packed together with at least one space of separation. Once the maximum field width has been determined for an array, the numbers are left-justified in that field. No attempt is made to align the decimal points. *APL* attempts to display all numbers without decimal points and exponents. When scalars, vectors, and arrays are being displayed, only those numbers that require exponentation are displayed in that form - for example:

```
          556 556.0 556E0 5.56E¯9
    556 556 556 5.56E¯9
```

Fractional numbers are displayed with a leading zero before the decimal point. Note that the maximum number of displayed digits has been set to six in the examples below.

```
          )DIGITS 6
    WAS 7
          1000000000
    1E+9
          1E¯2
    0.01
          ¯.0000000001
    ¯1E¯10
          1.234E5
    123400
          2 5ρ1
     1  1  1  1  1
     1  1  1  1  1
          2 3ρ.123 ¯.123 .123 .123 .123 5.4321E¯10
     1.2300E¯1  ¯1.2300E¯1   1.2300E¯1
     1.2300E¯1   1.2300E¯1   5.4321E¯10
          2 3ρ.123 3 .123 .123 123456 1E4
     0.1230   3        0.1230
     0.1230   123456   10000
```

When the length of a vector or array exceeds the maximum line width specified in the *)WIDTH* command, the excess numbers are indented beneath the second element of the first line, as shown below.

```
          )WIDTH 30                    (generate 30 consecutive numbers)
    WAS 72
          ⍳30
    1 2 3 4 5 6 7 8 9 10 11 12
          13 14 15 16 17 18 19
          20 21 22 23 24 25 26
          27 28 29 30


          ⍟⍳10                         (compute natural logarithms)
    0 0.69315 1.09861 1.38629
          1.60944 1.79176
          1.94591 2.07944
          2.19722 2.30259

          )DIGITS 3
    WAS 6
          ⍟⍳10
    0 0.69 1.1 1.39 1.61 1.79
          1.95 2.08 2.2 2.3
```

In *APL* there are actually two ways to control the number of digits displayed in numeric output. The *)DIGITS* system command sets the output precision directly, and the floor (⌊) function, illustrated in the last example below, rounds the numbers included in the function. In this example, the numbers are rounded to three places to the right of the decimal point.

```
      )DIGITS 6
WAS 3
      )WIDTH 72
WAS 30
      ⍟⍳5
0 0.69315 1.09861 1.38629 1.60944
      1E¯3×⌊.5+1E3×⍟⍳5
0 0.693 1.099 1.386 1.609
```

2.3 ERROR HANDLING

When an error is encountered in an *APL* statement, an error message is normally output, followed by a display of the line in which the error occurred. An up-arrow (↑) beneath this line identifies the particular point at which the error was discovered. Examples of several common error conditions are included below.

```
      B←3
      A×B

VALUE ERROR
A×B
↑
      1+1B+2+3

SYNTAX ERROR
1+1B+2+3
  ↑
      1 2+1 2 3 4 5 6 7

LENGTH ERROR
1 2+1 2 3 4 5 6 7
↑
```

Because *APL* is a highly interactive system, the user can almost always respond to an error condition simply by correcting the statement in which the error occurred. This characteristic of the language also facilitates a trial-and-error approach to program development.

In immediate mode, the user generally responds to an error message by reentering a corrected statement or by changing the value of a variable used in a computation. In function-execution mode, *APL* outputs an error message, along with the name of the function and the line number of the statement at which the error occurred; it also suspends execution of the function. The user may then terminate the suspended program, restart it at another statement, or perform debugging operations before resuming execution. These operations might include editing the suspended program, displaying the current values of variables used in the program, examining the status of functions called by the program, or developing a test program to analyze the output of the suspended program. Chapter 3 describes techniques for developing and executing functions.

If certain types of errors occur in function-execution mode, the user may not want execution of the function to halt to await correction of the error conditions. The implementation of *APL* described in this manual therefore allows the user to handle error conditions under program control by I-beam 16 (Section 4.3.2) and the execute (⍎) operator (Section 2.7.21).

2.4 ARRAY INDEXING AND COMPARISONS

This section introduces the use of array indexing in *APL* and provides background information on the function of the index origin and "fuzz" in performing comparisons. These concepts are helpful in understanding the examples included in subsequent sections of this chapter.

2.4.1 Indexing Arrays in APL

The concept of using and entering values for arrays in *APL* has already been introduced in this chapter. An element of an array may be indexed by specifying a bracketed element number to the right of the array name as follows:

```
    V[1]
```

This expression represents the simplest form of indexing and can be used to access the first element of vector *V*. If *V* consists of the vector shown below, then *V*[3] is 7.

```
          V←3 4 7 9
          V[3]
    7
```

In the examples shown above, the array being indexed is a vector, and the index is a scalar value representing the position of the desired element in the vector. A more complex form of indexing occurs when the array is of higher dimension or when the index is itself an array. The latter case is illustrated below.

```
          I←2 4 5
          V←10 22 31 49 56 68 72
          V[I]
    22 49 56
```

Here *V* and *I* are both vectors. The expression *V*[*I*] is used to access the elements of *V* referenced by *I* - the second, fourth, and fifth members of vector *V*. The result of *V*[*I*] is itself a vector consisting of the same number of elements as vector *I*. *I* may be a matrix or a higher-dimensional array; the result always has the same shape as *I*.

The array being indexed need not be a variable. It may be a constant set of values or even an expression to be evaluated, as shown below:

```
          7 6 5 4 3 2 1[2 4]
    6 4
          (2 4 8 16 * 2)[1 2]
    4 16
```

In general, there must be as many indices as there are dimensions in the array. In *APL*, the number of dimensions is known as the *rank*. For a vector, a single index is sufficient to identify the desired element. A matrix or 2-dimensional array requires two indices, separated by semicolons; a 3-dimensional array requires three. Thus, if

M is of rank *N*, then *M* must have *N* subscripts, separated by semi-colons. If *M* is a matrix, then *M*[2;4] is the element at the intersection of the second row and the fourth column of *M*; the first element in brackets identifies the row and the second specifies the column. The shape of the result of *M*[*I*;*J*] is (⍴*I*),⍴*J* - for example:

```
          M
  1  2  3  4
  5  6  7  8
          M[2;1]
5
          M[1 2;2 3]
  2  3
  6  7
```

A subscript may be omitted from an index specification, but the semi-colon must be included if only one matrix dimension specification is being omitted. If the right subscript is omitted, then all columns are selected from the matrix; if the left subscript is omitted, then all rows are selected - for example:

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

Note that the semicolon is required to indicate which subscript has been omitted. If the index specifications are completely omitted, as in the first example below, the entire array is displayed. In the second example, the entire array is displayed because one semicolon - one fewer than the number of dimensions in the array - is included.

```
          X←2 2⍴1 2 3 4
          X
  1  2
  3  4
          X[;]
  1  2
  3  4
```

Some additional examples are included below:

```
          V←'ABCDEF'
          V[3 5]
CE
          V[6 5 4 3 2 1]
FEDCBA
          M←2 3⍴2 5 4 6 5 4
```

```
        V[M]
BED
FED
        M←2 2ρ1 2 2 1
        A←M[M;M]
        A
 1  2
 2  1

 2  1
 1  2


 2  1
 1  2

 1  2
 2  1
        ρA
2 2 2 2
        A[1;;2;]
 2  1
 1  2
```

Indexing may also be used to change specified elements of an array by replacing their values with new values. An example of this is shown below.

```
        A
1  2  3
4  5  6
        A[1;2 3]←7 8
        A[2;1 2]←9
        A
1  7  8
9  9  6
        A[1;1]←12
        A
12  7   8
9   9   6
```

2.4.2 The Index Origin

The index origin specifies the index of the first element in an array. If the index origin is 1, then members of vector *V* are numbered *V*[1], *V*[2], and so on. If the index origin is 0, elements begin at *V*[0], not *V*[1]. The default index origin setting in the clear workspace is 1, but the user may change this setting to 0 or reset it to 1 by means of the ⎕IO system variable (Section 4.2.2) or the *)ORIGIN* system command (Section 5.4.1). The index origin setting is saved with the rest of the workspace.

The value of the index origin is also used by *APL* in many of the functions described in Chapter 2. These include:

- catenation (,)
- lamination (,)

- compression (/)
- expansion (\)
- dyadic transpose (⍉)
- reverse (⌽)
- rotation (⌽)
- grade up (⍋)
- grade down (⍒)
- roll (?)
- deal (?)
- reduction (*f*/)
- scan (*f*\)
- index generator (⍳)
- index of (⍳)

```
      ⎕IO←1
      A←⍳4
      A
1 2 3 4
      A⍳3
3
      A[3]
3
      ⍒A
4 3 2 1
      ?1
1
      5?5
1 2 3 4 5
      )ORIGIN 0
WAS 1
      A←⍳4
      A
0 1 2 3
      A⍳3
3
      A[3]
3
      ⍒A
3 2 1 0
      ?1
0
      5?5
4 0 3 1 2
```

2.4.3 Comparison Tolerance or Fuzz

When two very large numbers, or two numbers that have non-zero fractional components are compared in the *APL*-11 System, they are considered to be equal if they are within a certain comparison tolerance of "fuzz" quantity of each other. Comparison tolerance is used in the following *APL* functions:

- relational operators (<,≤,=,≥,>,≠)
- index function (dyadic ⍳)
- membership function (dyadic ∊)
- floor (⌊)
- ceiling (⌈)

The amount of tolerance applied by *APL*-11 may be controlled by the user by means of the ⎕CT system variable (Section 4.2.1) or the)FUZZ system command (Section 5.4.4). The default relative Fuzz in the clear workspace is set to 5E¯7 in single-precision systems and 5E¯15 in double-precision systems. The Fuzz setup is saved with the workspace.

The comparison tolerance is ⎕CT times the larger of the two numbers that are being compared, in absolute value. The formal definition for tolerant equality is the following:

```
     R←(|A-B)≤⎕CT×(|A)⌈|B
```

Examples of user control over comparison tolerance are included below.

```
          ⎕CT←1E¯9
          '01' [1+1=A←1+10*-⍳25]
    0000000011111111111111111
          ⎕CT←1E¯12
          '01' [1+1=A]
    0000000000001111111111111
```

2.5 INPUT/OUTPUT OPERATIONS

The implementation of *APL* described in this manual facilitates input and output operations on a variety of system devices. Chapter 6 describes the file system used to handle file-oriented I/O in ASCII sequential and random-access format. This section is oriented to terminal input and output, but most of the general information described here is applicable to all system I/O devices.

In *APL*, input and output operations are generally expressed by means of the special quad operator, ⎕. Input/output statements are special kinds of assignment statements. If a quad symbol appears immediately to the left of a left-arrow, the value of the expression to the right of that specification arrow is output - for example:

```
          X←15-⎕←3+4
    7
```

Here the quantity 3+4 is assigned to the quad operator and displayed. The value of X is computed but not displayed. Terminal output can also be accomplished simply by entering the name of the variable whose value is to be displayed:

```
          X
    8
```

If a quad symbol appears anywhere in an *APL* statement except immediately to the left of a left-arrow, input is accepted from the terminal, as in the following:

```
        A←3×⎕+5
⎕:
        7
```

Table 2-1 lists the formats of the input and output operations that can be performed in *APL*-11, along with section references.

Table 2-1
Input/Output Operators

Expression	Meaning	Section
A←⎕	Quad (evaluated) input	2.5.1
A←⍞	Quote-quad (character) input	2.5.2
A←⍫	Quad-del (unedited) input	2.5.3
A←*C*⌹[*type*]*N*	File input	6.2.2
A	Normal output	2.5.5
A;*B*;*C*	Heterogeneous output	2.5.6
⎕←*A*	Quad output	2.5.5
⍞←*A*	Bare output	2.5.7
C⌹[*type*]*A*	File output	6.2.3

The file input and output functions are described in detail in Chapter 6; the basic forms of the quad operator are discussed below.

2.5.1 Quad Input Mode

The most basic form of *APL* input is called *evaluated input*. Evaluated input means that the expression entered by the user is evaluated for a value, which then replaces the ⎕ character. In the following example, the value entered by the user is assigned to the variable to the left of the specification arrow.

```
        K←⎕
⎕:
        18
```

The *K* variable takes on the value (18) entered by the user at the terminal.

APL prompts the user to supply a value by displaying a quad character followed by a colon, as shown below.

```
        A←3×⎕+5
⎕:                          The user requests evaluated input.
        7
        A                   APL prompts and the user enters a
36                          string which again requests evaluated
        3×⎕÷8               input.
⎕:                          APL prompts again and the user enters 3.
        16×⎕-2
⎕:                          The second input string evaluates to
        3                   16×3-2 or 16, and the first evaluates
6                           to 3×16÷8=6; APL responds with 6.
```

To enter a character string as a value in quad mode, the user must enclose the string in single quotes - for example:

```
        MSG←⎕
⎕:
        'NOT ENOUGH CORE'
        MSG
NOT ENOUGH CORE
```

If the user enters only a carriage return or spaces followed by a carriage return, *APL* again displays the prompt and waits for the input to be reentered. While the system is awaiting input, the user may enter and execute a system command or may define a function. The input request remains pending. After the desired operation has been performed or the user has returned to immediate mode, *APL* prompts again and waits for input. If an error is encountered in the input, *APL* displays the appropriate message and allows the user to reenter the input but does not reprompt. To reenter the input, the user must first type →0, which will cause the prompt to appear; he may then reenter the input.

2.5.2 Quote-Quad Input Mode

A version of the quad operator called the quote-quad operator (⍞) is used especially for the input of character data. An example of quote-quad mode is shown below.

```
        X←⍞
THAT'S AMAZING
        X
THAT'S AMAZING
```

Unlike evaluated input, quote-quad input allows character strings to be entered without explicit quote characters. When *APL* encounters a ⍞ symbol, it positions the carriage at the left margin and accepts the data entered by the user up to the next carriage return as a character string. If a single character is entered, *APL* treats it as a literal scalar; a string is stored as a literal vector. If the user enters only a carriage return, *APL* treats this input as a vector of length 0; this is significantly different from the handling of empty input in evaluated input mode, in which *APL* rejects the input and waits for the user to reenter it.

Quote-quad input is also called unevaluated input. If the user enters an expression, *APL* does not evaluate it, but simply treats it as a character string. *APL* does edit the characters that are entered; for example, overstrikes which are made up of three separate characters are combined into a single character.

2.5.3 Quad-Del Input Mode

A special version of the quad operator, the quad-del operator (⍔) enters characters exactly as typed by the user. No special editing of *APL* characters is performed. The backspace, for example, is treated as a special character, and an overstrike symbol is not created. The following statements illustrate the difference between quad-del and quote-quad modes in entering overstruck *APL* characters.

```
      X←⍔
⍉A
      ⍴X
4
      X←⍞
⍉A
      ⍴X
2
      ⍴'⍉A'
2
```

The example included below shows the particular use of quad-del mode in accepting input from ASCII-mode terminals. Mnemonics entered in ASCII mode are not decoded.

```
      A_.QD
.TRB
      .RO A
4
      A_.QQ
.TRA
      .RO A
2
      .RO '.TRA'
2
```

As in quote-quad input mode, if the user enters only a carriage return, *APL* treats this input as a null vector of length zero.

2.5.4 Escaping from an Input Loop

If an input request occurs within an infinite loop in an *APL* defined function, the user can interrupt function execution by typing *OU*, as follows:

O<backspace>U

thus overstriking the two characters. Users of ASCII terminals escape by typing the .OU mnemonic. An escape of this kind causes function execution to be interrupted but does not cause an exit from the function.

2.5.5 Normal and Quad Output Modes

If a quad operator (⎕) appears immediately to the left of a left-arrow, the value of the expression to the right of the arrow is output. Because *APL* automatically displays the value of an expression or variable not explicitly assigned to another variable, it is often not necessary to express explicit terminal output requests. For example, the *APL* statement:

```
      ⎕←B
3
```

is equivalent to the statement:

```
      B
3
```

because both have the effect of displaying the value of *B*.

Quad output mode is especially helpful when an *APL* statement consists of multiple specifications - for example:

```
      A←3+⎕←5×4
20
```

This statement performs the computation and displays the desired output - the result of the computation 5×4. It is more efficient than the following similar examples:

```
      5×4
20
      A←3+20


      A←5×4
      A
20
      A←3+A
      A
23
```

If the last operation (the leftmost operation) being performed in a line is an assignment (←) or branch (→) (see Section 3.4.1), then no final output is produced. The following *APL* statement will not cause output to be displayed:

```
A←5×4
```

but the example shown below will display a value:

```
      4+A←5
9
```

2.5.6 Heterogeneous Output Mode

APL users often need to mix character and numeric data on the same output line. Mixed output lines of this kind are called *heterogeneous output*. The *APL* user requests heterogeneous output simply by entering a series of values or expressions, separated by semicolons, in the order in which they are to appear. The values may be parenthesized. The following is an example of the use of heterogeneous output.

As mentioned in Section 2.5.5, a value will not be displayed if the leftmost expression on a line is an assignment or branch operation.

The heterogeneous output facility may be useful for entering function lines that consist of multiple *APL* statements. The user should remember, however, that *APL* evaluates expressions in right-to-left order by line, without regard to embedded separating semicolons.

2.5.7 Bare Output Mode

Bare output is a special kind of *APL* output that is normally accomplished by means of the quote-quad character - for example:

```
    ⍞←'SPECIFY USER ID'
```

The normal output described in Section 2.5.5 is terminated by a carriage return/line feed pair so that the next input or output begins at a standard position on the following line. Bare output, on the other hand, is not concluded by a carriage return/line feed if it is followed by another bare output request or by quote-quad input. The character input accepted after a bare output operation is handled as if the user had spaced over to the position immediately following the final character of the bare output value. This implies that the resulting value of the input string normally contains a number of blanks, as shown in the example below. The use of bare output allows a character response to appear on the same line as the output text.

```
      ∇INIT
[1]   ⍞←'ARE YOU READY TO ENTER VALUES? '
[2]   A←⍞
[3]   ∇

      INIT
ARE YOU READY TO ENTER VALUES? NO

      A
                               NO
```

Carriage returns that would normally be inserted because of a limitation on page width are not included in bare output.

If bare output is specified in immediate rather than function-execution mode, it is usually not distinguishable from normal output. A bare output statement such as ⍞←*A* must be followed by an input entry at the terminal, and thus the output will be concluded by the conventional carriage return. Bare output is therefore more appropriately utilized in function-execution mode.

2.5.8 Terminating Output

The display of output on the terminal may be terminated before it has been completed by pressing the CTRL/O or CTRL/C key. See the discussion in Sections 1.5.1 and 1.5.2.

2.6 PRIMITIVE SCALAR FUNCTIONS

APL primitive functions are of two types: *scalar* and *mixed*. Scalar functions have the following characteristics:

- They take single-number (scalar) arguments
- They yield scalar results
- They are used primarily for basic arithmetic and logical operations, such as addition, exponentiation, maximum value, and logical OR

With a few exceptions, the primitive scalar functions take numeric scalar arguments. Only the relational functions (<,≤,=,>,≥,≠) take either character or numeric arguments.

The logical functions (∨,∧,⍱,⍲,~) must have arguments that are equal to 0 or 1.

Table 2-2 summarizes the primitive scalar functions available in this implementation of *APL*, and provides a definition or example of each. Most of the functions are straightforward and familiar arithmetic or logical functions and do not require detailed discussion.

The following subsections describe the difference between monadic and dyadic primitives, discuss the extension of scalar functions to arrays, describe the use of *APL* operators with primitive functions, and summarize any information about the functions in Table 2-2 that is either not obvious or different from the ordinary mathematical interpretation of the functions. The monadic roll (*?*) primitive is a scalar function and is included in the table for completeness; however, it is the only primitive scalar function that is origin-dependent (Section 2.4.2) and is more appropriately described in conjunction with the dyadic deal function in Sections 2.7.21 and 2.7.22.

2.6.1 Monadic and Dyadic Functions

Most of the primitive scalar functions and some of the mixed functions described in Section 2.7 have been implemented in two forms: monadic and dyadic. *Monadic functions* take only a right argument - for example, ÷*A* (reciprocal), !*B* (factorial) or ~1 (logical NOT). *Dyadic functions* take both left and right arguments - for example, 3+2 (addition), *A*⌈*B* (maximum), and *X*=*Y* (equal). The operator is always a single *APL* symbol, usually the same as the corresponding symbol used in ordinary mathematics.

The syntax of a function (i.e., the presence of one or two arguments) determines whether the function is monadic or dyadic. For example, |*A* is a monadic function used to determine the magnitude or absolute value of the argument *A*. *A*|*B* is a dyadic function used to obtain the residue or remainder available after dividing *B* by *A*. The particular function specifed by the | symbol is dependent upon the context of the statement.

Table 2-2
Primitive Scalar Functions

Monadic Form *fY*		Symbol	Dyadic Form *XfY*	
Definition or Example	Function		Function	Definition or Example
+*Y*←→0+*Y*	Plus	+	Plus	5.3+4.2←→9.5
-*Y*←→0-*Y*	Negative	-	Minus	5-6←→¯1
×*Y*←→(*Y*>0)-*Y*<0	Signum	×	Times	4×7.2←→28.8
÷*Y*←→1÷*Y*	Reciprocal	÷	Divide	5÷2←→2.5
⋆*Y*←→(e=2.71828...)⋆*Y*	Exponential	⋆	Power	9⋆0.5←→3
¯6.7←→6.7	Magnitude	|	Residue	5|7←→2 5|¯7←→3 ¯5|¯7←→¯2 7|0←→0
Y ⌈*Y* ⌊*Y* 5.47 6 5 ¯5.47 ¯5 ¯6	Ceiling	⌈	Maximum	5⌈2←→5
	Floor	⌊	Minimum	3⌊7←→3
⍟⋆*N*←→*N*←→⋆⍟*N*	Natural Logarithm	⍟	Logarithm	*X*⍟*Y*←→Log *Y* Base *X* *X*⍟*Y*←→(⍟*Y*)÷⍟*X*
!0←→1 !*Y*←→*Y*×!*Y*-1 or !*Y*←→Gamma(*Y*+1)	Factorial	!	Binomial Coefficient	*X*!*Y*←→(!*Y*)÷(!*X*)×!*Y*-*X* 3!5←→10 2!6←→15
?*Y*←→Random choice from ⍳*Y*	Roll	?		
○*Y*←→(3.14159...)×*Y*	Pi times	○	Circular	See Table 2-3
~1←→0 ~0←→1	Not	~		
		∧ ∨ ⍲ ⍱	And Or Nand Nor	*X* *Y* | *X*∧*Y* | *X*∨*Y* | *X*⍲*Y* | *X*⍱*Y* 0 0 | 0 | 0 | 1 | 1 0 1 | 0 | 1 | 1 | 0 1 0 | 0 | 1 | 1 | 0 1 1 | 1 | 1 | 0 | 0
		< ≤ = ≥ > ≠	Less Not greater Equal Not less Greater Not equal	Relationals: Result is 1 if the relation holds and 0 if it does not. 3>7←→0 '*A*'≤'*C*'←→1

Table 2-3
Dyadic Circle Functions

(-X)○Y	X	X○Y
(1-Y*2)*.5	0	(1-Y*2)*.5
Arcsin Y	1	Sine Y
Arccos Y	2	Cosine Y
Arctan Y	3	Tangent Y
(¯1+Y*2)*.5	4	(1+Y*2)*.5
Arcsinh Y	5	Sinh Y
Arccosh Y	6	Cosh Y
Arctanh Y	7	Tanh Y

2.6.2 Extending Scalar Functions to Arrays

The primitive functions described in this section are considered scalar functions because they take scalar arguments and yield scalar results. The operations performed by these functions can, however, be extended to arrays. A primitive scalar function is applied to an array on an element-by-element basis. Thus, if the user specifies an addition function in which both arguments are vectors, the corresponding elements of the vectors are added - for example:

```
      5 8 9+6 7 2
11 15 11
```

The arrays on which the primitive scalar functions operate may be of any dimensions. If a dyadic function is being performed, the arrays specified as the arguments of the function must generally have the same number of elements and be the same shape (e.g., a 2-by-3 array is not equivalent to a 3-by-2 array). There is one exception to this rule. If one argument is an array and the other is a scalar or a single-element array, the single value is applied to every element of the array. The following two examples are therefore equivalent.

```
      5 5 5+6 7 2
11 12 7
      5+6 7 2
11 12 7
```

The following examples illustrate the use of several other primitive scalar functions.

```
      A←3 3⍴5 6 8 3 2 1 6 4 2
      A
 5  6  8
 3  2  1
 6  4  2
      A×A
 25  36  64
 9   4   1
 36  16  4
      2×A
 10  12  16
 6   4   2
 12  8   4
      2*0 1 2 3 4 5 6 7 8
1 2 4 8 16 32 64 128 256
      4 9 16 25 36*0.5
2 3 4 5 6
```

2.6.3 Using Operators with Scalar Functions

Operators are special *APL* functions that take dyadic primitive scalar functions as their arguments. For example, the reduction operator combines the elements of a vector or the elements along one dimension of an array. The elements are combined in accordance with the specified function (e.g., addition, multiplication, etc.). The following example illustrates the addition of the elements of a vector.

```
      +/1 2 3
6
```

The plus sign in this statement could be replaced by any of the dyadic primitive scalar functions in order to perform a different function.

The formats of the four *APL* operators are listed below and are described in detail in Section 2.8.

- reduction (*f*/)
- scan (*f*\)
- inner product (*f*·*g*)
- outer product (∘·*f*)

2.6.4 Relational Functions

In *APL*, the relational functions (<, ≤, =, >, ≥, ≠) return results; they are not simply comparison operators. An expression of the form *A≤B* yields a result value of 1 if the relation holds - for example:

```
        9>6
1
        4>6
0
        'C'>'A'
1
```

These functions may take either numeric or character arguments; however, they may not have one numeric and one character argument, or a *DOMAIN ERROR* results. Note that = and ≠ will return a 0 result for arguments of different types. For characters, the ⎕*AV* system variable defines the collating sequence to be used in relational functions. A character appearing earlier in ⎕*AV* is "less than" one appearing later (Section 4.2.6).

When used with boolean arguments (0 and 1), the relational functions may be used to perform logical operations. For example, the not equal (≠) function performs an exclusive OR operation if its arguments are 0's and 1's.

```
        0 1 0 1≠0 0 1 1
0 1 1 0
```

2.6.5 |: Determining the Residue

The dyadic residue (|) function is used to obtain the remainder or residue of a number. In the function:

```
    5|8
```

where 5 and 8 are both positive, 3 is the remainder when 8 is divided by 5, and 3 is considered the 5 residue of 8. The residue is a unique number whose value is in the range between the value of the left argument and zero.

The residue function, *A|B*, has the following characteristics:

- If the left and right arguments are equal (*A=B*), the residue is 0.
- If the left argument is zero (*A=0*), the residue is the value of *B* (*A|B = B*).
- If *A* is not zero (*A≠0*), the residue is in the range *A* through *0*; it may equal *0* but not *A*. For some integer, *I*, the residue can be expressed as *B-I×A*.

Examples of these cases are included below. For a discussion of the outer product operator included below (*A*∘.|*B*), see Section 2.8.4.

```
          7|7
0
          7|0
0
          0|7
7
          0|¯7
¯7
        A←3 0 ¯3
        B←¯6 ¯5 ¯4 ¯3 ¯2 ¯1  0  1  2  3  4  5  6
        A∘.|B
 0  1  2  0  1  2  0  1  2  0  1  2  0
¯6 ¯5 ¯4 ¯3 ¯2 ¯1  0  1  2  3  4  5  6
 0 ¯2 ¯1  0 ¯2 ¯1  0 ¯2 ¯1  0 ¯2 ¯1  0

        X←21.824
        .01|X
0.004
```

The result of a residue function has the same sign as the left argument of the function. If the left argument is negative, then the sign of the result is negative, as shown below.

```
           ¯5|¯7
      ¯2
```

because ¯2 = ¯7 + | ¯5

The arguments of the residue function need not be integer numbers - for example:

```
          2|5.8
    1.8
          1.2|3.9
    .3
```

The formal definition of the *APL*-11 residue function is the following:

```
    A|B←→B-A×⌊B÷A+A=0
```

where ←→ indicates that the two sides of the expression have the same value.

2.7 PRIMITIVE MIXED FUNCTIONS

The functions presented in this section are primitive *mixed functions*. Primitive scalar functions take scalar arguments, yield scalar results, and are extended to arrays on an element-by-element basis. Mixed functions, on the other hand, may take array arguments and yield scalar or array results, or may take scalar arguments and yield array results.

2.7.1 Summary of Primitive Mixed Functions

Table 2-4 summarizes the primitive mixed functions available in this implementation of *APL*, along with the operators introduced in Section 2.6.3 and described in Section 2.8.

Table 2-4
Primitive Mixed Functions and Operators

Monadic Form fY			Takes Coordinate Argument[1]	Symbol	Origin-Dependent[1]	Dyadic Form XfY		
Section	Function	Definition				Definition	Function	Section
Mixed Functions:								
2.7.3	Returns array shape	⍴Y	no	⍴	no	X⍴Y	Reshapes an array	2.7.4
2.7.5	Generates consecutive integers	⍳Y	no	⍳	yes	X⍳Y	Finds an index	2.7.6
2.7.7	Converts to a vector	,Y	yes[2]	,	no	X,Y	Catenates or laminates	2.7.8
			yes	/	no	X/Y	Compresses an array	2.7.9
			yes	\	no	X\Y	Expands an array	2.7.10
			no	↑	no	X↑Y	Takes array elements	2.7.11
			no	↓	no	X↓Y	Drops array elements	2.7.12
2.7.13	Transposes an array	⍉Y	no	⍉	yes[3]	X⍉Y	Transposes an array	2.7.14
2.7.15	Reverses an array	⌽Y	yes	⌽	no	X⌽Y	Rotates an array	2.7.16
2.7.17	Sorts in ascending order	⍋Y	yes	⍋	yes			
2.7.18	Sorts in descending order	⍒Y	yes	⍒	yes			
2.7.19	Rolls random integers	?Y	no	?	yes	X?Y	Deals random integers	2.7.20
2.7.21	Constructs a character string	⊤Y	no	⊤	no	X⊤Y	Encodes a number in another base	2.7.22
			no	⊥	no	X⊥Y	Decodes a number representation	2.7.23
2.7.24	Executes a character string	⊂Y	no	∊	no	X∊Y	Determines array membership	2.7.25
2.7.27	Eliminates duplicates in a set	∪	no	∪	no	X∪Y	Determines union of two sets	2.7.26
			no	∩	no	X∪Y	Determines intersection of two sets	2.7.28
			no	~	no	X~Y	Excludes elements in first set but not in second	2.7.29
			no	⊂	no	X⊂Y	Determines a proper subset	2.7.30
			no	⊃	no	X⊃Y	Determines a strict superset	2.7.31
			no	⊇	no	X⊇Y	Determines a superset	2.7.31.1
			no	⊆	no	X⊆Y	Determines a subset	2.7.30.1
2.7.32	Formats an array	⍕Y	no	⍕	no	X⍕Y	Formats a numeric array with width and precision	2.7.33
2.7.34	Performs matrix inversion	⌹Y	no	⌹	no	X⌹Y	Performs matrix division	
Operators:								
2.8.1	Reduces an array	f/Y	yes	f/	no			
2.8.2	Scans an array	f\Y	yes	f\	no			
			no	f.g	no	Xf.gY	Computes inner product	2.8.3
			no	∘.g	no	X∘.gY	Computes outer product	2.8.4

[1]Apply to both monadic and dyadic forms
[2]Dyadic form only
[3]Dyadic form, left argument only

The boxed information at the beginning of each section provides additional summary information, which is repeated for quick reference in Appendix A (Table A-9). In these descriptions, "any" means that any argument domain (character or numeric) or argument shape (scalar, vector, or array) may be specified. If the argument domain is "any*", this indicates that arguments may be either character or numeric, but both arguments must be the same type.

2.7.2 Specifying Array Coordinates

When expressing mixed functions for arrays of two dimensions or more, it may be necessary to specify the particular array coordinate to which the function applies. This is done by including in the function a bracketed expression representing the desired coordinate in the specified array. For example, the following function catenates array *A* to dimension *1* of *B*.

```
    A,[1] B
```

An array coordinate can be specified for the following functions and operators:

Function	Symbol	Section
catenation	,	2.7.7
lamination	,	2.7.8
compression	/	2.7.9
expansion	\	2.7.10
reverse	⌽	2.7.15
rotation	⌽	2.7.16
sort (ascending)	⍋	2.7.17
sort (descending)	⍒	2.7.18
reduction	f/	2.8.1
scan	f\	2.8.2

The array coordinate is origin-dependent, that is, it depends upon the current value of the index origin. In the above example, *A* is catenated to the first dimension of *B* if the index origin is 1 and to the second dimension of *B* if the index origin is 0.

If the bracketed expression is omitted from a mixed function, the function is performed on the *last* coordinate of the array. If *B* is a 4-dimensional array, the following function compresses along coordinate 4.

```
    A/B
```

The user can specify that certain functions are to be performed on the *first* coordinate by using a special symbol, formed by overstriking the minus sign (-) with another symbol, usually the normal symbol of the function - for example:

```
    A⌿B
```

All symbols are shown below.

Function	Symbol
compression	⌿
expansion	⍀
reverse	⊖
rotation	⊖
reduction	f⌿
scan	f⍀

2.7.3 ρ: Returning the Shape of an Array

Function: monadic rho (ρ); *R*←ρ*Y*
Argument Domain:
  left: -
  right: any
Argument Shape:
  left: -
  right: any
Result Range: null or non-negative integers
Result Shape: vector; ρ*R*↔ρρ*Y*
Origin-Dependent? no
Take Dimension Argument? no

The monadic form of the rho (ρ) function returns the shape of an array. If *B* is a character vector consisting of '*ABCDEF*', then the rho function included below returns the number of characters in the array.

```
      B
ABCDEF
      ρB
6
```

Because *B* is a 1-dimensional array, ρ*B* returns only a single number. If *A* is a matrix with five rows and six columns, then the following result occurs.

```
      A
 1   2   3   4   5   6
 7   8   9  10  11  12
13  14  15  16  17  18
19  20  21  22  23  24
25  26  27  28  29  30
      ρA
5 6
```

If the vector that is the argument of the function is a 1-dimensional array with a length of 1, then the rho of the array will be 1. The following example illustrates the generation and examination of an array consisting of the single digit, 3.

```
      K←1ρ3
      ρK
1
```

See Section 2.7.4 for a discussion of the dyadic form of the rho function used in this example.

If the value of *K* generated in the example above is a scalar, not an array, then the rho of *K* is the null vector, a vector of length zero - for example:

```
K←3
ρK
```

APL simply displays a blank line in response to the ρ*K* statement. The shape of any single scalar, including zero, is the null vector. The shape of the null vector is zero. This is illustrated in the following example.

```
      ρ0

      ρ(ρ0)
0
```

The ρ*K* function always returns one element for each dimension of the array *K*. The following is an example of a rho function on a 2-dimensional array.

```
      A
5 9
6 3
2 2
      ρA
3 2
```

The expression ρ*A* returns the dimensions of *A* as number of rows, followed by number of columns.

The function ρρ*K* can be used to return the rank of *K* as follows:

Array	ρρ*K*
Scalar	0
1-dimensional	1
2-dimensional	2
3-dimensional	3

This effect is the result of the fact that ρ*K* is a vector containing one element for each dimension of *K*, so its rho, ρ(ρ*K*), is a 1-element vector consisting of the number of dimensions of *K*.

The function ρρρ*K* returns 1 for all possible *K*'s.

2.7.4 ρ: Reshaping an Array

Function: dyadic rho (ρ); *R*←*X*ρ*Y*
Argument Domain:
left: non-negative integers
right: any
Argument Shape:
left: scalar or vector; (ρρ*X*)≤1
right: any
Result Range: same as right argument
Result Shape: array; ρ*R*↔*X* for a vector
Origin-Dependent? no
Take Dimension Argument? no

The dyadic form of the rho function specifies a new array or reshapes an existing one. It is issued as shown in the following example:

```
      3ρ5
5 5 5
```

where the left argument, 3, specifies the shape of the array to be constructed and the right argument, 5, specifies the value to be assigned to each element of the array. The shape of the array describes both the number of dimensions of the array and the number of elements in each dimension. In the example above, a 1-dimensional array is created, because only a single value is supplied to the left of the ρ; the number of elements is the actual value of the argument, 3.

The right argument of the dyadic ρ function may be any shape. The example above illustrates the generation of a numeric constant array. An array consisting of literal characters can be constructed by including a character string as the right argument and enclosing it in quotes. A character vector reshaped in this way is displayed without spaces, as shown in the following example.

```
      2 3ρ'ABCDEF'
ABC
DEF
```

The examples included below illustrate the generation of two arrays. The first example reshapes an existing array; the second specifies the elements of a new array in the rho function.

```
      X←1 2 3 4
      Y←2 2ρX
      2 2ρ4 3 2 1
4 3
2 1
```

The array that is being reshaped need not have the same number of values as the array from which values are taken. In the following expression:

```
A←5ρB
```

A requires five elements. If *B* has more than five elements, then only the first five are used. If *B* has fewer than five elements, then the elements in *B* are repeated as often as necessary, in row-major order. The following example illustrates both of these operations, first shaping a 2-dimensional array and then reshaping it into a vector.

```
        A←1 2 3 4
        2 5⍴A
  1  2  3  4  1
  2  3  4  1  2
        3⍴A
 1 2 3
```

The next example reshapes a character vector into a 3-dimensional array.

```
        2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
  ABCD
  EFGH
  IJKL

  MNOP
  QRST
  UVWX
```

A general rule for the dyadic rho function can be expressed as the following: if *A*←*V*⍴*B*, then ⍴*A*←→*V* and *A* contains only elements of *B*. A relationship between the rho and ravel (Section 2.7.7) functions can also be described as *V*⍴*B*←→*V*⍴,*B*.

The rho function is often used in conjunction with iota (Section 2.7.5). The next example generates an array consisting of consecutive integers.

```
        ⎕←A←2 2⍴⍳4
  1  2
  3  4
```

Any number of array elements can be specified in a dyadic rho function, as long as the number is not negative or fractional and does not generate an array too large for the user's workspace.

The rho function may be used to generate a null or empty vector. A vector of this kind is often useful in executing *APL* functions. As described in Section 3.4.1, if an empty vector is the argument of a branch, then function execution will not branch but will continue to the next statement in sequence.

An empty vector is generated when the right argument of the rho is a scalar. Some examples of expressions that generate null vectors are included below.

```
   ⍴A          (where A is a scalar)
   0⍴''
   0⍴0
   ⍳0
```

2.7.5 ⍳: Generating Consecutive Numbers

Function: monadic iota (⍳); *R*←⍳*Y*
Argument Domain:
  left: -
  right: non-negative integers
Argument Shape:
  left: -
  right: scalar or 1-element vector
Result Range: non-negative integers
Result Shape: vector; ⍴*R*←→,*Y*
Origin-Dependent? yes, result
Take Dimension Argument? no

The monadic form of the iota (⍳) function is used as an index generator. It generates a number of consecutive integers, equal to the value specified as the argument of the iota, starting from the value of the index origin. The following is an example of this function.

```
      ⎕←A←⍳4
1 2 3 4
      ⍴A
4
```

The argument of the function must be a non-negative integer scalar or a 1-element array.

The expression ⍳*N* generates a vector containing *N* components. If the index origin is set to 1, these components have values 1 through *N*. If the origin is 0, then the resulting vector has values 0 through *N*-1. The index origin default is 1 in the clear workspace, but this setting can be changed by the user by means of the ⎕*IO* system variable (Section 4.2.2) or the)*ORIGIN* system command (Section 5.4.1), as shown below.

```
      ⍳3
1 2 3
      )ORIGIN 0
WAS 1
      ⍳3
0 1 2
```

The monadic iota function can be used in any expression to generate consecutive results. The following example illustrates the use of iota in generating powers of 2.

```
      2*⍳10
1 2 4 8 16 32 64 128 256 512
```

Iota is often used in conjunction with rho.

To generate a vector with the same number of entries as array *X*, the user can specify the expression shown below; in this case, array *X* contains four elements.

```
      X←7 1 3 4
      ⍳⍴X
1 2 3 4
```

As illustrated in the following example, the index generator function can be used to generate a null or empty vector; the shape of a null vector is always zero.

```
        ⍳0
                                    (APL outputs a blank line)
        ⍴⍳0
    0
```

This function may also be used to determine the value of the current index origin:

```
        ⍳1
    1                               (index origin is 1)

        ⍳1
    0                               (index origin is 0)
```

2.7.6 ⍳: Finding the Index of a Value

Function: dyadic iota (⍳); *R*←*X*⍳*Y*
Argument Domain:
  left: any*
  right: any*
Argument Shape:
  left: vector; (⍴⍴*X*)≤1
  right: any
Result Range: non-negative integers
Result Shape: scalar or array; ⍴*R*←→⍴*Y*
Origin-Dependent? yes, result
Take Dimension Argument? no

The dyadic form of the index (⍳) function locates the first occurrence of a particular value in a vector - for example:

```
      X
4 9 6 8
      Y
6
      X⍳Y
3
```

The value of *Y* occurs as the third element of vector *X*. When using the dyadic form of iota, *X* can be scalar or a vector and *Y* can be any scalar or array.

The index function can be used to locate a particular type of value in a vector. For example, to find the index of the largest value in *X*, the following is specified:

```
      ⎕←A←X⍳⌈/X
2
      X[A]
9
      X[X⍳⌈/X]
9
```

The right argument of the index function may be an array. If *B* is the vector:

```
B←0 1 2 3 4 5 6 7 8 9
```

and *A* is a 2-dimensional array:

```
      A
6  5
3  2
0  9
```

then the following can be specified:

```
      X←B⍳A
      X
7  6
4  3
1  10
```

The result of a dyadic iota function *X*←*B*⍳*A* always has the same shape as the right argument of the function - formally ⍴*X*←→⍴*A*. If *A* is a matrix, then the correspondence between *A* and *X* can be expressed as follows: *X*[*I*;*J*] is the smallest *K* such that *A*[*I*;*J*] is equal to *B*[*K*].

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

The right argument of the function can be an array of literal characters, as shown below.

```
        'ABCDEFGH'⍳'HEADED'
    8 5 1 4 5 4
```

If the array identified by this argument contains a number or literal that cannot be found in the left vector, then *APL* responds with the next index number after the last element of the vector. In the following example, *APL* tries to locate the numbers 1, 2, 3, and 4 in vector *V*. There is no occurrence of 1 in the 6-element vector, so the next available index, 7, is displayed as the index of 1.

```
        V←5 4 2 3 7 8
        A←2 2⍴⍳4
        V⍳A
    7  3
    4  2
```

The next index number can be expressed as 1+⍴*V*.

The examples included so far in this section have assumed that the index origin setting is 1. If the origin has been set to zero, index values are returned as shown in the example below.

```
        'ABCDEF'⍳'CX'
    3 7
        )ORIGIN 0
    WAS 1
        'ABCDEF'⍳'CX'
    2 6
```

2.7.7 ,: Converting a Value to a Vector

Function: monadic ravel (,); *R*←,*Y*
Argument Domain:
 left: -
 right: any
Argument Shape:
 left: -
 right: any
Result Range: same as argument
Result Shape: vector; ρ*R*↔×/ρ*Y*
Origin-Dependent? no
Take Dimension Argument? no

The monadic ravel (,) function constructs a vector from any scalar or array. The following example illustrates the use of the ravel function in transforming a 2-dimensional array into a vector.

```
      A
 1  2  3
 4  5  6
      ρA
2 3
      ⎕←B←,A
1 2 3 4 5 6
      ρB
6
```

The vector produced by ravel has the same number of elements as the original array. The elements of the array are preserved in the resulting vector in row major order. If the argument to the right of the , is already a vector, then *B*↔,*A*.

The ravel function may be used to transform a scalar value into a single-element vector. If *A* is a scalar, then ,*A* produces a vector containing one element:

```
  A←,A
```

Note below the difference between the shape of a scalar (null vector) and the shape of a scalar to which the ravel function has been applied.

```
      ρ4        (APL outputs a blank line)
      ρ,4
1
```

2.7.8 ,: Catenating and Laminating Variables

Function: dyadic catenation (,); *R←X,Y*
Argument Domain:
left: any*
right: any*
Argument Shape:
left: array
right: array
Result Range: same as argument
Result Shape: array
Origin-Dependent? no
Take Dimension Argument? yes

The dyadic catenation and lamination (,) functions are used to chain scalars or arrays together to form a new array. *Catenation* joins variables together along an existing dimension; *lamination* joins them together along a new dimension. The following example illustrates the catenation of two vectors to each other and to several scalar values.

```
      A←5 8 9
      B←6 7
      A,B
5 8 9 6 7
      10,A,B,12
10 5 8 9 6 7 12
```

Any number of items can be catenated. The order in which values are catenated is the order in which they are specified in the *APL* statement. The result of a catenation can be expressed as follows: if ρ*A*←→5 and ρ*B*←→3, then ρ*R*←*A*,*B* is 8, *R*[⍳5]←→*A* and *R*[5+⍳3]←→*B*.

Catenation is useful in adding new subtotals to a grand total or for inserting new elements between existing elements of a vector. The following example illustrates the insertion of the scalar value 6 in vector *A*.

```
      A←1 2 3 4 5 7 8 9 10 11 12
      ⎕←A←A[⍳5],6,A[5+⍳(ρA)-5]
1 2 3 4 5 6 7 8 9 10 11 12
```

Literal values can also be catenated, as shown in the following example:

```
      'NAME','XY'
NAMEXY
```

APL does not allow the user to catenate numbers to literal characters and displays a *DOMAIN ERROR* if such an operation is attempted.

The dyadic catenation function may also be used to joint multi-dimensional arrays together along an existing coordinate. The user includes this integer coordinate number in brackets in the function specification. If the coordinate is omitted, *APL* assumes the last coordinate (1 or the rank of the array, whichever is larger (1⌈ρρ*A*)).

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

For a 2-dimensional array, *APL* extends along the second dimension, thus adding a column, as shown in the first example below. In the second example, the scalar value 0 is catenated with the array *A* along the coordinate specified by the user; this has the effect of adding a row. As discussed in 2.6.2, *APL* extends the scalar argument, 0, to the array on an element-by-element basis.

```
         A←2 3⍴⍳6
         A,0
 1  2  3  0
 4  5  6  0

         A,[1]0
 1  2  3
 4  5  6
 0  0  0
```

A scalar value can be included in the catenation function, as shown in the following:

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

Both arguments of the catenation function may be arrays. In the following example, the arrays are of equal size.

```
         X
 8  7  3
 2  9  4

         Y
 0  1  2
 3  4  5

         X,[1]Y
 8  7  3
 2  9  4
 0  1  2
 3  4  5

         X,Y
 8  7  3  0  1  2
 2  9  4  3  4  5
```

The next example illustrates the catenation of two arrays of different sizes.

```
         A←2 3⍴⍳6
         A
 1  2  3
 4  5  6
         B←3 3⍴6+⍳9
         B
 7   8   9
 10  11  12
 13  14  15
         ⍴C←A,[1]B
```

```
5 3
        C
 1   2   3
 4   5   6
 7   8   9
 10  11  12
 13  14  15
```

Three general rules can be established for catenating arrays according to the form *A*,[*K*]*B*. If a catenation expression does not conform to any of the rules presented below, it is not a legal *APL* expression.

1. If the arrays have equal dimensions ((ρρ*A*)=ρρ*B*), then *K* must be in ιρρ*A* and ρ*A* must equal ρ*B* except in the *K*th dimension. This is illustrated in the following example:

```
        ρA
3 4 5
        ρB
3 6 5
        R←A,[2]B
        ρR
3 10 5
```

   Here *A* is equivalent to *R*[;ι4;] and *B* to *R*[;4+ι6;].

2. If the arrays have different dimensions ((ρρ*A*)≠ρρ*B*), then *B* must have one fewer coordinates than *A* or vice-versa (1=|(ρρ*A*)-ρρ*B*) and ρ*B* must equal ρ*A* without its *K*th coordinate. This is shown below.

```
        A←3 4 5ρ0
        B←4 5ρ0
        R←A,B
LENGTH ERROR
        R←A,B
         ^
        R←A,[1]B
        ρR
4 4 5
```

   Here, *A* is equivalent to *R*[ι3;;] and *B* to *R*[4;;].

3. If one of the arguments is a scalar, then the scalar element is expanded and applied to the array on an element-by-element basis along the *K*th dimension, as described in Section 2.6.2.

Lamination differs from catenation in that it joins variables along a new coordinate. The *APL* syntax is the same for catenation and lamination. However, the coordinate specification ([*K*]) is fractional in a lamination expression, indicating a position between existing coordinates in which the new coordinate is to be placed. If the two arguments in a lamination function do not have the same dimensions, then at least one of them must be a scalar value or *APL* will not accept the function.

The following examples illustrate some applications of the lamination feature.

```
          ⎕←X←'ABC',[0.5]'DEF'
ABC
DEF
          ⍴X
2 3
          ⎕←X←'ABC',[1.3]'DEF'
AD
BE
CF
          ⍴X
3 2
          ⎕←D←3 2⍴'UVWXYZ'
UV
WX
YZ
          A←3 2⍴'ABCDEF'
          A,[.2]D
AB
CD
EF

UV
WX
YZ
          A,[1.9]D
AB
UV

CD
WX

EF
YZ
          A,[2.3]D
AU
BV

CW
DX

EY
FZ
          A,[.5]'Z'
AB
CD
EF

ZZ
ZZ
ZZ
          'X',[1.5]A
XX
AB

XX
CD

XX
EF
```

```
      'Y',[2.5]A
YA
YB

YC
YD

YE
YF
```

2.7.9 /: Compressing an Array

Function: dyadic compression (/);*R←X/[K]Y*
Argument Domain:
  left: Booleans (0,1)
  right: any
Argument Shape:
  left: scalar or vector
  right: scalar or array
Result Range: same as right argument
Result Shape: array; *ρρR↔ρρY*
Origin-Dependent? no
Take Dimension Argument? yes

The dyadic compression (/) function builds a new vector or array from an old one by specifying the elements to be deleted and those to be preserved. The right argument of the function may be any array. The left argument must be the scalar argument 0 or 1 or a boolean vector (a vector containing only 0's and 1's). The compression function operates as shown below.

```
      A←5 7 9 11 13
      B←1 1 0 1 0
      ⎕←A←B/A
5 7 11
```

Elements in *A* whose positions correspond to the positions of 1's in *B* are preserved; elements corresponding to 0's in *B* are dropped. Because only 0's and 1's are valid values for *B*, the number of elements in the resulting array can be expressed as *+/B*. If *B* contains only 1's, all elements of *A* are preserved; if *B* contains only 0's, the result is the empty vector.

The lengths of *A* and *B* must generally be the same. However, if *A* is of length 1, it will automatically be extended to the length of *B*; if *B* is of length 1, it will be extended to the length of *A*. Thus:

```
      A←5 7 9 11 13
      B←1 1 0 1 0
      B/5
5 5 5
      1/A
5 7 9 11 13
      0/A
                              (APL outputs a blank line.)
```

The expression *0/A* produces the empty vector, because all elements of *A* are dropped.

As discussed in Section 2.7.2, a compression function may also be specified for one particular coordinate of a multi-dimensional array by including the coordinate number in brackets. For a matrix, compression along the first coordinate may cause certain rows to be omitted; compression along the second coordinate may cause columns to be dropped. The result in all cases is a matrix. Several examples of array compression are included below. These examples also illustrate the defaults which *APL* supplies when the coordinate number is omitted from the function.

```
        A←3 4⍴⍳12
        A
 1   2   3   4
 5   6   7   8
 9   10  11  12
        1 0 1/[1]A
 1   2   3   4
 9   10  11  12
        1 0 1 0/[2]A
 1   3
 5   7
 9   11
        ⍴0/A
3 0
        X←2 3⍴⍳6
        0 1 1/X
 2   3
 5   6                                    (Compress along last dimension)
        1 0⌿X
 1   2   3
                                          (Compress along first dimension)
```

The shape of the result of a compression function can be expressed as follows: if *R←B/[K]A*, then ⍴⍴*R*↔⍴⍴*A*.

2.7.10 \: Expanding an Array

Function: dyadic expansion (\); *R*←*X*\[*K*]*Y*
Argument Domain:
  left: Booleans (0,1)
  right: any
Argument Shape:
  left: scalar or vector
  right: scalar or array
Result Range: same as right argument
Result Shape: array; ⍴⍴*R*↔⍴⍴*Y*
Origin-Dependent? no
Take Dimension Argument? yes

The dyadic expansion (\) function builds a new vector or array by expanding the elements of another array into a new format. Expansion is the converse of compression (see Section 2.7.9). The right argument of the function may be any array. The left argument must be the scalar value 0 or 1 or a boolean vector containing only 0's and 1's. The expansion function operates as shown below.

```
      A←⍳3
      V←1 0 1 0 1
      V\A
1 0 2 0 3
      V\'APL'
A P L
```

The function expands the elements of *A* into the format specified by *V*. The values of *A* are inserted in positions corresponding to the occurrence of 1's in *V*. For numeric values, zeroes are inserted in positions corresponding to 0's in the boolean vector. If the right argument is a character string, as in the second example above, spaces are used rather than zeroes.

The number of 1's in the boolean vector must generally be the same as the number of values in the array included as the right argument. Thus, +/*V* must be equivalent to ⍴*A*. However, a scalar boolean value as the left argument of the function is extended as shown below.

```
      1 0 1\5
5 0 5
```

As discussed in Section 2.7.2, an expansion function may also be specified for one particular coordinate of a multi-dimensional array by including the coordinate number in brackets. Several examples of array expansion are included below. These examples also illustrate the defaults which *APL* supplies when the coordinate number is omitted from the function.

```
      ⎕←A←2 3⍴⍳6
 1  2  3
 4  5  6
      1 0 1\[1]A
 1  2  3
 0  0  0
 4  5  6
      1 0 1 1\[2]A
 1  0  2  3
 4  0  5  6
      0 0 0\⍳0
0 0 0



      ⎕←A←0 0 0\''

      ⍴A
3
      X
⋆THISISAN
EXPANSION
EXAMPLE⋆⋆
      ⍴X
3 9
      V←1 1 1 1 1 0 1 1 0 1 1
      V\X
⋆THIS IS AN
EXPAN SI ON
EXAMP LE ⋆⋆


      1 0 1 1⍀X
⋆THISISAN

EXPANSION
EXAMPLE⋆⋆
```

2.7.11 ↑: Taking Array Elements

Function: dyadic take (↑); R←X↑Y
Argument Domain:
left: integers
right: any
Argument Shape:
left: scalar or vector; (⍴X)↔⍴⍴Y
right: any
Result Range: same as right argument
Result Shape: array; ⍴R↔|X
Origin-Dependent? no
Take Dimension Argument? no

The dyadic take (↑) function builds a new vector or array by taking a specified number of elements from an existing array. The right argument of the function may be any array. The left argument can be a one element array or scalar, or a vector. The number of elements in the left argument must be equal the number of dimensions in the right argument. A scalar is treated as a one element vector.

The take function operates as shown below.

```
          V←1 2 3 4
          ⎕←X←2↑V
    1 2
```

This expression takes the first two elements of *V* and forms a new vector. If the value of the scalar is greater than the number of elements in *V*, then the resulting vector, *X*, is extended so that its length is the value of the scalar. As shown below, zeroes are used to extend a numeric vector and blanks are used to extend a character vector.

```
          2↑⍳3
    1 2
          4↑⍳3
    1 2 3 0
          ⍴⎕←10↑'APL11'
    APL11
    10
```

In the expression *R*←*S*↑*V*, if *S* is positive, then *R* consists of the first *S* elements of *V*. If *S* is negative, then *R* contains the last |*S* elements of *V*. If |*S* is greater than the number of elements in *V* ((|*S*)>⍴*V*), then zeroes or spaces are inserted in *R* before or after the values of *V*. Examples of the effects of negative scalars are included below.

```
          ¯6↑12 24 36 48
    0 0 12 24 36 48
          ¯10↑'FOO45'
         FOO45
          ¯2↑⍳3
    2 3
```

A take function may also be specified for a multi-dimensional array. In this case, the left argument of the function must be a vector containing one element for each dimension of the array. In the expression *S*↑*V*, the value of *S*[1] indicates the number of elements to be taken along the first coordinate of *V*, and so on. Several examples of taking an array are included in the following.

```
      A←⎕←3 5⍴⍳15
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
      2 ¯2↑A
4   5
9   10
      ¯4 2↑A
0   0
1   2
6   7
11  12
```

The shape of the result of the take function can be expressed as follows: if *R*←*A*↑*B*, and ⍴⍴*R*←→⍴⍴*B*, then ⍴*R*←→|*A*.

2.7.12 ↓: Dropping Array Elements

Function: dyadic drop (↓); R←X↓Y
Argument Domain:
  left: integers
  right: any
Argument Shape:
  left: scalar or vector; (ρX)↔ρρ*Y*
  right: any
Result Range: same as right argument
Result Shape: array; ρR↔(ρ*Y*-|*X*)
Origin-Dependent? no
Take Dimension Argument? no

The dyadic drop (↓) function builds a new vector or array by dropping a specified number of elements from an existing array. The right argument of the function may be any array. The shape requirements for the arguments are the same as for the take function (2.7.11).

The drop function operates as shown below.

```
      ⎕←V←⍳5
1 2 3 4 5
      ⎕←X←2↓V
3 4 5
```

This expression drops the first two elements of *V* and forms a new vector with the remaining elements. If the value of the scalar is greater than the number of elements in *V*, then the result is the null vector.

The drop function handles negative scalar values in much the same way as take. The function *R*←¯*S*↑*V* causes the last |*S* elements of *V* to be omitted from vector *R*. The following is an example of the effect of a negative scalar on a drop function.

```
      ¯2↓⍳5
1 2 3
```

A drop function may also be specified for a multi-dimensional array. In this case, the left argument of the function must be a vector containing one element for each dimension of the array. In the expression *S*↓*V*, the value of *S*[1] indicates the number of elements to be dropped along the first coordinate of *V*, and so on. The examples below illustrate the use of drop in multi-dimensional arrays and demonstrate the construction of identical arrays by means of alternative take and drop functions.

```
      ⎕←A←3 5⍴⍳15
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
      2 ¯2↑A
4   5
9   10
      ¯1 3↓A
4   5
9   10
```

The following example illustrates the use of drop in a character array.

```
          ⎕←A←2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
          1 2 3↓A
X
          1 0 2↓A
OP
ST
WX
          ⍴2 3 4↓A
0 0 0
```

2.7.13 ⍉: Transposing the Dimensions of an Array

Function: monadic transpose (⍉);*R*←⍉*Y*
Argument Domain:
  left: -
  right: any
Argument Shape:
  left: -
  right: any
Result Range: same as argument
Result Shape: array; ⍴*R*←→⌽⍴Y
Origin-Dependent? no
Take Dimension Argument? no

The monadic transpose (⍉) function interchanges the dimensions of any array. For a matrix, this function has the effect of exchanging the rows and columns. The symbol ⍉ is formed by overstriking the circle ○ with the backslash (\) character. The following is an example of a simple matrix transposition.

```
      ⎕←A←2 3⍴⍳6
 1  2  3
 4  5  6
      ⍴A
2 3
      A←⍉A
      A
 1  4
 2  5
 3  6
      ⍴A
3 2
```

Note that the ⍉ function changes the shape of the array from 2-by-3 to 3-by-2. For a matrix, the monadic transpose function often performs the same operation as the dyadic transpose function described in Section 2.7.14.

A transposition for a 3-dimensional array is shown below.

```
      X←2 2 2⍴⍳8
      X
 1  2
 3  4

 5  6
 7  8
      ⍉X
 1  5
 3  7

 2  6
 4  8
```

If the right argument of the monadic transpose function is a vector *A*, then ⍉*A*←→*A*. The shape of the result of the monadic transpose function can be expressed as follows: if *R*←⍉*A*, then ⍴⍴*R*←→⍴⍴*A* and ⍴*R*←→⌽⍴*A*.

2.7.14 (⍉): Transposing an Array

Function: dyadic transpose (⍉);*R*←*X*⍉[*K*]*Y*
Argument Domain:
  left: non-negative integers
  right: any
Argument Shape:
  left: vector; (⍴*X*)↔⍴⍴*Y*
  right: any
Result Range: same as right argument
Result Shape: array
Origin-Dependent? yes, left argument
Take Dimension Argument? yes

The dyadic transpose (⍉) function restructures an array of any shape. It can be considered an extended version of the monadic transpose and in some cases has the same effect on a matrix as the monadic function - for example:

```
        ⎕←A←2 3⍴⍳6
  1  2  3
  4  5  6
        ⍉A
  1  4
  2  5
  3  6
        2 1⍉A
  1  4
  2  5
  3  6
```

The right argument of a dyadic or scalar ⍉ function may be any array. The left argument must be a vector containing one element for each of the dimensions of the array to be transposed. The shape of the vector expresses the rank of the right argument. For the function *V*⍉*A*, this can be expressed as the following: ⍴*V* must equal ⍴⍴*A*. Thus *V* must have two elements if *A* is a matrix, three if *A* is a 3-dimensional array, and so on. A scalar argument is treated as a one element vector.

The dyadic transpose function rearranges the dimensions of an array by transposing them according to the vector provided as the left argument. The following illustrates an existing array and the way in which a new array is developed by transposing its dimensions.

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

  13  14  15  16
  17  18  19  20
  21  22  23  24
        ⍴A
  2 3 4
```

If the following *APL* statement is specified:

```
    3  1  2⍉A
```

then the vector supplied as the left argument is used to rearrange the dimensions of *A* as shown below. The elements of the vector determine the new positions to which the elements of ⍴*R* are to be moved.

(left argument) 3 1 2

(Shape of *A*)

(Shape of result)

$$\begin{tikzcd} 2 \arrow[drr] & 3 \arrow[dl] & 4 \arrow[dl] \\ 3 & 4 & 2 \end{tikzcd}$$

The new array has the structure shown below.

```
      R←3 1 2⍉A
      R
 1   13
 2   14
 3   15
 4   16

 5   17
 6   18
 7   19
 8   20

 9   21
10   22
11   23
12   24
```

The examples included above illustrate the case in which the coordinates of the original array are permuted. In a permutation, all of the coordinate numbers are the same, but they are arranged in a different order; for example, 3 1 2 is a permutation of 1 2 3. In the function *V*⍉*A*, if *V* is a permutation of ⍳⍴⍴*A*, then the following is true. If *K* represents a coordinate of array *A* and the function *R*←*V*⍉*A* is specified, then *R* is an array similar to *A* except that the *K*th coordinate of *A* is the *V*[*K*]th coordinate of *R* and (⍴*R*)[*V*] is equal to ⍴*A*.

In a dyadic transpose function, it is also legal to specify as the left argument a vector which is not a permutation of the coordinates. Two or more of the elements may be identical. Legal values for the elements of the vector must follow these rules:

- Each element of the vector must be a positive integer that is less than or equal to the rank of the right argument (*V*∊⍳⍴⍴*A*).

- All of the positive integers up to the largest in the vector must appear in the left argument ((⍳⌈/*V*)∊*V*).
  In a 3-dimensional array with shape 2 3 4, valid vectors include 3 1 2, 1 1 1, 1 1 2, 2 2 1, 2 1 1, 1 2 2, 1 2 1, and 2 1 2. Invalid vectors are 3 1 1, (2 is missing) 2 2 2, (1 is missing), and 2 3 2 (1 is missing).

Incomplete vectors have special meaning in the dyadic transpose function. Only particular elements will be selected by the vector, as shown below.

Vector	Selects
1 1 1	Elements whose first, second, and third indices are the same
1 1 2 2 2 1	Elements whose first and second indices are the same
2 1 1 1 2 2	Elements whose second and third indices are the same
1 2 1 2 1 2	Elements whose first and third indices are the same

The elements selected by this vector will be transposed as shown in the examples below. Such dyadic transpositions effectively take slices through the array along different diagonal directions. The first example below obtains the main diagonal of the matrix.

```
      A←2 3⍴⍳6
      A
 1  2  3
 4  5  6
      1 1⍉A
1 5
      A←2 3 4⍴⍳24
      A
 1   2   3   4
 5   6   7   8
 9  10  11  12

 13  14  15  16
 17  18  19  20
 21  22  23  24
      2 1 1⍉A
 1  13
 6  18
 11 23
```

The following examples illustrate dyadic transpositions of a character array.

```
      ⎕←A←2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
      1 1 1⍉A
AR
      1 2 1⍉A
AEI
NRV
      1 1 2⍉A
ABCD
QRST
```

```
      2 3 1⍉A
AEI
MQU

BFJ
NRV

CGK
OSW

DHL
PTX
```

Table 2-5 may be helpful in determining transpositions for a variety of arrays.

Table 2-5
Transpose Definitions

Case	ρR	Definition
R←1⍉V	ρV	R←V
R←1 2⍉M	ρM	R←M
R←2 1⍉M	(ρM)[2 1]	R[I;J]←M[J;I]
R←1 1⍉M	⌊/ρM	R[I]←M[I;I]
R←1 2 3⍉A	ρA	R←A
R←1 3 2⍉A	(ρA)[1 3 2]	R[I;J;K]←A[I;K;J]
R←2 3 1⍉A	(ρA)[3 1 2]	R[I;J;K]←A[J;K;I]
R←3 1 2⍉A	(ρA)(2 3 1]	R[I;J;K]←A[K;I;J]
R←1 1 2⍉A	(⌊/(ρA)[1 2]),(ρA)[3]	R[I;J]←A[I;I;J]
R←1 2 1⍉A	(⌊/(ρA)[1 3]),(ρA)[2]	R[I;J]←A[I;J;I]
R←2 1 1⍉A	(⌊/(ρA)[2 3]),(ρA)[1]	R[I;J]←A[J;I;I]
R←1 1 1⍉A	⌊/ρA	R[I]←A[I;I;I]

2.7.15 ⌽: Reversing an Array

Function: monadic reverse (⌽); *R*←⌽[*K*]*Y*
Argument Domain:
  left: –
  right: any
Argument Shape:
  left: –
  right: scalar or array dimension
Result Range: same as argument
Result Shape: array; ⍴*R*↔⍴*Y*
Origin-Dependent? no
Take Dimension Argument? yes

The monadic reverse (⌽) function is used to reverse a vector or the elements of one coordinate of a multi-dimensional array. The symbol ⌽ is formed by overstriking the circle ○ with the vertical line (|) used for absolute value. The reverse function differs from transpose in that it changes the order of an array, not its structure. The following is an example of reversing a vector.

```
      ⌽⍳5
5 4 3 2 1
```

As discussed in Section 2.7.2, the reverse function may also be specified for one particular coordinate of a multi-dimensional array by including the coordinate number in brackets. Several examples of array reversal are included below. These examples also illustrate the defaults which *APL* supplies when the coordinate number is omitted from the function.

```
      ⎕←A←2 4⍴⍳8
1  2  3  4
5  6  7  8
      ⌽[1]A
5  6  7  8
1  2  3  4
      ⌽[2]A
4  3  2  1
8  7  6  5
      ⌽A
4  3  2  1
8  7  6  5                (Reverse last dimension)
      ⊖A
5  6  7  8                (Reverse first dimension)
1  2  3  4
```

It is possible to reverse a matrix in both of its dimensions. This is not the same as transposing the matrix, as is indicated in the examples that follow.

```
      ⎕←X←2 3⍴⍳6
1  2  3
4  5  6
      ⌽⌽[1]X
6  5  4
3  2  1
      ⍉X
1  4
2  5
3  6
```

2.7.16 ⌽: Rotating an Array

Function: dyadic rotation (⌽); *R*←*X*⌽[*K*]*Y*
Argument Domain:
  left: integers
  right: any
Argument Shape:
  left: scalar or vector
  right: scalar or array
Result Range: same as right argument
Result Shape: array; ⍴*R*↔⍴*Y*
Origin-Dependent? no
Take Dimension Argument? yes

The dyadic rotate (⌽) function is used to rotate an array by a specified number of places. The right argument of the function may be any array. The left argument may be a scalar or a vector. The following example illustrates two rotations of a vector; note that a positive rotation causes a left shift and a negative rotation causes a right shift.

```
      3⌽⍳5
4 5 1 2 3
      ¯3⌽⍳5
3 4 5 1 2
```

If a vector is being rotated, the left argument of the function must be a scalar or a 1-element vector.

A rotation function may also be specified for a multi-dimensional array by including the coordinate number in brackets. If a multi-dimensional is rotated, the left argument of the function must be a scalar, a single-element vector, or an array whose elements correspond to the dimensions of the array to be rotated, with the dimension being rotated omitted from the array. For example, if a matrix containing three rows and four columns is rotated and a vector is included as the left argument of the function, that vector must contain three elements if the rows of the matrix are rotated and four elements if the columns are rotated. A scalar left argument will be extended to an array of proper shape. This is illustrated in the following examples.

```
      X←3 4⍴'ABCDEFGHIJKL'


      X
ABCD
EFGH
IJKL
      0 1 2⌽X
ABCD
FGHE
KLIJ
      1 1 2 3⌽[1]X
EFKD
IJCH
ABGL
```

Negative values in the left argument are handled as shown below. These examples also illustrate the defaults which *APL* supplies when the coordinate number is omitted from the function.

```
          ⎕←A←3 5⍴'ABCDEFGHIJKLMNO'
    ABCDE
    FGHIJ
    KLMNO
          1 ¯1 2 2 2⊖A
    FLMNO
    KBCDE
    AGHIJ
          5 2 ¯1⌽A
    ABCDE
    HIJFG
    OKLMN
```

The shape of the result of a rotation function can be expressed as follows: ⍴*R*←→⍴*B* if *R*←*A*⌽ [*K*] *B*, then ⍴*A*←→ (*K* ≠ ⍳⍴⍴*B*)/⍴*B*.

2.7.17 ⍋: Sorting an Array in Ascending Order

Function: monadic grade-up (⍋); *R*←⍋[*K*]*Y*
Argument Domain:
  left: -
  right: any
Argument Shape:
  left: -
  right: scalar or array; (⍴⍴*Y*)≤2
Result Range: non-negative integers
Result Shape: vector; ⍴*R*↔(⍴*Y*) [*K*]
Origin-Dependent? yes
Take Dimension Argument? yes

The monadic grade up (⍋) function aids in sorting an array in ascending order. The grade up function is extended to operate on matrices as well as vectors. The argument of the function represents the scalar, vector, or matrix whose elements are to be recorded. The array being sorted may contain either numeric or character elements.

If two or more elements of the array being sorted have the same value, then the order of the elements is determined by their relative positions in the original array.

The symbol ⍋ is formed by overstriking the delta (Δ) character with the vertical line (|) used for absolute value.

The following example illustrates the use of the grade up function in sorting the elements of a vector.

```
      A←2 9 7 4 3 10 4
      ⎕←B←⍋A
1 5 7 4 3 2 6
      A[B]
2 3 4 4 7 9 10
```

Note that the grade up function does not actually sort the vector. It creates a permutation vector of the index numbers of the elements; this vector is then used to sort the original vector, as shown in the examples above.

The current setting of the index origin determines the index values returned by the grade up function. An example of this is included below.

```
      X←3 7 5 1 2
      ⍋X
4 5 1 3 2
      )ORIGIN 0
WAS 1
      ⍋X
3 4 0 2 1
```

As discussed in Section 2.7.2, the grade up function may also be specified for one particular coordinate of a matrix by including the coordinate number in brackets. *APL* supplies defaults when the coordinate number is omitted from the function, as shown in the examples below.

If the array to be sorted by the grade up function is a matrix, the simplest operation causes each row of the matrix to be treated as a string. The result of the grade up function is a vector whose length is equal to the number of rows in the matrix. The following examples illustrate the sorting of two matrices - one character and one numeric.

```
      A
STEVE
SAM
STAN
      ρA
3  5
      ⍋A                    (Sort along last dimension)
2  3  1
      A[⍋A;]
SAM
STAN
STEVE

      B
3  2  1  5  0
3  1  9  7  0
3  2  0  8  0
      ρB
3  5
      ⍋B                    (Sort along last dimension)
2  3  1
      B[⍋B;]
3  1  9  7  0
3  2  0  8  0
3  2  1  7  0
```

The examples included above cause the matrix to be sorted by rows; however, by subscripting the function, as shown below, it is possible to sort on the basis of columns.

```
      A
SSS
TAT
EMA
V N
E


      ρA
5  3
      ⍋[1]A                 (Sort along first dimension)
2  3  1
      A[;⍋[1]A]
SSS
ATT
MAE
 NV
  E
```

2.7.18 ⍒: Sorting an Array in Descending Order

Function: monadic grade-down (⍒); *R*←⍒[*K*]*Y*
Argument Domain:
  left: -
  right: any
Argument Shape:
  left: -
  right: scalar or array; (⍴⍴*Y*)≤2
Result Range: non-negative integers
Result Shape: vector; ⍴*R*↔(⍴*Y*)[*K*]
Origin-Dependent? yes
Take Dimension Argument? yes

The monadic grade down (⍒) function aids in sorting an array in descending order. The grade down function is extended to operate on matrices as well as vectors. The right argument of the function represents a scalar, vector, or matrix whose elements are to be reordered. The array being sorted may contain either numeric or character elements.

Duplicate values are handled exactly as in the grade up function; the order of such elements is determined by their relative positions in the original vector. As in the case of grade up, the index origin setting determines the values returned.

The symbol ⍒ is formed by overstriking the del (∇) character with the vertical line (|). Following are several examples of using the grade down function to sort the elements of a vector.

```
      A
5 7 3 1 2 4 2
      ⎕←B←⍒A
2 1 6 3 7 5 4
      A[B]
7 5 4 3 2 2 1
      A[⍒A←5 7 3 1 2 4 2]
7 5 4 3 2 2 1
```

Like the grade up function, ⍒ creates a permutation vector that can be used to sort the original vector. The last two examples above illustrate the way in which grade down and indexing operations can be performed together. The grade down function operates on matrices in the same way as that described for grade up, except that it sorts the elements of the matrix in descending order.

2.7.19 ?: Rolling Random Integers

Function: monadic scalar roll (*?*); *R←?Y*
Argument Domain:
  left: -
  right: non-negative integers
Argument Shape:
  left: -
  right: any
Result Range: non-negative integers (0≤*Y*)
Result Shape: array; ρ*R*↔ρ*Y*
Origin-Dependent? yes
Take Dimension Argument? no

The monadic roll (*?*) function is used to generate an array of independent random integers. Roll is actually a scalar rather than a mixed function, but it is presented here because it is closely related to the dyadic deal function (Section 2.7.22).

The argument of the roll function is an array of positive integers. The shape of the array produced by the expression *R←?A* is the same as the shape of *A*. If the current index origin setting is 1, each element in *R* is a random integer in range 1 through the value of the corresponding element in *A*. If the origin is 0, the range is 0 through the value of the corresponding element in *A* minus 1. An example of a roll function performed on a vector is included below.

```
      ?5 10 15 20 25
 3 9 4 13 4
```

Note that the number 4 was generated twice, once as a random integer in range 1 through 15 and once in range 1 through 25. This can happen because numbers selected by roll are independently random within each range. The term "roll" relates to the analogy between the operation performed by this function and the rolling of several dice. The deal function differs from roll in that it generates a set of random numbers in which no number is selected twice.

2.7.20 ?: Dealing Random Integers

Function: dyadic deal (?); *R←X?Y*
Argument Domain:
left: non-negative integer (*X*≤*Y*)
right: non-negative integer
Argument Shape:
left: scalar
right: scalar
Result Range: non-negative integers (0≤*Y*)
Result Shape: vector; ρ*R*←→,*X*
Origin-Dependent? yes
Take Dimension Argument? no

The dyadic deal (?) function generates a vector of integers randomly selected from another vector; no number may be selected more than once. Unlike the roll function, which can be compared to rolling several dice independently, "deal" refers to the analogy of dealing a number of cards from a deck containing no duplicates.

Both arguments of this function must be positive scalars or single-element arrays. The length of the vector produced by the expression *R←A?B* is the same as the value of *A*. *A* identifies the number of elements to be selected randomly from the values in range 1 through *B* if the index origin is 1, or 0 through *B* minus 1 if the index origin is 0. The value of A must be less than or equal to the value of B (A≤B). Several examples of the deal function are included below.

```
      5?5
1 2 4 3 5
      5?1E30
6.190632307E+29 7.963339536E+29 2.944321859E+29 7.939762066E+29
      1.501112165E+29
      )ORIGIN 0
WAS 1
      5?5
3 1 4 0 2
```

Note in the first and last examples that if the values of the two arguments are the same (*A*←→*B*), then the resulting vector is a permutation of ι*B*.

2.7.21 ⊤: Constructing a Character String

Function: monadic quote (⊤); *R*←⊤*Y*
Argument Domain:
  left: numbers
  right: numbers
Argument Shape:
  left: -
  right: any
Result Range: null *or characters*
Result Shape: vector
Origin-Dependent? no
Take Dimension Argument? no

The monadic quote (⊤) function converts numeric values to character strings and may be helpful in preparing text to be processed by the execute function. The argument may be a scalar or an array and may have numeric or character values. If the argument is numeric, it will be converted to a character string as shown below.

```
       X←1 2 3 4
       ⎕←Y←⊤X
 1 2 3 4
       ⍴Y
 7
       A←2 3⍴⍳6
      ⎕←B←⊤A
  1  2  3
  4  5  6
      ⍴B
 18
      '123456'∊B
 1 1 1 1 1 1
      (⍳6)∊B
 0 0 0 0 0 0
```

In the second example above, array *A* is converted to a 20-character vector (spaces output by *APL* are included in the size) in which the character representations of 1 through 6 are members but the corresponding numeric values are not.

If the argument is already a character string, then special processing is performed to determine whether or not the string represents *APL* identifier (i.e., a variable or function name). If the character string is not defined as an identifier, ⊤*A* returns the null vector. If *A* is defined as a variable, ⊤*A* returns the value of the variable, converted to a character string. If *A* is defined as a function, ⊤*A* returns the lines of the function definition, separated by pairs of carriage return/line feed characters. Examples of these uses of the quote function are included below.

```
       A←1 2 3 4
       ⎕←B←⊤A
 1 2 3 4
       ⍴B
 7
       ⍴⊤'DDFDFDFDFD'
 0
```

```
          ∇Z←A G B
    [1]   Z←(3×A)+4×B
    [2]   Z←Z*2
    [3]   ∇
          2 G 1
    100
          ⎕←C←⍕'G'
          ∇ Z←A G B
    Z←(3×A)+4×B
    Z←Z*2
          ∇

          ⍴C
    48
          )ERASE G
          2 G 1

    SYNTAX ERROR

          2 G 1
            ↑
          ⍎C

          2 G 1
    100
```

Note that the definition of function *G* is effectively restored by the use of the character string that represented this function in an execute operation.

2.7.22 ⊤: Representing a Number in Another Base

Function: dyadic encode (⊤); *R*←*X*⊤*Y*
Argument Domain:
  left: numbers
  right: numbers
Argument Shape:
  left: numbers
  right: numbers
Result Range: numbers
Result Shape: array; ρ*R*↔(ρ*X*),ρ*Y*
Origin-Dependent? no
Take Dimension Argument? no

The dyadic encode (⊤) function is used to represent a scalar or an array in any number system. It is sometimes called the representation function. The right argument identifies the scalar or array to be translated. The left argument is a vector or scalar that represents the number base in which the value is to be expressed; the vector contains one element for each column of the representation. For example, to encode the decimal value 7 in four columns of binary representation, the following function may be specified.

```
      2 2 2 2⊤7
0 1 1 1
```

It is often useful to specify mixed bases for the number to be represented. The encode function can be used to express some number of inches in miles, yards, feet, and inches, or some number of milliseconds in days, hours, minutes, seconds, and milliseconds. The following are examples of these and other similar situations.

```
      0 1760 3 12⊤273125
4 546 2 5                             (miles, yards, feet, inches)
      0 24 60 60 1000⊤719732523
8 7 55 32 448                         (days, hours, minutes, seconds,
      0 4 2 2 16 3 120⊤100001         milliseconds)
1 0 0 1 5 2 41                        (gallons, quarts, pints, cups,
      0 3 320 5.5 3 12⊤100001         tablespoons, teaspoons, drops)
0 1 184 5 2 5                         (leagues, miles, rods, yards,
      0 12 8 3 20⊤100001              feet, inches)
17 4 2 2 1                            (pounds, ounces, drams, scruples, grains)
```

In the expression *A*⊤*B*, *A* is the representation rule to be applied to *B*. Each element of the vector *A* is defined in terms of the element immediately to its left. Thus, in encoding a number as miles, yards, feet, and inches, the following elements are specified from right to left.

- 12 inches in 1 foot
- 3 feet in 1 yard
- 1760 yards in 1 mile

A miles specification is desired, but is not being defined in terms of another quantity, so 0 is inserted in the miles column, as follows:

```
      0 1760 3 12⊤273125
4 546 2 5
```

The following examples of base 3 conversions demonstrate the specification of different numbers of columns in the rule vector and illustrate the way in which negative numbers are encoded.

```
          3 3 3⊤17
    1 2 2
          3 3⊤7
    2 1
          3 3 3⊤¯17
    1 0 1
```

Another useful application of the encode function is shown below. Here the integer and fractional portions of a number are returned.

```
          X←823.75
          0 1⊤X
    823 0.75
```

An encode function may also be specified for vectors and multidimensional arrays. The shape of the result of the function *R*←*A*⊤*B* is always (ρ*A*),ρ*B* or the same as the outer product. Examples of encoded arrays are included below.

```
          ⎕←A←⍉3 2⍴2 3
    2  2  2
    3  3  3
          ⎕←B←A⊤5 2
    1  0
    1  0
    1  0

    2  2
    2  2
    2  2
          ρB
   2 3 2
          ⎕←C←2 2⍴865 429 103 692
    865  429
    103  692
          10 10 10⊤C
    8  4
    1  6

    6  2
    0  9

    5  9
    3  2
```

2.7.23 ⊥: Decoding a Number Representation

Function: dyadic decode (⊥); *R←X⊥Y*
Argument Domain:
  left: numbers
  right: numbers
Argument Shape:
  left: any
  right: any
Result Range: numbers
Result Shape: array
Origin-Dependent? no
Take Dimension Argument? no

The dyadic decode (⊥) function reduces a representation in a number system to a value. It is the converse of the encode function (Section 2.7.19) and is sometimes called the base value function. Equivalent examples of the two functions as they operate on a quantity expressed in yards, feet, and inches are shown below.

```
      1760 3 12⊤63
1 2 3
      1760 3 12⊥1 2 3
63
```

The functions *A⊤B* and *A⊥B* differ only in the values included as *B*; *A* expresses the number base in both cases.

The number of elements in *A* and *B* must generally be the same; element 2 in *A* expresses the base in which element 2 in *B* is encoded, and so on. However, if *A* is a scalar or a single-element array, it is extended so that its length is the same as that of *B*. For example, the following function has the effect of producing the base 10 value of the base 8 number 3777.

```
      8⊥3 7 7 7
2047
```

The decode function may be viewed as a form of inner product. The following illustrates two equivalent functions.

```
      A←1760 3 12
      B←1 2 3
      A⊥B
63
      36 12 1+.×B
63
```

The following are several additional decode examples:

```
      2⊥1 0 1 0
10
      8⊥1 4 4
100
      ¯2⊥1 1 0 0
¯4
      1 4 2 4 2⊥1 2 1 2 1
109
```

(number of pints in bushel, 2 pecks, 1 gallon, 2 quarts, 1 pint)

A decode function may also be specified for multi-dimensional arrays. The function *A*⊥*B* is equal to *W*+.×*B* where *W* is the weighting vector given by *W*[⍴*A*]←→1 and *W*[(-*N*)+⍴*A*]←→*A*[(-*N*)+1+⍴*A*]×*W*[(-*N*)+1+⍴*A*]. The value of *A*[1] is thus irrelevant.

The arrays specified as the arguments of the decode function must conform according to the rules specified for inner products in Section 2.8.3. As with the inner product function, if neither decode argument is a scalar, then the number of elements in the last coordinate of the left argument must equal the number of elements in the first coordinate of the right argument, or either of these coordinates must contain exactly one element. In general, if the left argument (*A*) is a vector and *B* is the right argument, then the result is *W*+.×*B* where *W*[*I*]←×/*I*↓*A*. If *A* is a scalar or a vector of equal elements, the result is the decimal value of the right argument in base *A*.

Several examples of decode functions that use arrays are provided below.

```
          ⎕←A←⍉3 2⍴2 3
    2  2  2
    3  3  3
          ⎕←B←3 2⍴1 0 0 1 1 0
    1  0
    0  1
    1  0
          A⊥B
    5   2
    10  3
          ⎕←W←2 3⍴4 2 1 9 3 1
    4  2  1
    9  3  1
          W+.×B
    5   2
    10  3
```

2.7.24 ⍎: Executing a Character String

Function:	monadic execute (⍎); *R←⍎Y*
Argument Domain:	
left:	–
right:	characters
Argument Shape:	
left:	–
right:	vector
Result Range:	characters or numbers
Result Shape:	scalar or array
Origin-Dependent?	no
Take Dimension Argument?	no

The monadic execute or unquote (⍎) function is used to execute a character string as an *APL* statement. The scalar or vector included as the right argument of the function is evaluated as a character string to be executed by *APL*. An example of this function is shown below.

```
          ⎕←A←⍎')VARS'
  A         B          C          D          I          K          M          MSG        N
```

The effect of this example is to execute the *)VARS* system command (see Section 5.3.1) and thus obtain a display of the global variables available in the user's workspace.

The right argument of the ⍎ function may be a scalar or a character vector. If the scalar value *A* is numeric, then the value of ⍎*A* is equivalent to *A*. If *A* is a character scalar or vector, it is evaluated exactly as if it were quad input from the terminal. Carriage return/line feed characters in *A* are treated as *APL* statement separators, just as they would be in input from the terminal, so multiple line executes are allowed. The result of the expression *R*←⍎*A* is the value of the last statement evaluated in *A*. If the last statement has no value (e.g., *R*←⍎''), *R* is the null vector.

Errors encountered in the character string processed by the execute function are handled exactly as if they occurred in statements entered from the terminal. If an error is encountered while evaluating the execute string, an error message is output and the segment of the execute string currently being evaluated is displayed. (Output may be suppressed by ⌶-beam 16, described in Section 4.3.2.) If an error occurs, no further evaluation of the string is performed, and ⍎*A* returns a null array whose shape is 0 *E*, where *E* is a number indicating the error that was encountered. Appendix D contains a complete description of all *APL* error conditions.

The execute function is also known as the unquote function, because it strips quotes from the value entered as its argument. Other uses of this function besides the execution of system commands include the following:

- Function definition (line editing commands are not permitted)
- Conversion of vectors of characters representing numeric constants into numeric values
- Specification of an *APL* name as an argument to a function, rather than the value of that name

The examples included below illustrate the use of ε in function definition, the execution of system commands, and the evaluation of *APL* statements.

```
      2+2
4
      ε'2+2'
4
      A←'∇Z←F
Z←B+3
Z←Z×Z
∇'
      B←4
      F
VALUE ERROR
      F
      ∧
      C←εA
      ⍴C
0
      F
49
      D←ε'5+4
3+2
6'
9
5
      D
6
      E
VALUE ERROR
      E
      ∧
      E←ε'5+5
3+2,
4'
10
SYNTAX ERROR
3+2,
   ∧
      ⍴E
0 7

      ∇H
[1]   'THIS IS HARD TO BELIEVE'
[2]   Z←ε')SAVE THISWS'
[3]   'WHEN LOADED RESUMES AFTER EXECUTE AUTOMATICALLY'
[4]   ∇
      H
THIS IS HARD TO BELIEVE
WHEN LOADED RESUMES AFTER EXECUTE AUTOMATICALLY

      )LOAD THISWS
WHEN LOADED RESUMES AFTER EXECUTE AUTOMATICALLY

      ε'A←5'
5
      ⎕←ε'A←5'
5
      B←ε''
      ⍴B
0
```

2.7.25 ∊: Determining the Members of an Array

Function: dyadic membership (∊); *R*←*X*∊*Y*
Argument Domain:
  left: any
  right: any
Argument Shape:
  left: any
  right: any
Result Range: Booleans (0,1)
Result Shape: array; ⍴*R*←⍴*Y*
Origin-Dependent? no
Take Dimension Argument? no

The dyadic membership (∊) function is a set function that is used to determine whether or not particular elements of one array occur as elements of another array. Both arguments of the function *A*∊*B* may be arrays of any dimension; the left argument, *A*, contains the elements for which membership in array *B* is to be determined. The result of the membership function is a boolean array whose shape is the same as that of *A*. The result consists only of 0's and 1's; a 1 indicates that the corresponding element in *A* is a member of array *B*, a 0 that it is not. Following is an example of the use of the ∊ function in analyzing the membership of a vector.

```
      ⎕←A←'ABCDEFGH'∊'HEADED'
1 0 0 1 1 0 0 1
      A/'ABCDEFGH'
ADEH
```

The compression function is helpful here in identifying the particular characters that are members of the vector.

The two arguments of the membership function need not have the same rank, as is illustrated in the first example below.

```
      A←2 3⍴7 8 2 4 6 6
      A∊⍳6
 0  0  1
 1  1  1
      3 4∊'34'
0 0
      3 4∊⍳0
0 0
```

For all arrays *B*, the expression *A*∊*B* is equivalent to *A*∊,*B*.

2.7.26 ∪: Eliminating Duplicate Elements in a Set

Function: monadic elimination (∪); *R*←∪*Y*
Argument Domain:
  left: -
  right: any
Argument Shape:
  left: -
  right: any
Result Range: characters or numbers
Result Shape: vector
Origin-Dependent? no
Take Dimension Argument? no

The monadic elimination (∪) function is a set function that eliminates the duplicate elements in a single set. The argument of the function ∪A may be a numeric or character scalar or an array of any dimension. The result of the function is always a vector, regardless of the shape of the argument, A. The result vector contains only one occurrence of each argument element, even if it occurs multiple times in A.

```
      ∪1 0 1 0 1 0 1 0
1 0
      ∪'INVISIBLE'
INVSBLE
      ∪⍳6
1 2 3 4 5 6
```

2.7.27 ∪: Determining the Union of Two Sets

Function: dyadic union (∪); *R*←*X*∪*Y*
Argument Domain:
  left: any*
  right: any*
Argument Shape:
  left: any
  right: any
Result Range: characters or numbers
Result Shape: vector
Origin-Dependent? no
Take Dimension Argument? no

The dyadic union (∪) function is a set function that concatenates two arguments and creates a vector consisting of the elements of the arguments. The arguments of the function A∪B may be scalars or arrays of any dimension. The result of the function is always a vector, regardless of the shape of the argument. The arguments may be either numeric or character, but both arguments must be the same type or a DOMAIN ERROR will result.

In the union example below, note that duplicate elements from the concatenation of the two arguments are not discarded:

```
      'BARNYARD'∪'YARDARM'
BARNYARDYARDARM
```

The following examples illustrate the use of the union function with a variety of arguments of different shapes. The final example illustrates that the shape of the result is always a vector, even if it consists of a single element.

```
      1 2 3∪4
1 2 3 4
      1 2 3∪0 1 1
1 2 3 0 1 1
      A
 1  2  3

 4  5  6
      1 2 3∪A
1 2 3 1 2 3 4 5 6
      'INVISIBLE'∪''
INVISIBLE
      ⍴'A'∪''
1
```

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

2.7.28 ∩: Determining the Intersection of Two Sets

Function: dyadic intersection (∩); *R←X∩Y*
Argument Domain:
  left: any*
  right: any*
Argument Shape:
  left: any
  right: any
Result Range: characters or numbers
Result Shape: vector
Origin-Dependent? no
Take Dimension Argument? no

The dyadic intersection (∩) function is a set function that determines the elements that the two arguments of the function have in common. Both arguments of the function A∩B may be scalars or arrays of any dimension. The function returns the elements of the left argument, A, that are also in the right argument, B. The arguments may be either numeric or character, but must both be the same type. The result of the function is always a vector, regardless of the shape of the argument.

Note that multiple occurrences of an element in the left argument that also are present in the right argument will appear an equal number of times in the results. This is illustrated in the second example.

```
      10 20 30∩10 30 50 70
10 30
      'MISSOURI'∩'MISSISSIPPI'
MISSI
      (⍳6)∩⍳9
1 2 3 4 5 6
      B
 1  0  1  0
 0  1  0  1
      1∩B
1
```

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

2.7.29 ~: Excluding Set Elements

Function: dyadic exclusion (~); *R←X~Y*
Argument Domain:
  left: any*
  right: any*
Argument Shape:
  left: any
  right: any
Result Range: characters or numbers
Result Shape: vector
Origin-Dependent? no
Take Dimension Argument? no

The dyadic exclusion (~) function is a set function that returns the elements that are in the first argument of the function, but not in the second. Both arguments may be scalars or arrays of any dimension. In the function A~B, the result is a vector consisting of the elements in the left argument, A, that are not also in the right argument, B. The arguments may be either numeric or character, but must both be the same type. The function is always a vector regardless of the shape of the argument.

```
        0 1 2~2 3 4 5
0 1
        'ABCDEF'~'HEADED'
BCF
        'MISSISSIPPI'~'MISSOURI'
PP
```

Note that the exclusion function returns the elements in 'MISSISSIPPI' that are not in 'MISSOURI', but it does not return the elements in 'MISSOURI' that are not in 'MISSISSIPPI'. *APL* effectively crosses out the elements in the left argument that are also in the right argument. The set of elements that remain in the left argument is the result of the function.

When the left argument is null, the function returns the null vector, as shown in the second example below.

```
        'INVISIBLE'~''
INVISIBLE
        ''~'INVISIBLE'
```

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

2.7.30 ⊂: Determining a Proper Subset

Function: dyadic subset (⊂); *R←X⊂Y*
Argument Domain:
  left: any*
  right: any*
Argument Shape:
  left: any
  right: any
Result Range: Booleans (0,1)
Result Shape: 1-element vector
Origin-Dependent? no
Take Dimension Argument? no

The dyadic subset (⊂) function is a set function that determines whether or not the left argument is a subset of the right argument. Both arguments may be scalars or arrays of any dimension. The arguments may be either numeric or character, but must both be the same type.

In the function A⊂B, *APL* determines whether or not all of the elements of the left argument A, are contained in the right argument, B. The result of the subset function is always a single-digit boolean vector (0 or 1), regardless of the shape of the arguments. A value of 1 indicates that A is a proper subset of B; a value of 0 indicates that A is not a subset of B. Several examples of the subset functions are included below.

```
          'MISS'⊂'MISSOURI'

     1

          0 1 2⊂1 2 0 1 3

     1

          0 1⊂10 2 0
     0
          'BLISS'⊂'INVISIBLE'

     1
```

Every occurrence of a distinct element in the left argument must be matched by an occurrence in the right argument. In the last example above, 1 is returned even though the letter S occurs twice in 'BLISS' and only once in 'INVISIBLE'. A match need not be found for every occurrence of an element in the left argument; thus, "SS" in "BLISS" will be matched by "S" in "INVISIBLE".

The subset function can be expressed in terms of the and, compression, ravel, and membership functions as the following: A⊂B←→∧/A∊B.

2.7.30.1 ⊆: Determining a Subset - The description for a subset (⊆) is the same as for a proper subset (⊂), except that now a true result (1) is returned if the arguments contain the same elements.

```
          1 2 3⊂1 2 3
    0
          1 2 3⊆1 2 3
    1
```

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

2.7.31 ⊃: Determining a Strict Superset

Function:	dyadic superset (⊃); *R↔X⊃Y*
Argument Domain:	
left:	any*
right:	any*
Argument Shape:	
left:	any
right:	any
Result Range:	Booleans (0,1)
Result Shape:	1-element vector
Origin-Dependent?	no
Take Dimension Argument?	no

The dyadic superset (⊃) function is a set function that determines whether or not the left argument is a strict superset of the right argument. It is the converse of the subset function described in Section 2.7.30. Both arguments in the superset function may be scalars or arrays of any dimension. The arguments may be either numeric or character, but must both be the same type.

In the function A⊃B, *APL* determines whether or not the left argument, A, contains all of the elements of the right argument, B. As with the subset function, the result of the superset function is always a single-digit boolean vector (0 or 1), regardless of the shape of the arguments A value of 1 indicates that A is a superset of B; a value of 0 indicates that A is not a superset of B.

In the superset function, every occurrence of a distinct element in the right argument must be matched by an occurrence in the left argument; this is illustrated in the examples below.

```
        (⍳9)⊃1 3 5 7 9 11
  0
        1 2 1 2 3 4⊃4 2
  1
        'MISSOURI'⊃'MISS'
  1
        (⍳5)⊃1 2 3 4 5
  0
```

The superset function can be expressed in terms of the and, compression, ravel, and membership functions as the following: ∧/,B∊A.

2.7.31.1 ⊇: Determing a Superset - The description for a superset (⊇) is the same as for a strict superset (⊃), except that now a true result (1) is returned if the arguments contain the same elements.

```
        1 2 3 ⊃1 2 3
  0
        1 2 3 ⊇1 2 3
  1
```

*Both arguments must be either character or numeric; argument types cannot be mixed in the same function.

2.7.32 ⍕: Formatting an Array

Function: monadic format (⍕); *R*←⍕*Y*
Argument Domain:
  left: -
  right: any
Argument Shape:
  left: -
  right: any
Result Range: characters
Result Shape: array
Origin-Dependent? no
Take Dimension Argument? no

The monadic format (⍕) function is used to convert numeric arrays to character arrays. Whereas the right argument of the dyadic form of this function (Section 2.7.33) may only be a numerical array, the right argument of the monadic version may be a scalar or an array of any shape, and the value of the argument may be numeric or character. The symbol ⍕ is formed by overstriking the job character (∘) with the symbol ⊤.

When applied to a scalar or a character array, the result of the format function *R*←⍕*A* is an array identical to *A* - for example:

```
      ⍕')VARS'
)VARS
```

If *A* is a numeric, then the character array represented by *R* will be identical to *A* as it appears when displayed by *APL*. However, the blank characters displayed along with the values of *A* will actually be a part of the new array *R*. The format of a scalar number is always a vector. The following example illustrates the difference between the shapes of a displayed numeric array and a formatted character array.

```
      A←2 4⍴⍳8
      B←⍕A
      A
 1  2  3  4
 5  6  7  8
      ⍴A
2 4
      B
 1  2  3  4
 5  6  7  8
      ⍴B
2 11
      B[;¯1+3×⍳4]
1234
5678
```

2.7.33 ⍕: Formatting a Character Array with Width and Precision

Function: dyadic format (⍕); *R*←*X*⍕*Y*
Argument Domain:
  left: integers
  right: numbers
Argument Shape:
  left: scalar or vector
  right: any
Result Range: characters
Result Shape: array
Origin-Dependent? no
Take Dimension Argument? no

The dyadic format (⍕) function is used when control of output exceeding that available with the monadic format is required by the user. It offers a number of formatting options but does not provide the comprehensive formatting capabilities available with the format (⍕) function described in 2.7.32. The right argument of the dyadic format function may only be a numeric array. The left argument is used to control the format of the result. This argument may be a scalar, a pair of numbers, or a vector whose length is twice the number of columns in the numerical array.

Two numbers are normally supplied as the left argument of the format function. The first specifies the width of a numeric field and the second sets the *precision* of that field. Precision is expressed differently for decimal and scaled or exponential forms of output. The form is determined by the sign of the precision argument. For decimal output, precision is a positive number, expressed as the number of digits to the right of the decimal point. For scaled output, precision is negative and is considered the number of digits in the multiplier. Following are several examples of output using different width and precision specifications.

```
      X
 31.16   0       ¯1.0700
¯15.578  8      ¯235.61
      ⍴X
2 3
      ⎕←Y←12 3⍕X
      31.160       0.000      ¯1.070
     ¯15.578       8.000    ¯235.610
      ⍴Y
2 36
      A←9 2⍕X
      A
    31.16     0.00    ¯1.07
   ¯15.58     8.00  ¯235.61
      ⍴A
2 27
      R←6 0⍕X
      R
    31     0    ¯1
   ¯16     8  ¯236
      ⍴R
2 18
      B←9 ¯2⍕X
      B
  3.1E01   0.0E00  ¯1.1E00
 ¯1.6E01   8.0E00  ¯2.4E02
      ⎕←C←7 ¯1⍕X
  3E01   0E00  ¯1E00
 ¯2E01   8E00  ¯2E02
```

If the width specification is zero or is omitted from the function, *APL* provides a default width such that at least one space is inserted between pairs of numbers. If only one number is provided as the left argument of the function, the number is assumed to represent the precision of the result, not its width. An example is shown below, using array *X* as presented above.

```
      ⍴⎕←2⍕X
  31.16 0.00   ¯1.07
 ¯15.58 8.00 ¯235.61
2 20
```

The user may also specify width and precision arguments for each column of the array to be formatted. Following is an example of column formatting of array *X*.

```
      ⍴⎕←8 0 0 ¯2 8 0⍕X
      31 0.0E00       ¯1
     ¯16 8.0E00     ¯236
2 24
```

A format function may also be specified for a multi-dimensional array and applied to the last coordinate - for example:

```
      ⎕←A←2 2 2⍴⍳8
1  2
3  4

5  6
7  8
      5 2⍕A
1.00 2.00
3.00 4.00

5.00 6.00
7.00 8.00
```

In general, the width specified by the user must be large enough to accommodate the number field. However, *APL* does not require that space be inserted between columns, as is illustrated by the following logical array.

```
      B
 1  0  0
 1  0  1
 1  1  1
      1 0⍕B
100
101
111
```

The dyadic format function provides a powerful facility for formatting tables and providing headings and labels. Following is an example of a table prepared using the format function.

```
      ROWS←5 7⍴'APL    FORTRANCOBOL  BASIC  PLI    '
      COLS←'  USERS  PROGS  SYSTS'
      FORM←5 3⍴A
      (' ',[1]ROWS),COLS,[1]7 0⍕FORM
         USERS  PROGS  SYSTS
APL        112    608     14
FORTRAN    306    588     26
COBOL      596    821     45
BASIC      622    960     30
PLI         18     35      3
```

Note that array *A* contains the data formatted for inclusion in the table.

2.7.34 ⌹: Performing Matrix Inversion

Function: monadic domino (⌹); *R*←⌹*Y*
Argument Domain:
  left: –
  right: numbers
Argument Shape:
  left: –
  right: scalar or array; (⍴⍴*Y*)≤2
Result Range: numbers
Result Shape: array; ⍴*R*←→⍴*Y*
Origin-Dependent? no
Take Dimension Argument? no

The monadic domino (⌹) function inverts a matrix and thus facilitates matrix division and a variety of other matrix operations. The domino symbol, ⌹, is formed by overstriking the quad (⎕) character with the division (÷) symbol. The right argument may be a scalar, a vector, or a matrix. The most useful applications of the domino function include the following:

- finding the inverse of a matrix
- solving sets of linear equations
- determining a least squares solution to an overdetermined set of linear equations

Only the first application is discussed in this section. The dyadic version of the function is described in Section 2.7.28 and is used in performing more sophisticated matrix operations. The monadic inversion function operates as shown below.

```
      B
¯5  2
¯3  1
      ⌹B
 1 ¯2
 3 ¯5
```

```
      ⎕←X←⌹A
 9   ¯36   30
¯36   192 ¯180
 30  ¯180  180
      A+.×X
 1.000000000      ¯7.105427358E¯15  6.661338148E¯15
 3.330669074E¯16  1.000000000       2.442490654E¯15
 2.775557562E¯16 ¯1.554312234E¯15   1.000000000
```

The monadic expression ⌹*X* is equivalent to the dyadic *I*⌹*X*, where *I* is an identity matrix whose order can be described as 1↑⍴*X*. If the argument of the monadic function is a scalar, the expression ⌹*X* is equivalent to ÷*X*.

The argument of the matrix inversion function may be non-square, but the matrix must have at least as many rows as columns. In a non-square situation, the result is a left inverse of the argument. If the matrix has no inverse, a *DOMAIN ERROR* results.

2.7.35 ⌹: Performing Matrix Division

Function: dyadic domino (⌹); *R*←*X*⌹*Y*
Argument Domain:
  left: numbers
  right: numbers
Argument Shape:
  left: scalar or array; (⍴⍴*Y*)≤2
  right: scalar or array; (⍴⍴*Y*)≤2
Result Range: numbers
Result Shape: array
Origin-Dependent? no
Take Dimension Argument? no

The dyadic domino (⌹) function performs more complicated matrix operations than the inversions described in Section 2.7.34 - for example, solving linear equations and finding a least squares solution. Both arguments of the ⌹ function may be scalars, vectors, or matrices. In the expression *X*⌹*Y*, *X* and *Y* must conform, fulfilling all of the conditions described below.

1. *Y* must have a rank of 2 or less.

2. If the dimensions of *Y* are *M* by *N*, then *M*≥*N*.

3. *X* must have a rank of 2 or less and (1↑⍴*Y*) = 1↑⍴*X*.

This implies that for matrices, *X* and *Y* have the same number of rows and the columns of *Y* are linearly independent. If *Z*←*X*⌹*Y*, then ⍴⍴*Z*←→⍴⍴*X* and +/((*Y*+.x*Z*)-*X*)*2 is minimized.

The following example illustrates the use of the matrix division function in solving a set of linear equations. The equations are:

```
  3A+B = 9
  2A-B = 1
```

In expression *X*⌹*Y*, *Y* is a matrix whose values are the coefficients of the equations, and *X* is a vector containing the values 9 1.

```
      X←9 1
      Y←2 2⍴3 1 2 ¯1
      X⌹Y
2 3
```

The result is a vector in which the first element is the value of *A* in the linear equation, and the second is the value of *B*.

The domino function treats scalar arguments as matrices containing one row and one column. The expression *X*⌹*Y* is equivalent to scalar division *X*÷*Y*, except that the operation *0*⌹*0* produces an error condition. If the arguments are vectors, they are treated as matrices with a single column. As mentioned in Section 2.7.35, if *I* is an identify matrix of the same dimension as *X*, then ⌹*X* is equivalent to *I*⌹*X*.

A more general statement of the relationship between the monadic and dyadic forms of the domino function is the following. The expression ⌹*X*, where *X* is a matrix, is equivalent to ((⍳*Y*)∘.=⍳*Y*)⌹*X*, where *Y* is the number of rows in *X*.

Following are several examples of the use of the dyadic domino function, including a least squares solution.

```
      ⎕←A←(2 1⍴2 5),1
 2  1
 5  1
      B←10 19
      ⍴X←B⌹A
2
      A+.×X
10 19
      ⎕←A←(5 1⍴⍳5),1
 1  1
 2  1
 3  1
 4  1
 5  1
      B←2.001 2.998 4.002 4.997 6.01
      ⎕←X←B⌹A
1.0017 0.9965
      B−A+.×X
2.800000029E¯3 ¯1.899999947E¯3 4.00000077E¯4 ¯6.299999899E¯3
      5.000000125E¯3
      ⎕←X←⌹A
¯1.999999993E¯1  ¯1.000000023E¯1   4.557563222E¯10  1.000000032E¯1
                   1.999999979E¯1
 0.799999994       0.500000008      2.000000020E¯1  ¯1.000000043E¯1
                  ¯3.999999999E¯1
      X+.×A
 1.000000000       6.938893904E¯18
¯4.163336342E¯17   1.000000000
```

2.8 OPERATORS

The operators described in this section are *APL* functions that take primitive scalar functions such as + or × as their arguments.

2.8.1 *f*/: Reducing an Array

Function: monadic reduction (f/); *R*←f/[*K*]*Y*
Argument Domain:
  left: -
  right: same as for function f
Argument Shape:
  left: -
  right: vector or array
Result Range: same as for function f
Result Shape: array (ρρ*R*)←→ or ¯1+ρρ*Y*
Origin-Dependent? no
Take Dimension Argument? yes

The monadic reduction (*f*/) operator combines the elements of a vector or the elements along a specified dimension of an array.

The following example illustrates the use of reduction in obtaining a sum, product, maximum, and minimum value for vector *X*.

```
      ⎕←X←⍳6
1 2 3 4 5 6
      +/X
21
      ×/X
720
      ⌈/X
6
      ⌊/X
1
```

The general requirement for reduction is that the function (*f*) to the left of the reduction symbol (/) be a scalar dyadic function (see Section 2.6). If *f*/*V* represents a reduction, then an equivalent form is the following:

```
      V[1]fV[2]f...fV[ρV]
```

where the expression is evaluated from right to left in the conventional way. The result of reducing any vector is a scalar value. If *V* is a scalar or a vector with a single element, then *f*/*V*←→*V*. If *V* is an empty vector, then the result of a reduction is the identity element of the function, if one exists - for example:

```
      +/5 7 8
20
      -/1 6 7
2
      ⌈/7
7
      ×/⍳0
1
```

Table 2-6 summarizes the identity elements for the primitive scalar dyadic functions presented in Section 2.6 that may be returned by the reduction function.

Table 2-6
Identity Elements of Scalar Dyadic Functions

Dyadic Function	Symbol	Identity Element
Plus	+	0
Minus	-	0
Times	×	1
Divide	÷	1
Power	*	1
Residue	|	0
Maximum	⌈	¯1.70141*E*+38
Minimum	⌊	1.70141*E*+38
Logarithm	⍟	None
Out of	!	1
Circle	○	None
And	∧	1
Or	∨	0
Nand	⍲	None
Nor	⍱	None
Less	<	0
Not greater	≤	1
Equal	=	1
Greater or equal	≥	1
Greater	>	0
Not equal	≠	0

A reduction operation may also be specified for one particular coordinate of a multi-dimensional array by including the coordinate number in brackets. The result of reducing an array has a rank that is one less than the rank of the original array. Thus the reduction of a matrix yields a vector, as shown in the examples below. Note that +/[1]*A* operates on the first dimension of *A* and produces a column sum; +/[2]*A* operates on the second dimension and produces a row sum.

The following examples also illustrate the defaults which *APL* supplies when the coordinate number is omitted from the function.

```
      ⎕←A←2 4⍴⍳6
1  2  3  4
5  6  1  2
      +/[2]A
10 14
      +/[1]A
6 8 4 6
```

2.8.2 *f*\: Scanning an Array

Function: monadic scan (f\); *R*←f\[*K*]*Y*
Argument Domain:
  left: -
  right: same as for function f
Argument Shape:
  left: -
  right: vector or array
Result Range: same as for function f
Result Shape: array; ρ*R*←→ρ*Y*
Origin-Dependent? no
Take Dimension Argument? yes

The monadic scan (*f*\) operator is used to derive partial results in calculating the reduction of an array. For example, if *V* is a vector, the expression +\V produces a vector of the partial sums of *V*. An example of this is shown below.

```
      +\3 4 5
3 7 12
```

Here each element of the resulting vector can be considered a reduction of the original vector up to that point. In the resulting vector, the first element is always identical to the first element of the original vector, and the last element is equivalant to a reduction of the entire original vector.

The general requirement for the scan is that the function (*f*) to the left of the scan symbol (\) be a scalar dyadic function (see Section 2.6). Following are several other examples of the use of the scan function.

```
      ×\2 2 2
2 4 8
      ∨\0 1 0 0
0 1 1 1
      ×\ι7
1 2 6 24 120 720 5040
      ÷\8
8
```

If *f*\V represents a scan of a vector, then the scan of any given element in terms of reduction is the following.

```
      R[K] = f/K↑V
```

The shape of the result of a scan is the same as the shape of the original vector (ρ*R*=ρ*V*). If the right argument of the scan is a scalar or a vector with a single element, then *f*\V←→*V*. If *V* is an empty vector, then the result of a scan is the empty vector.

A scan operation may also be specified for one particular coordinate of a multi-dimensional array by including the coordinate number in brackets. Several examples of scan functions are included below. These examples also illustrate the defaults which *APL* supplies when the coordinate number is omitted from the function.

```
      ⎕←A←2 3⍴⍳6
1  2  3
4  5  6
      +\[1]A
1  2  3
5  7  9
      +\[2]A
1   3   6
4   9   15
      +\A
1   3   6
4   9   15
      +⍀A
1  2  3
5  7  9
```

If the dyadic function being performed by a scan is associative (e.g., +, ×), *APL* performs the scan in a way that is different from the conventional scan in order to increase efficiency by reducing the number of operations performed. The definition of *R←f\A* in this case is equivalent to *R[I]=f/I↑A* as follows:

```
R[1]=A[1]
R[I]=R[I-1]fA[I] for I∈1↓⍳⍴A
```

This definition requires fewer operations than the traditional scan. It is possible that the result of an associative function of this kind may differ slightly from the non-associative approach and should be used carefully if the results require a high degree of precision - for example

```
      A←1E6 ¯1E6 1E¯16
      +\A
1000000 0 0
      +/A
0
      +/⌽A
1E¯16
```

2.8.3 *f.g:* Computing the Inner Product of an Array

Function: dyadic inner product (f.g);
Argument Domain: *R←X*f.g*Y*
  left: same as for functions f and g
  right: same as for functions f and g
Argument Shape:
  left: any
  right: any
Result Range: same as for functions f and g
Result Shape: array ρ*R*↔(¯1↓ρ*X*),1↓ρ*Y*
Origin-Dependent? no
Take Dimension Argument? no

The dyadic inner product *(f.g)* operator returns the common algebraic matrix product and also extends this capability to other arithmetic operations and other array dimensions. The following example illustrates the use of the inner product function in calculating a matrix product.

```
        ⎕←A←2 3⍴⍳6
  1  2  3
  4  5  6
        ⎕←B←⍳3
 1 2 3
        A+.×B
 14 32
```

Here the corresponding elements of *B* and each row of *A* are multiplied (*g* function) and then summed (*f* function). Thus (1×1) + (2×2) + (3×3) = 14 and (1×4) + (2×5) + (3×6) = 32.

The inner product operation is expressed as *R←Af.gB*, and functions *f and g* may be any dyadic scalar functions (see Section 2.6).

In *APL*, this matrix product capability is generalized and may be expressed in terms of reduction. If *A* and *B* are both vectors, then the result is a scalar as shown below.

```
        (⍳3)+.×⍳3
 14
```

The expression *R←A*+.×*B* in this case yields the scalar +/*A*×*B*. If *A* is a vector, *B* is a matrix, and *I* and *J* are element indices, then *R* is a vector in which *R*[*J*] is equivalent to +*A*/×*B*[;*J*]. If *A* is a matrix and *B* is a vector, as illustrated in the first example in this section, then *R* is a vector and *R*[*I*] equals +/*A*[*I*;]×*B*. If *A* and *B* are matrices, then *R* is a matrix and *R*[*I*;*J*]↔*f*/*A*[*I*;]*gB*[;*J*]. Following are several examples of inner product functions with different argument dimensions. Note that the last two examples illustrate alternative solutions to the same problem.

```
        A←⎕←2 3⍴⍳6
  1  2  3
  4  5  6
        A+.×⍳3
 14 32
        2 6+.≤A
 0 1 2
```

```
        A+.×⍉A
 14  32
 32  77
        (⍳3)+.×⍳3
14
        +/(1 2 3)*2
14
```

It is often very useful to specify an inner product operation in which an operation other than ordinary multiplication is performed. It is possible to locate values containing specific characters by this method or to search for a row of one array in which all the elements are equal to those in a column of another array. The following example returns a logical vector in which 1 indicates that the text string *SIX* has been located in the corresponding row of array *X*.

```
        X
ONE
TWO
SIX
TEN
        ⍴X
4 3
        Y←'SIX'
        ⍴Y
3
        X∧.=Y
0 0 1 0
```

In general, *A* and *B* may be scalars or any arrays. If either argument is a scalar or a 1-element vector, it is extended so that its length matches the length along the first (last) dimension of the other argument. The result of an inner product function has dimensions such that ⍴*R* is equal to (⍴*A*),⍴*B*, except for the last dimension of *A* and the first dimension of *B*(the two inner dimensions); these can be expressed as ¯1↑⍴*A* and 1↑⍴*B*, and the shape of the result is ⍴*R*←→(¯1↓⍴*A*),1↓⍴*B*. (See the take and drop functions, Sections 2.7.11 and 2.7.12) If ⍴*A*←→*M N* and ⍴*B*←→*L*, then *L*=*N* and ⍴*R*=*M*.

A and *B* must *conform* in order to be used in an inner product operation. *A* and *B* conform if any of the following characteristics is true:

1. *A* or *B* is a scalar.

2. The results of ¯1↑⍴*A* and 1↑⍴*B* are equal.

3. Either ¯1↑⍴*A* or 1↑⍴*B* equals 1.

If the third characteristic is true, then the corresponding argument is extended so that the arguments have equal lengths along the specified coordinate. The basic test for conformability is whether or not the last dimension of the left argument matches the length of the first dimension of the right argument. The dimensions of the result can then be considered all except the last dimension of *A*, catenated to all except the first dimension of *B*. Table 2-7 may be helpful in determining the conformability of two arrays.

Table 2-7
Inner Product Definitions

ρ*A*	ρ*B*	ρ*Af.gB*	Conformability Requirements	Definition *Z←Af.gB*
				Z←f/AgB
	E			*Z←F/AgB*
D				*Z←f/AgB*
D	*E*		*D=E*	*Z←f/AgB*
	E F	*F*		Z[*I*]←*f/AgB*[;*I*]
C D		*C*		Z[*I*]←*f/A*[*I*;]*gB*
D	*E F*	*F*	*D=E*	Z[*I*]←*f/AgB*[;*I*]
C D	*E*	*C*	*D=E*	Z[*I*]←*f/A*[*I*;]*gB*
C D	*E F*	*C F*	*D=E*	Z[*I*;*J*]←*f/A*[*I*;]*gB*[;*J*]

2.8.4 ∘.*f*: Computing the Outer Product of Two Arrays

Function: dyadic outer product (∘.g); *R←X*∘.g*Y*
Argument Domain:
left: same as for function g
right: same as for function g
Argument Shape:
left: any
right: any
Result Range: same as for function g
Result Shape: array; ρ*R*↔(ρ*Y*), ρ*X*
Origin-Dependent? no
Take Dimension Argument? no

The dyadic outer product (∘.*g*) operator specifies an operation to be performed between every element of one array and every element of another array. The form of the function can be expressed as *R←A*∘.*gB*, where *A* and *B* are any arrays and *g* is any dyadic scalar function (see Section 2.6). Note that the ∘ symbol is the jot character (upper-case J on *APL* terminals). *R* is an array that results from applying *g* to every pair of elements of *A* and *B*. The shape of *R* is the dimensions of *A* catenated to the dimensions of *B*, or (ρ*A*),ρ*B*. The following example illustrates the use of this function when *A* and *B* are both vectors.

```
      1 2 3∘.×2 3 4 5
  2   3   4   5
  4   6   8  10
  6   9  12  15
```

Unlike the inner product operator, the outer product performs only one operation - in this case, multiplication. The resulting array is a matrix with three rows (ρ*A*) and four columns (ρ*B*). It is formed by multiplying each element of *A* by each element of *B* in turn - for example, 1×2=2, 1×3=3, 1×4=4, 1×5=5 for the first row, 2×2=4, 2×3=6, 2×4=8, 2×5=10 for the second row, and so on. The example included below illustrates the use of the outer product operator in searching for the occurrence of particular numbers.

```
      A←1 2 3 2 2 3
      (⍳3)∘.=A
 1  0  0  0  0  0
 0  1  0  1  1  0
 0  0  1  0  0  1
      +/(⍳3)∘.=A
1 3 2
```

If *A* is a vector and *B* is a matrix, then the result of *R*←*A*∘.*fB* contains *R*[*I*;*J*;*K*] ←→ *A*[*I*]*fB*[*J*;*K*].

Table 2-8 may be helpful in determining the definition of a variety of outer product results.

Table 2-8
Outer Product Definitions

⍴*A*	⍴*B*	⍴*A*∘.*gB*	Definition *Z*←*A*∘.*gB*
			Z←*AgB*
	E	*E*	*Z*[*I*]←*AgB*[*I*]
D		*D*	*Z*[*I*]←*A*[*I*]*gB*
D	*E*	*D E*	*Z*[*I*;*J*]←*A*[*I*]*gB*[*J*]
	E V	*E V*	*Z*[*I*;*J*]←*AgB*[*I*;*J*]
C C		*C D*	*Z*[*I*;*J*]←[*I*;*J*]*gB*
D	*E V*	*D E V*	*Z*[*I*;*J*;*K*]←[*I*]*gB*[*J*;*K*]
C D	*E*	*C D E*	*Z*[*I*;*J*;*K*]←[*I*;*J*]*gB*[*K*]
C D	*E V*	*C D E V*	*Z*[*I*;*J*;*K*;*L*]←*A*[*I*;*J*]*gB*[*K*;*L*]

CHAPTER 3

DEFINING AND EXECUTING APL PROGRAMS

3.1 MODES OF OPERATION

APL language statements operate in either of two modes:

- *Immediate or execution mode:* in this desk-calculator mode, *APL* statements and expressions entered by the user are executed immediately.

- *Function-definition mode:* in this mode, *APL* programs and functions are developed, edited, named, and saved for use at a future time.

The *APL* user can shift conveniently from one mode to the other by typing a mode-transfer "del" (∇) symbol. The mode in which *APL* statements are to be executed does not affect the syntax of language statements and expressions. However, there are a few special *APL* characters available for use in function-definition mode and a variety of practical considerations to be taken into account when constructing a function to be executed at some future time. This chapter discusses the use of function-definition mode in detail. It focuses on:

- Function definitions, headers, and variables

- Editing procedures for revision and line-editing modes

- Branching and the use of labels, trace vectors, stop vectors, and the state indicator

- Use of locked and suspended functions

3.2 DEFINING THE FUNCTION

APL provides a comprehensive facility for defining, changing, and invoking user functions that supplement the large set of primitive functions that exist in the language. Once the user has developed or rewritten a program in *APL* function-definition mode, that program may be used with the convenience of a primitive function.

A defined program or function is constructed in two parts: a function header and a function body. The *function header* defines the name of the program or function and the syntax of the function call. The *function body* consists of a number of program statements that define the actions to be performed by the function when it is executed. The user enters function-definition mode by specifying a del character (∇), followed by the function header and a carriage return. This signals the *APL* processor not to execute subsequent lines as they are entered, as it would in immediate mode.

In function-definition mode, *APL* prompts the user for successive statements of the function body by displaying successive bracketed line numbers for every line. Lines entered by the user are treated as function lines until *APL* encounters another ∇ character, which signals a return to immediate mode. The format of a function definition is shown in the following:

```
     ∇ function header
[1]  .
[2]  .
[3]  .
[4]  function body
[5]  .
[6]  .
[7]  .
[8]  ∇
```

There are no restrictions on the type of statements that can be included in a function definition. System commands may be included in a definition, and function definition and execution are permitted in quad input mode (Section 2.5.1). In this case, the input request remains pending until the user returns from function-definition mode to immediate mode.

3.2.1 The Function Header

The function header specifies the name of the function and the syntax of its call. There are six distinct types of functions; a function may have zero, one, or two arguments and the function may or may not return a result value. If a defined function has an explicit result value associated with it, this value must be assigned during execution of the function. Defined functions that return results may occur in expressions; those that do not return explicit results must appear alone in statements or be the last function to be executed in an *APL* statement line.

Defined functions may be classified as:

- niladic (no arguments)
- monadic (one argument)
- dyadic (two arguments)

Examples of function headers in these three categories are included below. Note that each type may or may not return a result value (*Z*).

Type	Explicit Result	No Explicit Result
Niladic	∇ *Z←FNAME*	∇ *FNAME*
Monadic	∇ *Z←FNAME ARG*	∇ *FNAME ARG*
Dyadic	∇ *Z←LARG FNAME RARG*	∇ *LARG FNAME RARG*

Sample niladic, monadic, and dyadic functions are included in Section 3.2.5.

3.2.2 Variable Classifications

There are three types of variables that may be used in function definitions:

- dummy variables
- local variables
- global variables

Characteristics of these classes of variables are described in the subsections that follow, along with an explanation of dynamic localization.

3.2.2.1 Dummy Variables - Variables specified in the header component of a defined function (e.g., *Z*, *ARG*, *LARG*, *RARG* in the examples above) are considered *dummy variables*. These dummy variables are included in the header to define the syntax of the function call. In the function body, they hold places for the actual arguments supplied at the time the function is called.

The scope of dummy variables is local to the execution of the function, and the values of all dummy variables except the result (*Z* in the examples above) are provided on calling the function.

3.2.2.2 Local Variables - Variables that have significance only during the execution of a particular function are called *local variables*. If variable *A* is used in functions *F* and *G*, execution of function *F* does not affect the value of *A* within function *G*. Variables may be designated as local by specifying each variable, preceded by a semicolon, in the function header. The following function header:

```
∇RES←A;I;TEMP
```

establishes *I* and *TEMP* as local variables.

During execution of a function, the local value of a variable is always dominant. Local variables are not automatically initialized when a function is called, and any local values are lost upon exit from the function.

Function line labels (Section 3.4.2) are treated as local variables and are also initialized when the function is called; however, labels may not be assigned a value.

3.2.2.3 Global Variables - Variables that have essentially the same significance inside and outside a function definition are considered *global variables*. If a variable is not explicitly defined as a dummy or local variable, it is treated as a global variable. A global variable has the same significance regardless of where it is used, except in certain cases of dynamic localization (Section 3.2.2.4) and suspended execution (Section 3.4.3).

3.2.2.4 Dynamic Localization - The following description provides an example of dynamic localization, which is actually a dynamic form of block structuring. If there exist global variables *A* and *B*, and a function *F* is called with local variables *B* and *C*, then the global value of *B* is not accessible during the execution of *F*. If function *F* calls another function named *G*, with local variable *C*, then within *G* any value assigned to *C* within *F* is not accessible. Upon return to function *F*, local variable *C* resumes its former significance. Finally, upon exit from *F*, variables *A* and *B* resume their global significance and *C* becomes undefined.

The name of a function used in function-definition mode refers to the most global value of the name.

3.2.3 Function Input and Output

The input and output of data values and results of function execution are handled by means of the standard *APL* input/output operators. All of the quad symbols implemented in *APL* can be used in both immediate and function-definition mode. File input and output are discussed in Chapter 6. The other varieties of input and output are described in detail in Section 2.5.

One aspect of *APL* I/O is particularly relevant to a discussion of function execution. An input request may be included within an infinite loop in a function. In this case, the user may escape from input mode by typing the following:

O<backspace>U

in *APL* mode or the mnemonic .OU in ASCII mode. This has the same effect as function suspension (Section 3.4.3); it causes function execution to be interrupted but does not result in an exit from the function.

3.2.4 Comment Lines

Current lines may be included anywhere in an *APL* program; they are particularly appropriate when included in function definitions to annotate the statements included in the definition. Comments may appear on separate lines or be included on the right end of lines containing *APL* statements.

The first character in a comment line must be a lamp (⍝) character, formed by overstriking the down union (∩) and jot (∘) characters. If an ASCII terminal is being used, the first character in a comment line must be a double quote ("). The text that follows the comment character is treated as a comment and may consist of any combination of valid *APL* characters. A comment ends at the end of the line and cannot extend across a line boundary. Examples of comment lines are shown in the function included in Section 3.3.

3.2.5 Examples of Defined Functions

This section contains examples of the three categories of defined functions.

3.2.5.1 Niladic Function - The following niladic function returns no explicit result.

```
        ∇ AVG
  [1]   'ENTER THE VECTOR TO BE AVERAGED:'
  [2]   VECTOR←⎕
  [3]   'THE RESULT IS ';(+/VECTOR)÷⍴,VECTOR
  [4]   ∇
        AVG
```

```
ENTER THE VECTOR TO BE AVERAGED:
⎕:
      3 5 4 6 7
THE RESULT IS 5

      VECTOR
3 5 4 6 7
```

3.2.5.2 Monadic Function - The following monadic function returns an explicit result in *ANS*. Note that the name of the function, *AVERAGE*, can be used in an arithmetic expression just as an *APL* primitive function could be.

```
      ∇ ANS←AVERAGE VEC
[1]   ANS←(+/VEC)÷ρ,VEC
[2]   ∇
      AVERAGE 3 5 4 6 7
5
      100×AVERAGE 3 5 4 6 7
500
```

3.2.5.3 Dyadic Function - The dyadic function included below returns an explicit result. *AVER* is the function name, and *NUM* and *VEC* are global variables used as function arguments.

```
      ∇ ANS←NUM AVER VEC
[1]   'COMPUTATIONAL NUMBER ',⍕NUM
[2]   ANS←(+/VEC)÷ρ,VEC
[3]   ∇
      112 AVER 2 3 2 8 5
COMPUTATIONAL NUMBER 112
4
      113 AVER 5 8 9 9 4
COMPUTATIONAL NUMBER 113
7
      114 AVER 7 7 3 0 1
COMPUTATIONAL NUMBER 114
3.6
```

3.3 EDITING THE FUNCTION

A function definition may be altered by the user in a variety of ways. Definition lines can be added, deleted, and changed, and the function header can be altered. The user must be in function-definition mode in order to perform any of the editing functions described in this section.

The function to be edited is "opened" by typing:

∇*function name*

The user may not attempt to enter or change the entire function header at this time; there is a special method for changing the header, described in Section 3.3.6. After an addition, replacement, insertion, deletion, or display operation, *APL* displays a line number to allow the user to add or enter additional text. If the user does not wish to enter text, he can type a del character (∇) to close the function and thus shift from function-definition to immediate mode. The user may also type the ∇ character on an edit line - for example:

```
        ∇STAT
    [7] [5] MEANX←SUMX÷NSUBJS∇
```

APL replaces line [5] and then exits immediately from function-definition mode.

If the user intends to edit only a single function line, it may be convenient to open the function, specify the line change, and close the function, all in a single statement. The replace operation illustrated above could be specified in the following way:

```
    ∇STAT [5] MEANX←SUMX÷NSUBJS∇
```

The ∇ character can be included on any line except a comment line.

3.3.1 Adding Function Lines

Lines can be added to the end of a function-definition in a very convenient manner. When an existing function is opened, and an editing command is not included on the same line as the del character, *APL* assumes that new lines are to be added and displays the next available line number. For example, the function name *STAT* may exist in the following form before it is edited to remove errors.

```
          ∇ STANDX←NSUBJ STAT X
    [1]    SUMX←X
    [2]    SUMX2←+/(X*2)
    [3]    ⍝COMPUTE MEAN, VARIANCE, STANDARD DEVIATION
    [4]    MEANX←SUMX÷NSUBJS
    [5]    MEANX←SUMX÷NSUBJ
          ∇
```

The user adds two lines in response to the bracketed line numbers displayed by *APL*.

```
           ∇STAT
    [6]   ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
    [7]   STANDX←VARX*0.5
    [8]   ∇
```

The user terminates the specification of additional lines by entering a ∇ character to transfer from function-definition to immediate mode.

3.3.2 Replacing Function Lines

Existing lines in a function-definition can be replaced by specifying the affected line number, followed by the new text of the line. Line number [8], displayed by *APL* below, is simply overridden by specifying line [1].

```
           ∇STAT
    [8]   [1] SUMX←+/X
    [2]   ∇
```

The new specification replaces the erroneous contents of line [1]. *APL* then displays the next line number after the replaced line - in this case, [2]. The user can enter new text for line [2], can specify another line number, or can escape from function-definition mode by typing ∇. This same action could have been performed in a single editing line, as shown below.

```
    ∇STAT[1] SUMX←+/X ∇
```

The line number included in a function body replacement operation must refer to an existing line and must be a positive number less than 1000. It may have a decimal point but may have no more than three decimal places.

3.3.3 Inserting Function Lines

The user can insert new lines between existing lines of the function definition by specifying a new line number, followed by the text of the new line. To insert a line between [5] and [6], for example, the user might specify line number [5.5]. To insert a line before the start of the existing function body, any line number in the range [0] (function header line) to [1] is valid, as shown below.

```
        ∇STAT
[8]   [0.5] ⍝SUM ELEMENTS OF ARRAY X
[0.6]   [5.5] VARX←(SUMX2÷NSUBJ)-MEANX*2
[5.6]   ∇
```

The new specifications are inserted between existing lines [0] and [1] and [5] and [6] respectively. In each case, *APL* displays the next line number after the inserted line. To derive the line that is "next" in an inserted sequence, *APL* adds 1 to the rightmost digit of the user-specified line number. The next line after [0.5] is thus [0.6], the next line after [5.5] is [5.6], and the next line after [8.29] is [8.3]. The user may enter new text for the line number displayed, may escape from function-definition mode by typing ∇, or may override the line number displayed by specifying another line.

After the function definition is closed, the function lines are renumbered by *APL*. As in the case of replacement lines, the numbers of lines to be inserted must be positive numbers less than 1000, with or without a decimal point, and with no more than three decimal places. The renumbered function definition now exists in the form shown below.

```
      ∇ STANDX←NSUBJ STAT X
[1]    ⍝SUM ELEMENTS OF ARRAY X
[2]    SUMX←+/X
[3]    SUMX2←+/(X*2)
[4]    ⍝COMPUTE MEAN, VARIANCE, STANDARD DEVIATION
[5]    MEANX←SUMX÷NSUBJS
[6]    MEANX←SUMX÷NSUBJ
[7]    VARX←(SUMX2÷NSUBJ)-MEANX*2
[8]    ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
[9]    STANDX←VARX*0.5
      ∇
```

3.3.4 Deleting Function Lines

Existing lines in a function definition may be deleted by specifying an erase character (Δ), followed by the line number of the line to be deleted. In the following example, line [5], an incorrect duplicate of line [6], is deleted from the function definition.

```
      ∇STAT
[10]  [Δ5]
[5]   ∇
```

APL displays the number of the line just deleted to give the user an opportunity to specify a new version of the deleted line. The user can enter new text, can specify another line number, or can escape from function-definition mode by typing ∇. After the function is closed, the function-definition lines are renumbered by *APL*.

NOTE

Do not use CONTROL/C to delete a function line.

3.3.5 Displaying Function Lines

The user may display individual lines of the function definition, the function definition from a specified line to the end, or the entire function definition. To display an individual line, the user specifies in brackets the line number of the line to be displayed, followed by a quad character (⎕). In the example included below, line [3] is displayed.

```
        ∇STAT
[9]   [3⎕]
[3]    SUMX2←+/(X⋆2)
[3]   ∇
```

APL displays the number of the line just displayed to give the user an opportunity to specify a new version of the existing line or to override the line number with a new line specification. The user can enter new text, can specify another line number, or can escape from function-definition mode by typing ∇.

To display the function definition from a particular line to the end, the user reverses the sequence described above by specifying the brackets the quad character, followed by the line number from which lines are to be displayed. The following is an example of such a technique.

```
        ∇STAT
[9]   [⎕7]
[7]    ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
[8]    STANDX←VARX⋆0.5
[9]   ∇
```

APL displays the number of the next line after the final line of the function definition - in this case [9] - to give the user the opportunity to add more text or to specify a different line number.

To display the entire function definition, the user simply types a bracketed quad character with no line specification, as shown below.

```
        ∇STAT
[9]   [⎕]
      ∇ STANDX←NSUBJ STAT X
[1]    ⍝SUM ELEMENTS OF ARRAY X
[2]    SUMX←+/X
[3]    SUMX2←+/(X⋆2)
[4]    ⍝COMPUTE MEAN, VARIANCE, STANDARD DEVIATION
[5]    MEANX←SUMX÷NSUBJ
[6]    VARX←(SUMX2÷NSUBJ)-MEANX⋆2
[7]    ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
[8]    STANDX←VARX⋆0.5
      ∇
[9]   ∇
```

The ∇ characters preceding line [1] and following line [8] are displayed by *APL*. They indicate the delimiters of the function and identify its name. They are not true user-specified del characters and therefore do not change the mode. *APL* displays the number of the next line after the final line of the function definition to give the user the opportunity to add new text or to specify a different line number.

NOTE

> Any display of a user-defined function can be terminated by entering a CTRL/O character.

3.3.6 Editing the Function Header

The name or arguments stored in the function header can be edited by accessing line number [0] of the function definition. The header line can be replaced, displayed, or even deleted temporarily. The following example illustrates the display of the function header.

```
          ∇STAT
[9]   [0⎕]
[0]STANDX←NSUBJ STAT X
[0]   ∇
```

The user must include a valid specification for the header before leaving function-definition mode.

3.3.7 Renumbering Function Lines

In function-definition mode, the user is free to include fractional line numbers and line numbers that are not immediately consecutive (e.g., line [15] followed by line [60]). He may also delete existing lines. When the user leaves function-definition mode by entering a del character, *APL* automatically renumbers the lines of the function as consecutive integers, starting at line number [1]. The user should ordinarily display the current version of the function at this time, to avoid referencing the wrong line numbers the next time he edits the function.

3.3.8 Line-Editing Procedures

APL allows the user to edit a function definition in the revision mode described in Sections 3.3.1 through 3.3.7 or in the line-editing mode discussed below. In line-editing mode, the user can alter individual characters in an existing line. To modify an *APL* statement in this mode, the user specifies the line number, followed by a quad character, followed by the estimated character position at which editing is to begin:

```
          ∇DIESEL
[7]   [1⎕19]
[1]     A←R * GAMMA - 1 + (IMAX×9)
                      ↑
```

APL displays the statement and then on the next line indents to the number position specified in the command. The position at which editing is to begin is represented by the up-arrow (↑) character in the example above; it is the 19th position in the line, counting from the first character in the line (e.g., the [character). The user then begins entering edit control characters according to the following rules.

1. Type a slash (/) beneath each character to be deleted.

2. Type a digit or letter beneath each character before which blanks are to be inserted; the particular digit or letter represents the number of blanks to be inserted. For example, a '2' will insert two blanks to the left of the corresponding character in the function line. The alphabetic characters are used to insert multiples of five blanks. For example, '*A*' will insert five

blanks, '*B*' will insert 10 blanks, and so on. If the number of spaces specified plus the current length of the line exceeds the current length of the terminal line, a *DEFN ERROR* is displayed.

3. All other characters typed on the edit control line are ignored by *APL*.

4. The normal rules of correction-before-entry apply. Thus backspacing to insert characters is permitted, and creating illegal overstrikes to facilitate retyping of the line is allowed.

When the carriage returns after the user has finished with the edit control line, the function line is displayed without the deleted characters and with the inserted spaces. The carriage is positioned at the first inserted blank or, if no blanks were inserted, at the end of the line. The user can then enter new text in the blanked area or can make further modifications to the existing text. In this case as well, backspacing to insert new characters and creating illegal overstrikes to facilitate retyping of the line are allowed.

Line editing is a multiple-step process. The first step involves deleting characters no longer needed and inserting sufficient blanks in the line to allow additional desired text to be typed. The second step involves typing in the new text. Repetition of these steps is often necessary. The final appearance of the function line should be identical to a function line just entered from the keyboard.

If the user alters the statement number while editing the line, the function line corresponding to the new number is altered and the original line remains unchanged. This facilitates the movement to or replication of statements in other parts of the program.

Special processing is also performed if the user specifies a character position of zero to the right of the quad character, as shown in the following example:

```
          ∇SECANT
[12]      [2⎕0]
[2]       SECSPEC←ISEC-I↑
```

The function line requested by the user is displayed, and the carriage stops at the next available character position at the end of the line, as shown by the up-arrow (↑) in the example above. The effect is as if the line had been entered by the user from the keyboard. The user can now add text to the line or can backspace to make corrections. The carriage also stops at the end of the line if the number to the right of the quad character is larger than the number of characters in the line.

The following example illustrates the use of line-editing in correcting the line:

```
[1]      T←(LETTR=STRING/⍳8P,STRING
```

There are several errors in this line:

1. *LETTER* is misspelled *LETTR*.

2. The right parenthesis has been omitted after *STRING*.

3. "8" should not appear after the ⍳ character.

4. "*P*" should be a ⍴ character.

Because the first error occurs in *LETTR*, the following command can be supplied:

```
              ∇FUNC
      [5]     [1⎕14]
      [1]     T←(LETTR=STRING/⍳8P,STRING
```

The user now enters the necessary control characters, and *APL* displays the corrected line.

```
      [1]     T←(LETTR=STRING/⍳8P,STRING
                     1        1 //1
      [1]     T←LETT R=STRING /⍳ ,STRING
```

The carriage is positioned at the space between *T* and *R*, and the user simply enters the new characters, spacing over the text to be preserved. He types:

1. "*E*" in the space between *LETT* and *R*.
2. ")" in the space between *STRING* and /.
3. "ρ" in the space between ⍳ and ,.

The new function line is therefore:

```
      [1]     T←(LETTER=STRING)/⍳ρ,STRING
```

This line is entered as a replacement to the existing function line [1] when the user presses the carriage return.

If the user alters the statement number while editing the line, the function line corresponding to the new number is altered and the original line remains unchanged. This facilitates the movement to or duplication of statements in other parts of the program.

3.4 EXECUTING THE FUNCTION

In function-definition mode, the *APL* statements that make up a function definition are neither executed nor checked for syntactic validity when entered. The user simply enters statements, edits them to correct obvious mistypings and inconsistencies, and saves them for future use. The process of defining a function associates the function header provided by the user with the statements entered as the function body. When the user decides to execute the defined function, he uses the function name as he would a primitive *APL* function. The information provided in the function header specifies the number of arguments to be supplied in the function call and determines whether or not a value will be returned. Section 3.2.5 provides examples of defined functions and their corresponding function calls. It is, of course, also possible to issue function calls from within other functions. In the implementation of *APL* described in this manual, function calls may be nested to a depth of about 30 functions.

This section provides information on function execution. It focuses on branching, suspending, tracing, and locking functions, and using the state indicator.

3.4.1 Branching Within a Function

APL statements included in a function definition are normally executed in the order determined by their line numbers. Execution begins at the first statement following the function header, terminates after the last statement in the definition, and is performed only once. It is possible to modify this standard order of execution by including *branching statements* in the function definition. The use of branching also facilitates the specification of execution loops within the body of the function definition.

The simplest form of an *APL* branch statement consists of a branch symbol (→), followed by the number of the function line to which control is transferred. For example:

```
      ∇ FOO
  [5] →1
```

causes an unconditional branch from line [5] to line [1]. Line [1] is thus the next statement executed.

The object of the branch symbol can be a constant, a variable, or an expression; it must evaluate to an integer line number within the current function definition to allow execution to continue. If the integer does not reference a line number in the current function, the branch statement causes a return from the function. Users often deliberately specify an out-of-range-number in order to stop execution. A common specification is:

```
  →0
```

because 0 references the function header and cannot legitimately be accessed by a branch. If the object of the branch is a non-empty vector, control passes to the line referenced by the first element of the vector. If the vector is empty, the branch statement is not meaningful and the normal order of execution within the function definition continues.

Several kinds of *conditional branches* can be specified in function definitions. In *APL*, a conditional branch is executed as the result of evaluating a logical expression, not in response to any specific IF logic. An example of one form of an *APL* conditional statement is shown below. The value of the expression evaluated in the branch statement determines either that control will pass to a specified line number or that the function will return.

```
  →9×I>IMAX
```

The logical expression to the right of the →9 specification is evaluated. If *I* is greater than *IMAX*, the value of the expression will be 9×1 and control will pass to line number [9]. If *I* is not greater than *IMAX*, then the value of the expression will be 9×0; because line [0] is not a legal specification, function execution will return.

In the second version of the conditional branch, the value of an evaluated expression determines whether execution will branch to a specified line number or continue at the next statement. For example, in:

```
  →(VALN≤VALZ)/INIT
```

control will pass to the line labeled *INIT* if the value of the parenthesized expression is true. If it is false, execution will continue at the next line after the branch statement.

3.4.2 The Use of Statement Labels

Because *APL* automatically renumbers function lines as consecutive integers when the user exits from function-definition mode, branch statements should generally not refer explicitly to function line numbers. Instead, the user can associate a *label* with a particular statement in a function definition and then branch to this statement using the label, not the explicit line number, as the object of the branch - for example:

```
[15] INCR: I←I+1
      .
      .
      .
[27]  →INCR×I<IMAX
```

As shown in this example, a statement label consists of an identifier, followed by a colon (:). The internal value of the label is the number of the function line with which it is associated - in this case, line number [15]. Here a branch to the line associated with the *INCR* label is performed, if *I* is less than *IMAX*.

Labels defined within a function must be distinct identifiers. The scope of a label is local to the function in which it occurs, and label values are internally respecified upon each exit from function-definition mode. The user cannot explicitly define a value for a statement label, and a label cannot appear in the function header.

The following are two examples of defined functions that use branching and statement-labeling techniques. Note that function lines containing labels are automatically exdented (i.e., begun one character position to the left of the rest of the *APL* text) when the function-definition is displayed.

```
     ∇R←FACTORIAL N
[1]   R←1
[2]   →0×⍳0=N                   (Branch to line [0] (halt) if 0 is
[3]   R←R×N                     equal to N)
[4]   N←N-1
[5]   →2                        (Unconditional branch to line [2])
[6]   ∇
     ∇Z←FAC N
[1]   →NZERO×⍳N=0               (Branch to the line labeled NZERO
[2]   Z←N×FAC N-1               if  N is equal to 0)
[3]   ⍝NOTICE THAT RECURSIVE DEFINITIONS
[4]   ⍝ARE PERMITTED.
[5]   →0                        (Unconditional branch to line [0]
[6]  NZERO: Z←1                 (halt))
[7]   ∇
```

```
      FAC     5
120
```

3.4.3 Suspending Function Execution

Function execution is suspended before normal completion if an error occurs, if the user types a CTRL/C character, or if a stop vector (see Section 3.4.6) is set. When execution is suspended, the name of the suspended function and the line number of the statement that would have been executed next are displayed. *APL* then begins a new line, indents six spaces, and awaits input in immediate mode. The user can perform virtually any *APL* operation at this time, except for editing or erasing the suspended function.

The suspended function remains active until terminated or until the current state indicator or active workspace is cleared. The user can resume execution at any time by typing:

```
    →line
```

where *line* identifies the statement number at which execution is to be continued. A suspended function can be terminated by typing:

```
    →0
```

The local variables associated with the suspended function remain active. The user can examine these variables and can specify their values by means of an immediate-mode assignment.

3.4.4 Examining the State Indicator

The state indicator, a status vector that resides in the user's active workspace, can be examined to determine the status of all active functions in the *APL* system. The user can specify an *)SI* system command (Section 5.3.9) to obtain a listing of the active functions, as in the following:

```
          )SI
T[1] *
S[7]
R[6]
F[3] *
```

The listing displays functions in the order in which they were most recently active. The example included above indicates that execution was suspended just before executing statement [1] of function *T*, which was called during line [7] of function *S*, which was called during line [6] of function *R*. Before this sequence of calls, execution was suspended just before executing line [3] of function *F*.

In the *)SI* display, an asterisk (*) following the name and line number indicates a *suspended function*, and a blank indicates a *pendent function*. A pendent function is usually one which is awaiting return from another function - possibly a suspended one - which it called.

The user can also determine from the *)SI* listing when quad input requests are pending or an execute operation (⍎) has been invoked. Examples of both of these special conditions are shown below.

```
          )SI
T[1] *
S[7]
R[6]
F[3] *
```

```
        ε'⎕'
⎕:
        )SI
⎕
ε
T[1] *
S[7]
R[6]
F[3] *
```

The user can clear the state indicator by terminating the execution of each suspended function in the list. There are several ways to accomplish this. The user may type one right arrow (→) for each function marked by an asterisk (each right arrow on a separate line); he may issue an ɪ30 ɪ-beam function to clear the state indicator completely (Section 4.3.14); or he may clear the state indicator by saving the active workspace, then clearing and loading it again (see the *)SAVE* and *)LOAD* system commands, Sections 5.2.3 and 5.2.4). A cleared state indicator is displayed in the form of a blank line.

The *)SIV* system command (Section 5.3.10) can be used to obtain a more extensive display of the state indicator. In addition to the information accessible to *)SI*, *)SIV* returns a list of local variables for each function displayed. The following is an example of an *)SIV* display.

```
        )SIV
TRIG[1] * A Q R
T[1] * N
S[7] N
R[6]
F[3] *
```

This indicates that the variable *N* local to function *T* is currently dominant, and that the variable *N* local to function *S* is currently inaccessible.

3.4.5 The Trace Vector

The user may find it helpful for debugging purposes to obtain an automatic display of the intermediate results of function execution. As a program tracing aid, the values computed by one or more function statements can be output each time those statements are executed. To establish a trace for function *F*, the user specifies a vector in the following format:

```
    T∆F←4 6 7
```

For each execution of the line numbers [4], [6], and [7], this command causes the following information to be displayed, in the order shown:

- function name
- bracketed statement line number
- final value returned by the statement

If the statement being traced is a branch statement, then the value printed is the value to which control is passed by the branch.

To trace all the statements of a function *F*, the following specification can be supplied if the index origin is currently set to 1:

```
    T∆F←⍳N
```

where *N* is a number at least as large as the number of statements in *F* of the index origin is 0, the user issues the statements.

because the function neader (line 0) cannot be traced. To disable the trace vector for function *F*, the user includes either of the following statements:

```
    T∆F←0
    T∆F←⍳0
```

A new trace vector does not override an existing specification. If lines [4], [6], and [7] are currently being traced, the user may add line [5] to this list simply by entering trace vector:

```
    T∆F←5
```

However, to omit line [6] from an existing trace vector, the user must disable the trace vector for the function and then enter a new trace vector, as shown in the following:

```
    T∆F←⍳0
    T∆F←4 5 7
```

NOTE

Editing a line for which a trace vector has been defined causes the trace to be disabled for that line.

The following is an example of a function definition followed by two executions of that function, the first with the trace vector enabled.

```
          ∇ ANSWR←FACTORIAL N ;COUNT
    [1]   ⍝CALCLATES FACTORIAL OF N
    [2]   ANSWR←1
    [3]   →(0≥N)/0
    [4]   ANSWR←N×FACTORIAL N-1
          ∇


          T∆FACTORIAL←2 3 4
          FACTORIAL 4
    FACTORIAL[2] 1
    FACTORIAL[3]
    FACTORIAL[2] 1
    FACTORIAL[3]
    FACTORIAL[2] 1
    FACTORIAL[3]
    FACTORIAL[2] 1
    FACTORIAL[3]
    FACTORIAL[2] 1
    FACTORIAL[3] 0
    FACTORIAL[4] 1
    FACTORIAL[4] 2
    FACTORIAL[4] 6
    FACTORIAL[4] 24
    24
          T∆FACTORIAL←⍳0
          FACTORIAL 4
    24
```

3.4.6 The Stop Vector

APL allows the user to suspend execution of a function from within the function itself by specifying a stop control vector. The syntax of this vector is similar to that of the trace vector. The stop vector can be used to suspend function execution just before execution of one or more specified statements. To cause function *F* to be suspended before executing line [12] and line [19], the user includes the following statement in the function definition:

```
    S∆F←12 19
```

For each suspension, this command displays the function name and line number that was about to be executed. To disable the stop vector for function *F*, either of the following specifications may be supplied:

```
    S∆F←0
    S∆F←⍳0
```

After function execution has been suspended by means of the stop control vector, the system is in the normal suspended state. An entry is included in the state indicator, identifying the suspended function and the line at which it was suspended. Execution can be resumed by specifying a branch to the desired line number.

Execution of a function cannot be suspended before line 0 (the function header). The stop control vector can be set from within a function to cause suspension only under certain circumstances.

NOTE

> Editing a line for which a stop vector has been defined causes the stop vector to be disabled for that line.

An example of the use of the stop vector is included below.

```
          S∆FACTORIAL←3
          FACTORIAL 4

    FACTORIAL[3]
          )SI
    FACTORIAL[3] *
          →3

    FACTORIAL[3]
          →3

    FACTORIAL[3]
          →3

    FACTORIAL[3]
          →3

    FACTORIAL[3]
          →3
    24
          S∆FACTORIAL←⍳0
          FACTORIAL 4

    24
```

3.4.7 Locking a Function

It may be desirable to prohibit users from changing and possibly damaging existing function definitions. *APL* allows a user to lock a function definition in order to protect it from unauthorized use, to maintain security, or to treat a function as a proprietary program. To create a locked function or to lock an existing function, the user closes the function-definition with a del-tilde (⍫) character rather than a simple del (∇). The ⍫ is created by overstriking ∇ and ~. The following example illustrates the locking of a previously unlocked function-definition.

```
              ∇ TRIG
        [19]  ⍫
```

A locked function cannot be edited in the manner described in Section 3.2. Function lines cannot be added, changed, deleted, or displayed for locked functions. Trace and stop control vectors cannot be defined or changed for the function. Any trace or stop settings in effect at the time a function-definition is locked are automatically nullified.

If an error occurs during execution of a locked function, the function name and the line number at which the error occurred are displayed, but the contents of the statement are not included in this display. *APL* then causes an exit to immediate-mode.

CHAPTER 4

APL SYSTEM VARIABLES AND I-BEAM FUNCTIONS

4.1 INTRODUCTION

There are a variety of ways in which the user may communicate with the *APL* system in order to change system parameters, determine hardware or operational characteristics, and modify processing methods. The system commands documented in Chapter 5 facilitate many of these system operations. The system elements described in this chapter allow *APL* users to communicate with the system from within the *APL* language itself. These elements are subject to the *APL* language syntax and rules of function definition. They may be included in *APL* functions and defined in conjunction with other language operations.

The system elements described in this chapter can be grouped in two categories: system variables and ⌶-beams. In some cases, system variables and ⌶-beams perform related functions. In other cases, these system features provide alternative ways of performing operations invoked by the *APL* system commands.

4.2 SYSTEM VARIABLES

System variables have been implemented in this version of *APL* to facilitate communication with the *APL* system. They are used to perform such operations as the following:

- set the index origin and relative fuzz
- change the output precision and line width
- reference the characters in the collating sequence
- report on executing functions and available workspace area

System variables are syntactically similar to ordinary variables and may be used in any language expression or function. System variables differ from ordinary variables because of their special significance to the system. System variable names are *distinguished names;* they begin with a quad (⎕) character and cannot be used for user-defined purposes. They cannot be copied, erased, or collected in a group by means of the *APL* system commands (see Chapter 5).

The system variables described in this section are considered *shared variables* because they are shared by the user's workspace and the *APL* processor and serve as an interface between the two. The sharing facility is invoked automatically when the workspace is activated. Sharing implies that the workspace and processor may each use values specified by the other, as appropriate to the particular operation being performed. It also implies that the value of a variable being used in a workspace may sometimes be different from the value last specified by the user of the workspace. The variables described in this section fall into two categories:

- System variables that assume the value provided by the user and retain it until the user overrides the value or clears the workspace. These variables are described in Sections 4.2.1 through 4.2.5 and have default values in effect when the workspace is loaded. An example of such a variable is *⎕PP*, which is used to determine the precision of numeric output. If the value specified by the user is invalid for the operation, *APL* will return a *DOMAIN ERROR* when the assignment is attempted.

- System variables that retain the values supplied by the *APL* system. Because of their syntactic similarity to ordinary variables, these system variables can be set by the user; however, they will continue to have the values supplied by the system. These variables are described in Sections 4.2.6 through 4.2.8.

4.2.1 *⎕CT*: Establishing the Comparison Tolerance

Default: 5*E*¯15 (double-precision)
5*E*¯7 (single-precision)

Example:
```
      ⎕CT
 1E¯13
      ⎕CT←1E¯15
```

The *⎕CT* system variable is used to set the degree of tolerance or relative fuzz to be applied in performing comparisons. It is used in conjunction with the relational operators: (<, ≤, =, ≥, >, ≠) and with the dyadic-index (⍳) and membership (∊) functions and floor (⌊) and ceiling (⌈).

The *⎕CT* value specified by the user is saved when the active workspace is saved. See the description of fuzz in Section 2.4.3. The value for *⎕CT* must be in the range *0*≤*⎕CT*≤ approximately .38.

4.2.2 *⎕IO*: Setting the Index Origin

Default: 1

Example:
```
 ⎕IO
 1
       ⎕IO←0
       ⍳3
 0 1 2
```

The *⎕IO* system variable is used to change the setting of the index origin. This setting is important in array operations and in conjunction with roll and deal (Sections 2.7.19 and 2.7.20) and iota (Sections 2.7.5 and 2.7.6). The value of *⎕IO* is saved when the active workspace is saved and is only meaningful if it is 0 or 1. This variable is equivalent to the *)ORIGIN* system command (Section 5.4.1).

4.2.3 *⎕PP*: Determining the Output Precision

Default: 10 (double-precision)
6 (single-precision)

Example:
```
      ⎕PP
 10
      ⎕PP←15
```

The ⎕*PP* system variable is used to determine the precision of non-integer output by setting the number of significant digits to be displayed. It is also relevant to the expression of characters by means of the monadic format (⍕) function (Section 2.7.32). Legal values for ⎕*PP* are integers in the range 1 through 7 for single-precision systems and 1 through 17 for double-precision systems. This system variable does not affect the precision of internal calculations or the display of numerical constants. The precision specified by the user is saved when the active workspace is saved. ⎕*PP* is equivalent to the *)DIGITS* system command (Section 5.4.2).

4.2.4 ⎕*PW*: Determining the Width of the Output Line

Default: 120

Example:
```
⎕PW
120
⎕PW←130
```

The ⎕*PW* system variable is used to set the maximum number of characters that may appear in an output line. Legal values for ⎕*PW* are integers in the range 30 through 384. It does not affect the display of messages on the terminal or the allowable length of input lines. The width specified by the user is saved when the active workspace is saved. ⎕*PW* is equivalent to the *)WIDTH* system command (Section 5.4.3).

4.2.5 ⎕*RL*: Setting a Random Link

Default: 0

Example:
```
⎕RL←¯1+2*15
```

The ⎕*RL* system variable is used to set the sequence used by the pseudo random number generator in *APL*. This random number generator is used in the *APL* roll and deal functions (Sections 2.7.19 and 2.7.20). The value of ⎕*RL* is the starting point of the chain used to generate the numbers. This system variable has a meaningful range of 0 through ¯1+2*15. The value of ⎕*RL* specified by the user is saved when the active workspace is saved.

4.2.6 ⎕*AV*: Storing a Vector of Characters

Example:
```
LINEFD←⎕AV[99]
```

The ⎕*AV* (atomic vector) system variable is a vector containing all possible characters; ⎕*AV* is 256 elements in length and is used to express the binary representation of any character in the *APL* system. For example, if the index origin setting is 0, the following expression refers to the carriage return, backspace, and line feed characters:

```
⎕AV[10 8 13]
```

The indices associated with any of the *APL* characters can be retrieved; if the index origin is 0, the following expression returns the elements shown below.

```
      ⎕AV⍳'ABCABC'
       97  98  99  150  151  152
```

Many of the elements of the atomic vector are non-printing characters, and some do not even exercise control.

See the discussion of relational functions in Section 2.6.4.

4.2.7 ⎕*LC*: Reporting on Executing Functions

Example: →⎕*LC*

The ⎕*LC* (line counter) system variable is used to obtain a partial report on functions that are currently being executed. It is stored as a vector of the line numbers contained in the state indicator, arranged in order of most recently suspended function first. ⎕*LC* is particularly useful in branch statements; the user can simply specify that execution is to resume immediately following the line number at which function execution was most recently suspended, as shown in the example above. ⎕*LC* is related to the following ⌶-beams:

⌶-beam	Meaning
⌶27	Vector of line numbers of functions in the state indicator
⌶26	Current value of the first line number in the state indicator

4.2.8 ⎕*WA*: Reporting the Available Working Area

```
Example:    ⎕WA
      20000
```

The ⎕*WA* system variable is used to determine the maximum amount that the active workspace may increase. The size is given in bytes and is obtained by subtracting the current low-segment size from the maximum low-segment size. ⎕*WA* is equivalent to ⌶-beam 22, which also returns the available working area.

4.3 I-Beams

There are two types of ⌶-beam functions. The first type consists of functions used to return information about the user's workspace and the *APL* system. The following are examples of information returned by the ⌶-beams in this category:

- Symbol table size
- Date and time of day
- Terminal character set
- Line numbers of functions in the state indicator
- Precision of *APL* version

Some of these ⌶-beams report general system characteristics (e.g., date) and others return information relevant only to the particular user's workspace and session (e.g., line numbers of suspended functions).

The second type of ⌶-beams consists of functions used to perform system actions and to change workspace parameters. The following are examples of actions performed by the ⌶-beams in this category:

- Turning on and off error displays for the execute operator
- Clearing the state indicator
- Terminating the *APL* session
- Changing the random number sequence

⌶-beam functions are initiated by means of the following format:

```
        ⌶A
```

where the ⌶ character is formed by overstriking the ⊤ and ⊥ characters. The *A* argument is a number identifying the particular function to be invoked. *A* may be a constant or a variable. It must be a scalar or a one-element array.

4.3.1 ⌶15: Reinitiating Error Displays for the Execute Function

Example:

```
        ⌶16

        ⍎'B←⎕×9'

        ⌶15

        ⍎'B←⎕×9'

VALUE ERROR
B←⎕×9
  ↑
```

⌶-beam 15 turns on the display of error messages for the execute (⍎) function after these messages have been suppressed by ⌶-beam 16 (Section 4.3.2). If an error is encountered while *APL* is processing an execute string, the system does not display an error message or echo the line in which the error occurred if ⌶-beam 16 has been issued. To reinitiate error displays for the execute function, the user may specify an ⌶15 function. The execute function is described in detail in Sections 2.7.24 and 5.6.

4.3.2 I16: Suppressing Error Displays for the Execute Function

Example:

```
      ⍎')COPY FOO'
?Can't find file or account
      ⌶16
      A←⍎')COPY FOO'
      A
      ⍴A
0 55
```

I-beam 16 turns off the display of error messages for the execute (⍎) function. If I-beam 16 has been issued and an error is encountered while *APL* is processing the execute string, execution is interrupted but the system does not display an error message and echo the line in which the error occurred. This allows the user to retain control, to handle the error condition under program supervision, and to continue executing the function if desired. After an error has been detected, the value returned by the execute string is a null array whose shape is 0 *E*, where *E* is a number indicating the error that was encountered.

In the example at the beginning of this section, error number 55 occurred, because the specified file could not be located. Appendix D contains a complete description of all *APL* error conditions.

To turn on the display of execute error messages after they have been suppressed, the user may issue I-beam 15 (Section 4.3.1).

4.3.3 I18: Returning the Condition of the Workspace

Example:

```
      ⌶18
0
```

I-beam 18 returns the condition of the active workspace. A value of 0 indicates that the workspace is intact, and a value of 1 indicates that the workspace has suffered some kind of damage. If I-beam 18 returns a value of 1, the user should correct the damage by clearing the active workspace with a *)CLEAR* system command (Section 5.2.1) or replacing it with a *)LOAD* (Section 5.2.3) command.

4.3.4 I20: Returning the Time of Day

Example:

```
      ⌶20
2887053
      24 60 60 60⊤⌶20
13 22 11 40
```

I-beam 20 returns the current time of day as time since midnight in 60ths of a second (50ths of a second in Europe). The user may apply an *APL* encode (⊤) function to the returned value to format the time in hours, minutes, seconds, and 60ths of seconds. This is illustrated in the second example above.

4.3.5 I21: Returning the CPU Time (RSTS/E Only)

Example:

```
      I21
5844
      24 60 60 60⊤I21
0 1 37 24
```

I-beam 21 returns the CPU time expended since the user signed on in the current *APL* session. Time is expressed in 60ths of a second (50ths of a second in Europe). As illustrated in the second example above, the user may apply an encode (⊤) function to the returned value to format the CPU time in hours, minutes, seconds, and 60ths of seconds.

I-beam 21 is useful in comparing the execution times of different programs. It may also be included in a function, and the execution of that function made dependent on the compute time used so far in the session.

I-beam 21 is a RSTS/E function; under RT-11, RSX-11M and IAS, it returns a "NOT YET IMPLEMENTED" error.

4.3.6 I22: Returning Workspace Availability

Example:

```
      I22
16394
```

I-beam 22 is used to measure the maximum amount that the active workspace may increase. The size is given in bytes. I-beam 22 may be used in a function whose execution is dependent on the amount of free space available in the workspace.

4.3.7 I23: Returning the System Job Number (RSTS/E Only)

Example:

```
      I23
11
```

I-beam 23 returns the system job number associated with the user's current *APL* session in base 10 notation. This I-beam is a RSTS/E function; under RT-11, RSX-11M, and IAS, it returns a value of zero.

4.3.8 ⌶25: Returning Today's Date

Example:

```
      ⌶25
30579
      A←(3⍴100)⊤⌶25
      A
3 5 79
```

⌶-beam 25 returns today's date in base 10 notation in the form MMDDYY. As illustrated in the second example above, the user may apply encode (⊤) and rho (⍴) functions to this returned value to format the date as a three-element vector.

4.3.9 ⌶26: Returning a Line Number

Example:

```
      FUNC1

FUNC2[2]
      )SI
FUNC2[2] *
FUNC1[1]
      ⌶26
2
```

⌶-beam 26 returns the line number of the statement currently being executed or about to be executed. The scalar returned by ⌶-beam 26 is the first line number in the state indicator (Section 3.4.4) and the first element of the vector returned by ⌶-beam 27 (Section 4.3.11). This number represents the line at which the innermost function was suspended. If *APL* displays a blank line, this indicates that the state indicator is empty and no functions are currently suspended.

⌶-beam 26 is particularly useful in branch statements. The user can simply resume execution of the innermost function by specifying →⌶26, as shown in the example, rather than entering the line number displayed at the time the last function was suspended. To branch two lines from the current line in the suspended function, the user specifies →2+⌶26.

4.3.10 ⌶27: Returning a Vector of Line Numbers

Example:

```
      )SI
FUNC2[2] *
FUNC1[1]
      ⌶27
2 1
```

⌶-beam 27 returns a vector of function line numbers currently in the state indicator (Section 3.4.4). The first element of the array is the line number that would be returned by ⌶-beam 26 and represents the line at which the innermost function was suspended. If *APL* displays a blank line, this indicates that the state indicator is empty and no functions are currently suspended.

I-beam 27 is an aid in resuming function execution without including a specific line number at which the function was suspended. The user may define function *RES*, as shown in the example below, and then resume execution of the second function in the state indicator by entering →*RES*.

```
       ∇ A←RES
[1]   A←(⌶27)[2]
[2]   ∇
       →RES

^C

EXECUTION STOP
FUNC2[1]  A←A
           ↑
```

4.3.11 ⌶28: Returning the Terminal Character Set

Example:

```
       ⌶28
1
```

I-beam 28 returns a value indicating the character set specified for the user's terminal. The value returned by this I-beam is one of the following:

Value	Meaning
0	*APL* character set
1	ASCII character set

The character set is selected at the time the user begins the *APL* session (Section 1.5).

4.3.12 ⌶29: Returning the User's Project-Programmer Number (RSTS/E Only)

Example:

```
       ⌶29
129 149
```

I-beam 29 returns the project-programmer number of the *APL* user as a two-element vector in base 10 notation. This I-beam is a RSTS/E function; under RT-11, RSX-11M, and IAS, it returns two zeroes.

4.3.13 ⌶30: Clearing the State Indicator

Example:

```
⌶30
)SI
```

⌶-beam 30 clears the state indicator. It is equivalent to typing a series of right arrows (→), one for each suspended function. ⌶-beam 30 removes all pendent and suspended function calls from the system. As shown in the example, an *)SI* system command (Section 5.3.9) issued after the ⌶-beam results in the display of a blank line, or null vector.

If several errors have occurred during function execution, ⌶-beam 30 should be specified before a *)SAVE* system command (Section 5.2.3) is issued for that function. See Section 3.4.4 for a discussion of alternative ways of clearing the state indicator.

4.3.14 ⌶36: Terminating the APL Session

Example:

```
      ⌶36

Ready
```

⌶-beam 36 exits from the *APL* system and returns control to command level. This ⌶-beam performs the same function as the *)OFF* system command (Section 5.5.1). Under RT-11, the *APL* user returns automatically to system command level after issuing ⌶-beam 36. RSTS users return automatically to the BASIC environment, as illustrated in the example above. RSX-11M users return to the Monitor Console Routine (MCR). IAS users return to the Program Development System (PDS).

4.4 SYSTEM FUNCTIONS

The version of *APL* described in this manual supports a variety of system functions, implemented as part of the *APL* shared variable facility. The six system functions described in this section allow the user to perform such operations as the following:

- Express the canonical representation of a function and store function definitions as data
- Erase a named object
- Construct a name list of labels, variables, or functions and return the classification of a named object

System functions are an integral part of the *APL* language and may be used freely in all *APL* function definitions. They can be clearly distinguished from the primitive functions available in the *APL* language; like system variables, the names of the system functions described in this chapter begin with a quad (⎕) character and are reserved for the use described below. And like system variables, these functions cannot be copied, erased, or collected in a group.

4.4.1 ⎕*CR*: Obtaining a Canonical Representation

Format: ⎕*CR A*

Rank: 1≥⍴⍴*A*

Example: ⎕*CR 'TRIG'*

The ⎕*CR* system function is used to obtain a *canonical representation* of a defined function. ⎕*CR* operates on a character array that identifies the name of the function; this array is represented by *A* in the format above. A canonical representation of a defined function is a character matrix with rows consisting of the original lines of the function definition, reformatted to be of equal length. The ∇ symbols, line numbers, and brackets are removed from the definition. Lines that contain labels are shifted to the right so all text begins at the same character position. Lines are then right-padded with blanks to make all lines equal in length to the longest line of the function. This reformatting allows the function definition to be treated as data. The example shown below illustrates the original function to be referenced by ⎕*CR* and the matrix or canonical representation that results from the operation of the system function.

```
          ∇MEAN[⎕]∇
        ∇ MEANX←NSUBJ MEAN X
[1]       ASUM VECTOR X
[2]       SUMX←+/X
[3]       MEANX←SUMX÷NSUMJ
        ∇
          A←⎕CR 'MEAN'
          A
MEANX←NSUBJ MEAN X
ASUM VECTOR X
SUMX←+/X
MEANX←SUMX÷NSUBJ
          ⍴A
4  18
          X←8 6 3 9 5 4 2 1 7 4
          10 MEAN X
4.9
```

If the *A* argument in the ⎕*CR* function does not represent the name of a defined and unlocked function, the resulting matrix is of dimension 0 by 0. *APL* returns a *RANK ERROR* if *A* is not a vector or scalar and a *DOMAIN ERROR* if the argument is not a character array.

4.4.2 ⎕*FX*: Establishing a Function

Format: ⎕*FX M*

Rank: 2=⍴⍴*M*

Example: ⎕*FX A*
TRIG

The ⎕*FX* (fix) system function effectively reverses the operation performed by ⎕*CR*. This function operates on a character matrix that contains a canonical representation of a function; this array is represented by *M* in the format above. It establishes in the user's workspace a function that has the name of the function associated with the canonical representation *M*. If a function with the same name already exists in the active workspace, ⎕*FX* will replace it. The matrix identified by *M* is not affected by the ⎕*FX* operation. The following example can be considered a continuation of the example begun in Section 4.4.1.

```
      A[3;6]←'×'
      ⎕FX A
MEAN
      ∇MEAN[⎕]∇
    ∇ MEANX←NSUBJ MEAN X
[1]   ASUM VECTOR X
[2]   SUMX←×/X
[3]   MEANX←SUMX÷NSUBJ
    ∇
      X
8  6  3  9  5  4  2  1  7  4
      10 MEAN X
145152
```

Another example of the use of ⎕*FX* in conjunction with the execute operator is shown below.

```
      ⍎('10 ',⎕FX A),' X'
145152
```

The normal rules about local names apply to the names of any functions established by the ⎕*FX* function. If the *BG* function is fixed within function *Z* and the name *BG* is a local one, the *BG* definition is not preserved after execution of the *Z* function comes to an end. Standard function-definition mode applies only to global names.

⎕*FX* will not establish a function if the name of the function to be established is the same as that of an existing label, variable, or group or an existing function that is currently pendent or suspended. A pendent function is usually one that is awaiting return from another function. ⎕*FX* will execute properly if the matrix referenced by ⎕*FX* is identical to a canonical representation except for the addition of blank characters in rows other than those consisting only of blanks. If ⎕*FX* cannot establish a function, a scalar index representing the row in *M* where an error was found is returned. No change is made to any function or matrix in the user's workspace. *APL* returns a *RANK ERROR* if *M* is not a matrix and a *DOMAIN ERROR* if the argument is not a character array.

4.4.3 ⎕*EX*: Erasing a Named Object

Format:	⎕*EX A*
Rank:	2≥⍴⍴*A*
Example:	⎕*EX 'ABMAX'*

```
              1
```

The ⎕*EX* (expunge) system function is used to erase an existing use of a name. ⎕*EX* operates dynamically on a character array that identifies the name to be erased; this array is represented by *A* in the format above. This function has capabilities similar to those of the)*ERASE* system command, except that it cannot erase a named object that refers to a label, a group, a suspended or pendent function, or a system variable. In addition, ⎕*EX* operates only on global or dominant local variables. It is used particularly to avoid conflicts that may occur because of duplicate occurrences of the same name in the *APL* workspace. ⎕*EX* applies to a matrix of names and produces as a result a logical vector. It returns a value of 1 if an existing version of a name is successfully erased and the name is now free to be used, as shown in the example above. If the name cannot be erased for any of the reasons described, a result of 0 is returned. A 0 result is also returned if the *A* argument does not represent a legal *APL* variable name. *APL* returns a *RANK ERROR* if *A* has a rank higher than that of a matrix and a *DOMAIN ERROR* if the *A* argument is not a character array.

4.4.4 ⎕*NL*: Constructing a List of Labels, Variables, or Functions

Monadic Form:

Format: ⎕*NL N*

Rank: 1≥ρρ*N*

Example: *LIST*←⎕*NL* 2

Dyadic Form:

Format: *A* ⎕*NL N*

Rank: 1≥ρρ*N*
1≥ρρ*A*

Example: '*GKM*' ⎕*NL* 1 3

The ⎕*NL* system function is implemented in both monadic and dyadic form. Both forms of the function are used to construct a list of named objects residing in the active workspace. The *N* parameter is included in both forms of the function to identify the type of named objects to be included in the name list. The parameter is an integer scalar or vector that can have one of the following values:

Values	Meaning
1	Labels
2	Variables
3	Functions

For example:

```
X←⎕NL 1 2
```

causes the names of all labels and variables in the workspace to be included in name list *X* in alphabetical order. Each row of the matrix will contain the name of one label or variable.

The dyadic form of the ⎕*NL* function allows the user to restrict the name list to names beginning with specified characters by including an *A* parameter in the command. For example:

```
NLIST←'ABCDEF' ⎕NL 3
```

causes a name list to be constructed of function names whose initial letters are *A* through *F*; the list is arranged in alphabetical order. The *A* parameter must be a scalar or vector of alphabetic characters. The letters supplied in the character string must be included in alphabetic order.

The ⎕*NL* system function can be used for a variety of purposes. Some of these are described below.

- ⎕*NL* can interact with ⎕*CR* in creating functions that can automatically display the definitions of all or a subset of functions in the workspace. It can also be used to analyze interactions between variables and functions.
- In its dyadic form, ⎕*NL* can guide the user in choosing names while developing or interacting with a workspace.
- In conjunction with ⎕*EX*, the ⎕*NL* function can cause all of the named objects in a certain category to be erased dynamically. It also facilitates the design of a function that can be used to clear a workspace of all but a preselected collection of named objects.

The following example illustrates the construction of a matrix containing the names of variables in the active workspace that begin with the letter *V*.

```
      NLIST←'V' ⎕NL 2
      NLIST
VAR1
VAR2
VAR203
VAR204
VAR99
VBMAX
```

4.4.5 ⎕*NC*: Returning a Name Classification

Format: ⎕*NC A*

Rank: 2≥ρρ*A*

Example:
```
      ⎕NC 'VAR99'
2
```

The ⎕*NC* system function is used to return the classification of a name or series of names. ⎕*NC* operates on the matrix, vector, or scalar represented by argument *A*. If *A* is a character matrix, ⎕*NC* returns the class of the name represented by each row of *A*. If *A* is a vector or scalar, ⎕*NC* returns the classification of a single name. The ⎕*NC* function returns a numerical value representing each name classification as follows:

Value	Meaning
0	Name available for any use
1	Label name
2	Variable name
3	Function name
4	Not available for use as a name

A value of 4 implies that argument *A* is not a valid name or that it is currently in use as a group name.

CHAPTER 5

SYSTEM COMMANDS

5.1 OVERVIEW OF SYSTEM COMMANDS

A wide variety of system commands have been implemented to provide a means of communicating with the *APL* system and controlling the operational environment in which an *APL* session is conducted. System commands allow the user to examine or change the state of the system in such ways as the following:

- Clear, name, and save the active workspace.
- Load and delete a workspace from a secondary storage device.
- List variable and function names.
- Display the status of functions and local variables in the workspace.
- Set and display the index origin, maximum number of significant digits, output line width, and comparison tolerance.

System commands are not considered a part of the *APL* language itself, but can be viewed as an interface between the user and the language processor. System commands implemented for use with the *APL* file system are described in Chapter 6. Appendix B provides a summary of the format of all system commands, in alphabetical order.

This chapter is structured in the following way. Section 5.1 provides an overview of the format, function, and interaction of system commands. Sections 5.2 through 5.5 describe the system commands implemented for use with *APL* in the following categories:

Section	Commands
5.2	Basic workspace-control commands
5.3	Workspace-content commands
5.4	Workspace-environment commands
5.5	*APL* termination commands

Section 5.6 discusses the special function of the execute operator (⍎) in relation to system commands.

5.1.1 System Command Format

System commands begin with a right parenthesis, as shown in the following format:

)command-name [parameter-list]

Some system commands require the inclusion of one or more parameters or arguments in the command line. If required or optional parameters are included, at least one space must separate the individual elements of the system command.

The examples included below illustrate the format of several system commands.

`)CLEAR`	(No parameters required)
`)DIGITS 6`	(Parameter required)
`)VARS Q`	(Parameter optional)
`)ERASE A B C D`	(One or more parameters required)

5.1.2 Action and Inquiry Commands

APL system commands may be used in two distinct modes: action and inquiry. *Action commands* invoke some change in the state of the *APL* system. *Inquiry commands* report on the state of the system but do not change this state in any way. The *)ORIGIN* command is an example of an action command. It indicates the index origin to be used during the current *APL* session and is specified in the following way:

```
      )ORIGIN 0
WAS 1
```

The *)SI* command, on the other hand, operates in inquiry mode and is used to report on the status of *APL* program execution. It is issued as shown below:

```
      )SI
FUNC2[1] *
FUNC1[1]
```

The *)WSID* command may be used in both action and inquiry mode. In action mode, *)WSID* assigns the name included in the command line as the new name of the active workspace and returns the previous name of the workspace. In inquiry mode, *)WSID* is issued without an argument and returns the current name of the active workspace. The following examples illustrate the two forms of the *)WSID* command.

```
      )WSID
BOZO
```

5.1.3 APL Workspaces

The *APL* system uses a buffer in the user's memory area to store functions, variables, values, information on the status of functions, and any temporary results obtained while executing *APL* statements. When available in memory, this buffer area is known as the *active workspace*.

The user may issue system commands that cause this active workspace to be saved on a secondary storage device; the saved workspace can subsequently be loaded into the buffer area to function as the active workspace once again. The term "workspace" is used to refer either to the active workspace or to a version of an active workspace now saved on secondary storage.

Many of the system commands described in this chapter have been implemented to facilitate workspace-manipulation operations. The *APL* user has extensive control over the activity and characteristics of the workspace in his system. The workspace can be cleared, named, saved, loaded, and deleted. The names of functions and variables in the active workspace can be displayed. The user can change such active workspace characteristics as index origin setting, number of significant digits in output, and comparison tolerance.

Each *APL* workspace defined in a user's disk area has a unique name associated with it. This workspace name is represented by the *filename* parameter in many of the system command formats included in this chapter and in Chapter 6.

In RT-11 systems, *filename* has the following format:

device:name.ext[size]

All of these fields are optional. The *name* component must usually be supplied, but can be omitted if an output *device* name is specified, as in the filename LP:. If a file *size* is specified, it must be enclosed in square brackets.

In RSTS/E systems, several additional components may be included in the *filename* format, as shown in the following example:

device:name.ext<prot>[prj,prg]/SIZE:size/CLUSTER:clus/MODE:mode

As in the RT-11 format, all fields are optional.

In RSX-11M and IAS systems, *filename* has the following format:

device:[uic]name.ext;version

In all systems, a comma should be inserted instead of a period to separate the *name* and *ext* components when an ASCII terminal is being used (see Section 1.3.2).

Table 5-1 summarizes the characteristics of each *filename* component. Alphanumeric characters included in *device*, *name*, and *ext* fields may be letters (*A-Z*) and numbers (0-9).

Detailed information on these filename components is included in the BASIC-PLUS LANGUAGE MANUAL.

Examples of legal filenames are included below.

```
RTFILE.TXT[3]
RSTSFL.TXT/SIZE:3
RSXFIL.TXT;2
```

(RT-11 only)
(RSTS/E only)
(RSX-11M and IAS only)

Table 5-1
Filename Components

Component	Meaning
All systems:	
device	Valid device name with optional unit number, followed by a colon - for example: *LP:* *DT*5: The default device name is *SY*:.
name	Filename consisting of a maximum of six alphanumeric characters (nine for RSX-11M), beginning with a leter - for example: *TEMP* *FIL*001 There is no default.
ext	Period or comma, followed by a maximum of three alphanumeric characters - for example: .*TMP* ,*APL* For most system commands, the default extension is .*APL*. For the *)SAVE* and *)LOAD* commands, the default is .*APC*.
RT-11 only *size*	Size of the file in blocks of 512 bytes each. The size is used in reserving room for output files and by the *)CREATE* command (Section 6.3.2) to allocate space for new files. It must be enclosed in square brackets - for example: [2048]
RSTS/E only:	
prot	Protection code used in creating a new file. A code of <40> allows other users to read but not to alter a file. It must be enclosed in angle brackets - example: <40> Default is the system default.
prj, prg	Project-programmer number (in decimal) of the disk area in which the file is stored. It must be enclosed in square brackets - for example: [7,31] Default is the user's project-programmer number.

Table 5-1 (Cont.)
Filename Components

Component	Meaning
size	Slash, followed by the size of the file in blocks of 512 bytes each. It must be specified as a /SIZE switch - for example: /SIZE:16 There is no default.
clus	Slash, followed by the cluster size associated with the file. It must be specified as a /CLUSTER switch - for example: /CLUSTER:64 There is no default.
mode	Slash, followed by the mode associated with the file. It must be specified as a /MODE switch - for example: /MODE:1 There is no default.
RSX-11M and IAS only:	
uic	Project-programmer number (in octal) of the disk area in which the file is stored. It must be enclosed in square brackets - for example: [100,1] Default is the current user default.
version	A single octal number in the range 1-77777 representing the desired version of the file. (Note that RSX-11M and IAS allow multiple versions of a single file to be stored.) Default is the highest available number.

5.2 BASIC WORKSPACE-CONTROL COMMANDS

This section describes the basic workspace-control commands, which allow the user to manipulate *APL* workspaces in a variety of ways:

- Clear and name the active workspace
- Save the active workspace on a secondary storage device and retrieve it when required
- List workspace names
- Delete workspaces or files when no longer needed

5.2.1 *)CLEAR*: Clearing the Active Workspace

Format: *)CLEAR*

Example:
```
      )CLEAR
CLEAR WS
```

The *)CLEAR* system command operates in action mode. It closes all open files and clears the active workspace by replacing it with a special workspace known as the *clear workspace*. There are a number of characteristics associated with this special workspace. The clear workspace:

1. contains no functions, variables, or open files
2. has an index origin of 1
3. has an output line length of 72
4. displays numbers with six (single-precision) or ten (double-precision) significant digits
5. has a comparison tolerance (fuzz) of $5E^{-}7$ (single-precision) or $5E^{-}15$ (double-precision)
6. has a clear symbol table and state indicator

In RSTS/E systems, the file named $APLCLR.APC is used to clear the active workspace. If this file cannot be found in the system, the *APL* session will immediately be terminated and control will return to BASIC, which will display the "Ready" message.

5.2.2 *)WSID*: Identifying the Active Workspace

Format: *)WSID [filename]*

Examples:
```
      )WSID BOZO          (Names the active workspace)
WAS CLEAR WS
      )WSID               (Returns name of active work-
BOZO                       space)
```

The *)WSID* system command may be used in both action and inquiry mode. As an action command, *)WSID* allows the user to change the name of the active workspace. As an inquiry command, the *)WSID* command returns the current name of the active workspace. The *filename* parameter is required in action mode, but the user need not specify all components of the workspace name (see Section 5.1.3). When parts of the name are omitted, the default values summarized in Table 5-1 are assumed.

As illustrated in the examples above, the *)WSID* system command returns a workspace name in both action and inquiry mode. In inquiry mode, the name displayed is the current name of the workspace. In action mode, the name displayed is the workspace name before the user changed it by means of the *)WSID* command. When *)WSID* returns a workspace name, it displays only the name, not the other parts of the *filename*.

5.2.3 *)SAVE*: Saving a Copy of the Active Workspace

Format: *)SAVE [filename]*

Examples:

```
      )SAVE                         (Clear workspace cannot be
NOT SAVED, WS IS CLEAR WS           saved)
      )SAVE BOZO                    (Change name of active
SAVED 14:05:16   5-MAR-79 BOZO      workspace)
      )SAVE                         (Save active workspace under
SAVED 14:05:23   5-MAR-79 BOZO      default name)
      )WSID FOOBAR                  (Change name of active
WAS BOZO                            workspace)
      )SAVE                         (Save active workspace under
SAVED 14:05:37   5-MAR-79 FOOBAR    default name)
      )WSID FOO                     (Change name of active
WAS FOOBAR                          workspace)
      )SAVE FOOBAR                  (Duplicate of existing file
NOT SAVED, WS IS FOOBAR             cannot be saved unless the
                                    specified name is also the
                                    name of the active workspace)
```

The *)SAVE* system command is an action command that saves a copy of the active workspace on a secondary storage device. The saved workspace may be stored as a file in core-image format on disk, floppy disk, DECtape, or magnetic tape. If a *filename* parameter is included, *)SAVE* stores the active workspace under the specified name. If the *filename* parameter is omitted, *)SAVE* stores the workspace under the current name of the active workspace. In both cases, the default file extension is .APC. *APL* substitutes the default components described in Table 5-1 for any other missing *filename* components.

APL does not allow the user to save the clear workspace (see the first *)SAVE* in the sequence of examples above). *APL* also attempts to prevent users from accidentally destroying saved files. If the *filename* specified in the *)SAVE* command is identical to the name of an existing file but different from the workspace filename of the currently active workspace, then *APL* refuses to save the workspace (see the last example above).

The *)SAVE* system command responds to the user's specification by displaying the time and date.

When a workspace is *)SAVE*d, the following values are preserved:

- symbol table
- current contents of state indicator
- value of index origin
- output line width
- number of significant digits
- relative fuzz factor
- current random number sequence

All open files are closed automatically before the workspace is saved. Once a file has been saved in core-image format, it may only be retrieved from secondary storage by the *)LOAD* system command (Section 5.2.4).

If the user saves the active workspace while a function is executing, the function will be interrupted before the *)SAVE* is performed. When the workspace is subsequently loaded, execution of the interrupted function will resume automatically.

5.2.4 *)LOAD*: Retrieving a Workspace

Format: *)LOAD filename*

Examples:

```
        )SAVE STRDFY                        (Save active workspace)
SAVED 14:46:59    5-MAR-79 STRDFY           (Clear active workspace)
        )CLEAR
CLEAR WS                                    (Reload file as active
        )LOAD STRDFY                        workspace)
SAVED 14:46:59    5-MAR-79
```

The *)LOAD* system command operates in action mode and retrieves a workspace from such secondary storage devices as disk, floppy disk, DECtape, and magnetic tape. The workspace that is loaded becomes the active workspace, replacing the currently active workspace. The workspace specified in the *filename* parameter must be a core-image file that was saved by means of a *)SAVE* command (Section 5.2.3). The default extension for the file being loaded is .APC. *APL* substitutes the default components described in Table 5-1 for any other missing *filename* components.

The *)LOAD* system command responds to the user's specification by displaying the word *SAVED*, followed by the time and date when the workspace was saved.

5.2.5 *)LIB*: Listing Workspace Names (RSTS/E, RSX-11M, and IAS only)

Format: *)LIB [filename]*

Examples:

```
        )LIB

        )SAVE WS41
SAVED 15:21:34    5-MAR-79 WS41
        )LIB
WS41

        )LIB WS50.*
WS50.
WS50.FIL
WS50.VAR

        )LIB COS.MAC
COS.MAC
```

The *)LIB* system command operates in inquiry mode. It is used to display a list of workspaces in the user's disk area or selected files on any directory device. *)LIB* assumes that any file in the user's disk area with the extension *.APC* contains a workspace.

The files displayed by *)LIB* need not be *APL* workspaces. If the *filename* parameter is included in the command, the user can specify the filename or category to be displayed. The *filename* specification can identify a particular file or can serve as a "wild-card" reference when an asterisk is substituted for the file-name and/or the file extension. The asterisk matches any name. For example:

```
)LIB WS40.*
```

will list the names of all files that have *WS40* as their filename. Another example of this usage is shown in the third *)LIB* in the sequence of examples above. The command:

```
)LIB DSKH:*.*
```

will list the names of all files on device *DSKH:*.

If the *filename* is omitted from the *)LIB* command, all workspaces in the user's disk area will be displayed.

5.2.6 *)DROP*: Deleting Stored Workspaces or Files

Format: *)DROP filename*

Example:

```
      )DROP STRDFY
14:51:04    5-MAR-79
```

The *)DROP* system command operates in action mode and allows the user to delete from secondary storage the workspace or file identified in the *filename* parameter. *)DROP* can be used to delete any system file for which the user has the necessary protection privileges. A default extension is not supported for the *)DROP* command, so an explicit extension name must be supplied in the *filename* parameter.

5.3 WORKSPACE-CONTENT COMMANDS

This section describes the system commands that facilitate the examination of functions and variables in the user's workspace. The following operations can be performed:

- Display a list of variables defined in the active workspace
- Display a list of functions defined in the active workspace
- Erase defined functions and variables
- Display the *APL* state indicator to report on the execution of functions in the workspace.

5.3.1 *)VARS*: Displaying a List of Global Variables

Format: *)VARS [letter]*

Examples:

```
      )VARS
A AAA ABC B C D ZZZ
      )VARS A
A AAA ABC B C D ZZZ
      )VARS C
C D ZZZ
      )VARS Z
ZZZ
```

The *)VARS* system command operates in inquiry mode and displays an alphabetical list of names defined as global variables in the active workspace. The optional parameter specification identifies the letter at which the alphabetical listing is to begin. If the parameter is omitted, the entire set of global variable names is displayed.

5.3.2 *)FNS*: Displaying a List of Functions

Format: *)FNS [letter]*

Examples

```
      )FNS
DMD INSTR MMD NUM WUMPUS
      )FNS N
NUM WUMPUS
```

The *)FNS* system command is an inquiry command. It displays an alphabetical list of global names used as defined function names in the active workspace. The optional parameter specification identifies the letter at which the alphabetical listing is to begin. If the parameter is omitted, the entire set of global function names is displayed.

5.3.3 *)GROUP*: Defining or Dispersing a Group

Format: *)GROUP group-name [group-member-list]*

Examples:

```
)GROUP FINANCIAL INTEREST FUTUREVAL PRESENTVAL
)GROUP FINANCIAL
)GROUP FINANCIAL TAX FINANCIAL
```

The *)GROUP* system command operates in action mode. It is used to place a collection of named objects under one group name and to disperse an existing group. The objects may be variables, functions, and other group names. The *)GROUP* command is used primarily in conjunction with the *)COPY* and *)PCOPY* commands. After collecting a set of functions and variables under one group name, the user can specify this name in a *)COPY* or *)PCOPY* command in order to copy the entire collection at one time. In the first example above, the functions and variables named *INTEREST*, *FUTUREVAL*, and *PRESENTVAL* are collected under the group name *FINANCIAL*.

In addition to its function in establishing a new group, the *)GROUP* system command can be used to disperse an existing group. If the *group-member-list* parameter is omitted and only the *group-name* is included in the command line, then the named group will be dispersed.

The group name will no longer be defined, but the individual members of the group will be preserved under their original names. In the second example above, the group named *FINANCIAL* is eliminated. The members of the group, *INTEREST*, *FUTUREVAL* and *PRESENTVAL* are unaffected.

The)*GROUP* command can be used to add a new member to an existing group. To accomplish this task, the user specifies the group name itself as an element in the member list, as illustrated in the third example above. In this case, the function named *TAX* is added to the existing group named *FINANCIAL*. The following example illustrates another use of this feature.

```
)GROUP GEOMETRY ANGLE ACUTE OBTUSE
)GROUP GEOMETRY GEOMETRY PYTHAG ANGL1 ANGL3
```

5.3.4)*GRP*: Displaying the Members of a Group

Format:)*GRP group-name*

Examples:

```
      )GROUP ROOTS TRAPEZOID REGFALSI NEWTON SECANT
      )GRP ROOTS
TRAPEZOID  REGFALSI     NEWTON     SECANT
```

The)*GRP* system command is an inquiry command used to display the members of the group named in the command line. The members are listed in the order in which they were entered into the group.

5.3.5)*GRPS*: Displaying a List of Groups

Format:)*GRPS [letter]*

Examples:

```
      )GRPS
FINANCIAL     ROOTS

      )GRPS G
ROOTS
```

The)*GRPS* system command operates in inquiry mode and is used to display an alphabetical list of global names used as group names in the active workspace. The optional parameter specification identifies the letter at which the alphabetical listing is to begin. If the parameter is omitted, the entire set of group names is displayed.

5.3.6)*COPY*: Copying Objects from a Workspace

Format:)*COPY filename [named-object-list]*

Examples:

```
      )COPY MYWORK
SAVED  9:43:10  3-OCT-75

       )COPY MYWORK EXAM A B REG
SAVED  13:22:10  5-SEP-75

       )COPY MYWORK XYZ

NO SUCH FILE
```

The *)COPY* system command operates in action mode. It is used to retrieve functions, variables, and groups from a workspace called the copy workspace and to copy them into the active workspace. The user may copy all of the named objects in a workspace, or may copy only a subset. The *named-object-list* parameter can be used to identify the specific objects to be copied. If this parameter is omitted, all functions, variables, and groups in the workspace will be copied.

)COPY does not have the effect of copying the workspace itself. Local variables, the state indicator, and the width, origin, and significant digit settings are not transferred.

If objects in the copy workspace are homographs of (i.e., have the same name and characteristics of) objects in the active workspace, the objects in the active workspace will be replaced by their copy counterparts. However, homographs in the active workspace that are pendent functions or are functions in the process of being defined are not replaced. See the third sample command above for an example of a function of this kind.

If a group name is included in the *named-object list*, then all of the members of the group are copied along with the group name.

Named objects that cannot be found in the copy workspace or cannot be copied to the active workspace are displayed, as shown in the third example above. The format of the *)COPY* command response is identical to that of the *)LOAD* command described in Section 5.2.4.

5.3.7 *)PCOPY*: Copying from a Workspace with Protection

Format: *)PCOPY filename [named-object-list]*

Examples:

```
      )PCOPY MYWORK F PLUSROW PRIMES A
SAVED  11:02:21  21-APR-75
NOT COPIED:  A

      )PCOPY MYWORK G B F
SAVED  11:02:21  21-APR-75
NOT FOUND:  G
```

The *)PCOPY* system command operates in action mode. Its format is identical to that of *)COPY*, but it is used to protect functions, variables, and groups in the active workspace from accidental destruction. Unlike *)COPY*, the *)PCOPY* command does not replace objects in the active workspace that are homographs of objects in the copy workspace.

When copying groups, the *)PCOPY* command does not copy any members of the group that have homographs in the active workspace. If the group name itself has a homograph in the active workspace, then *)PCOPY* will not copy the group name but will copy all members of the group that do not have homographs in the active workspace.

The format of the *)PCOPY* command response is identical to that of the *)COPY* (Section 5.3.6) and *)LOAD* (Section 5.2.4) commands. Named objects that cannot be found in the copy workspace or cannot be copied to the active workspace are displayed, as shown in the examples above.

5.3.8 *)ERASE*: Erasing Global Names

Format: *)ERASE name-list*

Examples:

```
      A←2 3 4
      A
2 3 4
      B←C←0
      )ERASE A B
      A

VALUE ERROR
      A
      ↑

      )FNS
F1     F2

      )SI
F2[1] *
F1[2]

      )ERASE F1
NOT ERASED:    F1

      →30
      )SI                    (Clear the state indicator)
      )ERASE F1 F2
      )FNS
```

The *)ERASE* system command operates in action mode. It erases names from the active workspace by undefining the global functions and variables specified in the *name-list* parameter. There may be any number of names in the list. The names must be separated by at least one space.

)ERASE may not be used to erase a function whose name appears in the state indicator (see Section 5.3.9). Examples of such attempts to erase pendent and suspended functions are included in the sequence at the beginning of this section.

5.3.9 *)SI*: Displaying the State Indicator

Format: *)SI*

Example:

```
      )SI
INSTR[2] *
WUMPUS[3] *
```

The *)SI* system command is an inquiry command that displays the state indicator associated with the active workspace. The state indicator serves as a report on the execution of functions in the workspace. By analyzing the *)SI* listing, the user can determine such function status conditions as the following:

- pendent functions
- suspended functions

- pending quad input requests
- operations involving the execute operator

The format of the *)SI* display line indicates the particular status of the function. A function name followed by a bracketed line number indicates that the function stopped at that line number. If an asterisk (*) follows the bracketed line number, the function is currently suspended. If the asterisk is omitted, the function is pendent, that is, awaiting a return from another function.

The order in which function names are displayed in the *)SI* list is significant; the function that was most recently active is listed first. In the example included at the beginning of this section, *WUMPUS* called *INSTR* at line number [1]. Function *INSTR* was then suspended at line [3]. Execution of *INSTR* can be resumed by typing →1.

The state indicator also reports on pending quad input requests and pending execute operations (Sections 2.7.24 and 5.6). A quad input request is indicated by a quad character (⎕) in the *)SI* display line, and an execute request by an epsilon character (ϵ). Both of these conditions are illustrated in the example included below.

```
      ⍎⎕                                (Execute an evaluated input)
⎕:
      )SI                               (List the state indicator)
⎕
⎕:
      1
1
```

The use of the state indicator is discussed in terms of function execution in Section 3.4.4.

5.3.10 *)SIV*: Displaying the State Indicator and Local Variables

Format: *)SIV*

Examples:

```
      )SIV                      (Local variables B and I defined for
IMAX[4] *  B  I                  function IMAX)

      )SIV                      (Local variables A and B defined for
Z[2] *  A  B                     function Z)
```

The *)SIV* system command operates in inquiry mode. Like the *)SI* command (Section 5.3.9), it displays the state indicator associated with the active workspace and reports on pendent and suspended functions, pending quad input requests, and execute operations. In addition to this information, however, *)SIV* also displays a list of local variable names defined for each pendent or suspended function. The names of global variables used in the function are not displayed.

5.4 WORKSPACE-ENVIRONMENT COMMANDS

This section describes a variety of system commands that allow the user to display and control the characteristics of the workspace environment. These commands perform such tasks as the following:

- Specify the index origin setting
- Specify the maximum number of significant digits to be displayed in *APL* output
- Set the width of the output line
- Set the fuzz or comparison tolerance

5.4.1 *)ORIGIN*: Determining the Index Origin

Format: *)ORIGIN* $\begin{bmatrix}0\\1\end{bmatrix}$

Examples:

```
          ⍳5
1   2   3   4   5
          )ORIGIN 0
WAS 1
          ⍳5
0   1   2   3   4

          )ORIGIN
0
```

The *)ORIGIN* system command can be used in either action or inquiry mode. As an action command, *)ORIGIN* allows the user to change the setting of the index origin for array operations and returns the previous setting. In inquiry mode, the *)ORIGIN* command returns the current setting of the index origin. A parameter (0 or 1) is required in action mode. The default setting is 1.

The effect of the *)ORIGIN* system command is to renumber the elements of arrays to begin at zero or one, depending on the index origin setting. This command is particularly relevant when used in conjunction with the *APL* iota operator (Sections 2.7.5 and 2.7.6)for a more detailed discussion of the index origin, see Section 2.4.2.

)ORIGIN is equivalent to the ⎕*IO* system variable (Section 4.2.2). The index origin setting is preserved when the active workspace is saved.

5.4.2 *)DIGITS*: Determining the Output Precision

Format: *)DIGITS [n]*

Examples:

```
      )DIGITS
7
      B
1.234567

      )DIGITS 3
WAS 7
      B
1.23
```

The *)DIGITS* system command operates in either action or inquiry mode. As an action command, *)DIGITS* can be used to specify the maximum number of significant digits to be displayed in *APL* output; it returns the previous maximum number. In inquiry mode, the *)DIGITS* command returns the number of significant digits currently being displayed. A parameter must be included in action mode to specify the number of significant digits to be displayed. The default number of digits is 10 for double-precision systems and 6 for single-precision. Legal values are integers in range 1 through 17 for double-precision systems and 1 through 7 for single precision.

The *)DIGITS* system command does not affect the precision of internal calculations or the display of numeric constants. See Section 2.2 for an example of formatting numeric output.

)DIGITS is equivalent to the *⎕PP* system variable (Section 4.2.3). The precision setting is preserved when the active workspace is saved.

5.4.3 *)WIDTH*: Determining the Width of the Output Line

Format: *)WIDTH [n]*

Examples:

```
      )WIDTH 50
WAS 120
      ⍳15
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15

      )WIDTH 30
WAS 50
      ⍳15
1  2  3   4   5   6   7   8   9
     10  11  12  13  14  15

      )WIDTH
30
```

The *)WIDTH* system command can be used in either action or inquiry mode. As an action command, *)WIDTH* allows the user to set the maximum number of characters that may appear in an output line and returns the width previously in effect. In inquiry mode, the *)WIDTH* command returns the current width of the output line. The *n* parameter must be included in action mode to specify the maximum number of characters in the output line; it must be an integer in range 30 through 133 inclusive. The default setting is 72, except in the RSTS/E environment, in which *APL* defaults to the current user width.

The *)WIDTH* system command does not affect the display of messages on the terminal or the allowable length of input lines. *)WIDTH* is equivalent to the *⎕PW* system variable (Section 4.2.4). The width setting is preserved when the active workspace is saved.