HONEYWELL

MULTICS APL USER'S GUIDE

SOFTWARE

MULTICS APL
USER'S GUIDE

SUBJECT

Description of the Multics Implementation of APL

SPECIAL INSTRUCTIONS

This edition of the APL manual supersedes the previous edition of the manual, Order Number AK95-01, dated March 1979 and its associated addendum, AK95-01A, dated May 1980. This edition does not contain marginal change indicators.

Section 7 (System Functions), Section 8 (System Variables), Section 9 (Stream I/O), Section 10 (External Functions), and Section 11 (APL Sample Programs) are new. See the Preface for a complete description of all changes to the document.

SOFTWARE SUPPORTED

Multics Software Release 11.0

ORDER NUMBER

AK95-02

December 1985

Honeywell

Preface

This manual describes the Multics implementation of APL (A Programming Language). The document assumes no prior knowledge of APL.

The manual does not attempt to provide the reader with extensive information on the Multics system. The reader is referred to the *Multics Programmer's Reference Manual* (Order No.: AG91) or the *Introduction to Programming on Multics* (Order No.: AG90) for details on programming in the Multics environment.

Section 1 briefly describes the characteristics of APL and the nature of the Multics implementation.

Section 2 describes Multics APL processing conventions.

Section 3 is a description of the APL language.

Section 4 lists APL functions that can be created and modified by the APL function editor.

Section 5 provides information on the system commands that can be used to adjust or control the operation of APL.

Section 6 describes the Multics APL file system.

Section 7,8,9, and 10 describe, respectively, Multics APL system functions, system variables, stream I/O, and external functions.

Section 11 contains APL sample programs.

A glossary of terms is provided in Appendix A.

Appendix B lists the Multics commands that relate to APL.

Significant Changes in AK95-02

The character representations assigned to octal codes 044 and 045 were changed. See Table 2-1 on page 2-3.

The description of "Escape Processing" on page 2-13 was changed to add information describing the proper procedure for removing an escape sequence and also to indicate that, if the character immediately following the escape character does not follow the stated rules, then that character (and the escape character) will not appear in the input line.

On page 3-4, the Note attached to the description of "Small Numbers" was changed.

On page 3-5, there are new examples accompanying the description of "Large Numbers."

On page 3-54, the example illustrating use of "Inner Product" is new.

The description of "Mixed Output" on page 3-61 was changed to indicate that a pair of semicolons are the only characters that can be used as delimiters.

The description of "Character Input" on page 3-74 was changed to indicate that a strong interrupt can be generated if it is entered before any characters are typed.

There are new APL language elements in new Sections 7, 8, 9, and 10.

A new Section 11 describes APL sample programs.

CONTENTS

		Page
Section 1	Introduction	1-1
	History of APL	1-1
	Characteristics of APL	1-1
	The APL Character Set	1-2
	Multics APL	1-2
Section 2	Communicating With Multics APL	2-1
	Calling APL	2-1
	APL-only Users	2-1
	Input Prompt	2-2
	APL Character Set	2-2
	Internal Codes	2-2
	Terminal I/O Conventions	2-8
	Overstrikes	2-8
	Tabulating	2-9
	EBCDIC Terminals	2-9
	Ascii Terminals	2-10
	Input Line Processing	2-10
	Canonicalization	2-11
	Erase, Kill, and ATTN Processing	2-11
	Kill	2-11
	Erase	2-11
	ATTN	2-12
	Escape Processing	2-13
	Using APL	2-13
Section 3	The APL Language	3-1
	Values	3-1
	Type	3-1
	Rank	3-2
	Shape	3-2
	Output of Values	3-3
	Shape Operator	3-3
	Elements	3-3
	Scalars	3-4
	Scientific Notation	3-4
	Small Numbers	3-4
	Large Numbers	3-5
	Vectors	3-5
	Character	3-5
	Numeric	3-6
	Matrices and Arrays	3-6

CONTENTS (cont)

Page

Matrices 3-6
Arrays 3-6
Character 3-7
Input of Values 3-8
Numeric 3-8
Character 3-8
Numeric 3-9
Operators 3-10
Matrices and Arrays 3-10
Scalar Operators 3-11
Extension to Non-Scalar Arguments 3-11
Monadic 3-11
Dyadic 3-12
General Properties of Scalar Operators 3-13
General Properties of Monadic Scalar Operators 3-13
General Properties of Dyadic Scalar Operators 3-14
Add, Subtract, Multiply, Divide + - × ÷ 3-15
Plus, Negative + - 3-15
Signum × 3-16
Reciprocal ÷ 3-16
Power, Logarithm * ⍟ 3-16
Residue | 3-16
Magnitude | 3-17
Factorial ! 3-17
Binomial Coefficients ! 3-17
Maximum and Minimum ⌈ ⌊ 3-18
Ceiling and Floor ⌈ ⌊ 3-18
Roll ? 3-19
Comparison Operators < ≤ = ≠ ≥ > 3-19
Generating Algorithm 3-19
Logical Operators ~ ∧ ∨ ⍲ ⍱ . . 3-20
Circular ○ 3-21
PI Times ○ 3-21
Mixed Operators 3-21
Shape ⍴ 3-22
Reshape ⍴ 3-22
Ravel , ... 3-23
Catenate , ⍪ 3-23
General Rules for Catenate 3-28
Laminate *A*,[*I*]*B* 3-31
General Rules For Laminate 3-35
Index Generator ⍳ 3-37
Index of ⍳ 3-38
Take, Drop ↑ ↓ 3-38

CONTENTS (cont)

Page

Grade Up, Grade Down ⍋ ⍒ 3-39
Reverse ⌽ ⊖ 3-40
Rotate ⌽ ⊖ 3-40
Transpose ⍉ 3-41
Compress / ⌿ 3-42
Expand \ ⍀ 3-43
Membership ∊ 3-43
Encode ⊤ 3-44
Decode ⊥ 3-44
Deal ? 3-44
Matrix Inverse ⌹ 3-45
Matrix Divide ⌹ 3-45
I-Beam ⌶ 3-46
Format ⍕ 3-47
Composite Operations 3-49
Reduction ⊙/ ⊙⌿ 3-50
Scan ⊙\ ⊙⍀ 3-52
Outer Product ∘.⊙ 3-53
Inner Product ⊙.⊙ 3-54
Expressions 3-54
Right-to-Left Rule 3-55
Subexpressions 3-55
Implicit Subexpressions 3-56
Explicit Expression Delimiters ();[]⍝:◇ 3-57
Explicit Subexpressions 3-57
Parenthesized Expressions and Subexpressions 3-58
Lists 3-58
Indexing 3-59
Mixed Output 3-61
Argument Lists 3-62
Comments 3-63
Labels 3-63
Statements 3-63
Closed Expressions 3-64
Objects 3-65
Diamond Lines 3-65
Names 3-65
Syntax of Names 3-66
Identifiers 3-66
Variable Names 3-66
Function Names 3-67
Group Names 3-67
Variables 3-67
Functions 3-68
Groups 3-68
The Assignment Pseudo-Operator ← . . . 3-69
Indexed Assignment 3-70

CONTENTS (cont)

Page

Results 3-70
The I/O Pseudo-Variables ⎕⍞ 3-71
Output of Explicit Results 3-71
Evaluated Input ⎕ 3-72
Character Input ⍞ 3-74
Output ⎕⍞ 3-74
The Execute Pseudo-Operator ⍎ 3-75
Pornography: Dependence Upon Undefined Evaluation Order 3-76
Workspaces 3-79
Example 1 3-79
Example 2 3-79
The Active Workspace 3-80

Section 4 Functions 4-1
Functions 4-1
Arguments 4-1
Results 4-1
Local Identifiers 4-2
Local Objects 4-3
Global Identifiers 4-3
Global Objects 4-4
Immediately Local Identifiers and Objects 4-4
The Function Header 4-4
Syntax Definition 4-4
Argument Identifiers 4-5
Result Identifier 4-6
Local Identifier List 4-6
The Function Body 4-7
Line Numbers 4-7
Execution Flow 4-7
The Branch Pseudo-Operator → . . . 4-8
Function Return 4-10
Labels 4-11
Recursion 4-12
Implicit Results 4-12
Scalar Functions 4-13
Trace Pseudo-Variables *T*∆name_of_fn 4-14
Locked Functions 4-16
The State Indicator 4-17
Execution Termination 4-17
Attn; Weak & Strong Interrupts . 4-17
Weak Interrupt 4-17
Strong Interrupt 4-17
Error Handling 4-18
Suspension 4-18
Syntax and Context Errors . . 4-19

CONTENTS (cont)

Page

Stop Pseudo-Variables
S∆name_of_fn 4-19
Impl$\overline{\text{ici}}$t Results 4-19
Locked Functions 4-20
Halted Function Calls 4-21
Suspended Function Calls . . 4-21
Pendent Function Calls . . . 4-21
The)*SI* System Command 4-22
The)*SIV* System Command 4-22
Exploring the SI 4-22
The ⎕*SI* System Function 4-22
*I*27, *I*26 and ⎕*LC* 4-23
The Escape → 4-23
Clearing the SI 4-23
Restarting a Suspended Function Call 4-23
SI Damage 4-24
Effects 4-25
Editing Halted Functions 4-25
Editing A Function 4-26
Editing A Line 4-28

Section 5 System Commands 5-1
Environment Parameters 5-1
Workspace Parameters 5-2
Session Parameters 5-2
The)*ORIGIN* System Command 5-3
The)*WIDTH* System Command 5-3
The)*DIGITS* System Command 5-4
The)*ERRS* System Command 5-4
The)*TABS* System Command 5-5
The)*CHECK* System Command 5-5
The)*HUH* System Command 5-5
The)*VARS* System Command 5-5
Symbol Table 5-5
The)*FNS* System Command 5-6
The)*GROUP* System Command 5-6
The)*GRP* System Command 5-7
The)*GRPS* System Command 5-7
The)*ERASE* System Command 5-7
The)*SYMBOLS* System Command 5-8
The)*SI* System Command 5-8
The)*SIV* System Command 5-9
Workspace Management 5-10
Workspace Identification 5-10
Passwords 5-11
The)*CLEAR* System Command 5-12
The)*LIB*,)*LIBD* System Command . . 5-12
The)*SAVE* System Command 5-12

CONTENTS (cont)

		Page
	The)*SAVE* System Command	5-12
	The)*LOAD* System Command	5-14
	The)*COPY* System Command	5-14
	The)*PCOPY* System Command	5-16
	The)*CONTINUE* System Command . . .	5-16
	The)*WSID* System Command	5-16
	The)*DROP* System Command	5-17
	Version 1 APL Workspaces	5-17
	Communicating With Multics	5-17
	The)*Q*,)*QUIT*,)*OFF* System Commands	5-17
	The)*PORTS* System Command	5-18
	The)*EXEC*,]*E* System Commands . . .	5-18
	The)*HELP* System Command	5-18
	The)*MSG* System Command	5-19
	The)*DFN*,)*MFN*,)*ZFN* System Commands	5-19
	External Functions	5-19
	Definition Syntax	5-20
	Definition Errors	5-20
	External Functions Cannot Be Edited	5-20
	External Function Calling Sequence	5-21
	Status Codes for Use by External Functions	5-23
	Conventions for Using the Value Stack	5-24
	External Function Include Segment .	5-25
	apl_push_stack_	5-26
	Usage	5-26
Section 6	File System	6-1
	The Multics APL File System	6-1
	Organization of APL Files	6-1
	Use of APL Files	6-2
	File Manipulation Functions	6-2
	Access Control	6-8
	File Sharing	6-9
Section 7	System Functions	7-1
	The Multics APL System Functions . . .	7-1
	Active Function	7-3
	Accounting Information	7-5
	Call	7-6
	Canonical Representation	7-8
	Character Set	7-9
	Delay	7-10
	Execute Command	7-11
	Expunge	7-12
	Fix	7-13
	Line Counter	7-15

CONTENTS (cont)

		Page
	Name Count	7-16
	Name List	7-18
	Time Stamp	7-20
	Terminal Type	7-21
	User Load	7-22
	Workspace Available	7-23
	Workspace Used	7-24
Section 8	System Variables	8-1
	The Multics APL System Variables	8-1
	Comparison Tolerance	8-2
	Index Origin	8-3
	Integer Tolerance	8-4
	Latent Expression	8-5
	Printing Precision	8-6
	Page Width	8-7
	Random Link	8-8
Section 9	Stream I/O	9-1
	The Multics APL Stream I/O File System	9-1
	The Stream I/O Functions	9-2
	CREATE	9-2
	EOF	9-2
	NUMS	9-2
	POSITION	9-2
	READ	9-2
	REWIND	9-3
	TIE	9-3
	UNTIE	9-3
	WRITE	9-3
Section 10	External Functions	10-1
	The Multics APL External Functions	10-1
	apl_erf_	10-2
	apl_pickup_float_bin_2_	10-3
	apl_ioa_	10-4
	apl_get_list_nums_	10-5
	apl_read_segment_	10-6
Section 11	APL SAMPLE PROGRAMS	11-1
	APL Sample Programs	11-1
	APL Programming Style	11-1
	Executing a Sample Program	11-1
	Sample Programs	11-2
	Card Dealer	11-2
	Graph Plotting	11-4
	Get Month	11-12
Appendix A	Glossary	A-1

CONTENTS (cont)

Page

Appendix B Commands B-1
- apl, v2apl B-2
  - Usage B-3
  - Note B-4
- apl_end B-4
  - Usage B-4
  - Notes B-5
- apl_start B-5
  - Usage B-5
  - Notes B-5
- convert_tsoapl_workspace, ctw B-7
  - Usage B-7
- display_tsoapl_workspace, dtw B-8
  - Usage B-8
- read_tsoapl_tape, rtt B-9
  - Usage B-9
  - Notes B-9

SECTION 1

INTRODUCTION

HISTORY OF APL

A Programming Language (APL) originated as a mathematical notation for the discussion of the theory of algorithms. It was invented by Dr. Kenneth E. Iverson and was described by him in his book, A Programming Language.* The value of the notation as a practical means for expressing an algorithm to a computer was soon noticed. An interpreter which realized a subset of the notation was developed by IBM for its 7090 computer. The success of this pilot interpreter led to a second and more powerful implementation, known as APL\360, on the IBM System 360 series.

CHARACTERISTICS OF APL

The success of APL can be attributed to some characteristics which distinguish it from more conventional programming languages. First, it is interactive by design rather than by decree -- it is fast, succinct, forgiving, informative, and even fun to use. Next, it is at once both simple and powerful -- it is transparent and easy to learn; yet it attacks abstruse problems with ease.

APL can be characterized as a line-at-a-time desk calculator with many sophisticated operators and a stored-program capability. The user needs little or no prior acquaintance with digital computers to use it. After invoking APL, the user types an expression (or statement, or line) to be evaluated. The APL interpreter performs the calculations, prints the results -- if any, and awaits a new input line.

*John Wiley and Sons, 1962

The result of an expression evaluation can also be assigned to a variable and remembered from line to line. In addition, there is a capability for storing, by an assigned name, an ordered sequence of unevaluated APL lines. A later mention of the name causes the statements to be recalled and interpreted -- almost as if they had been entered from the terminal at the time.

Finally, there is the ability to save the entire current environment -- complete with all variable values and stored programs -- so that the user may continue at a subsequent APL session.

The APL Character Set

The APL language uses its own specially designed character set, in which each operator is represented by a single character. The most convenient access to APL is via a terminal with a complete APL character set. Among these are:

a) IBM 2741 -- and other Selectric-based terminals -- with an appropriate type sphere mounted: IBM part number 1167988 for BCD machines, or IBM 1167987 for correspondence terminals.

b) Selecterm System 75 -- and other terminals based on the Diablo HyType I or HyType II printers -- with an APL print wheel mounted: Diablo part number 38150.

c) Anderson Jacobson 630A -- and other terminals based on dot-matrix printers -- equipped with an APL ROM and some facility for switching to and from the APL character set (from and to the ASCII character set).

MULTICS APL

Multics APL behaves much like the other major commercially available APL systems (i.e.: Scientific Time Sharing's APL*PLUS, and IBM's APLSV). This minimizes the learning effort required of those already familiar with other APL's, and promotes compatibility at the source language level.

The Multics APL processor consists of three main components: the interpreter for the mathematical expressions of the APL language; a system command processor, which provides bookkeeping aids and maintains an environment within which the language runs; and an editor that is used to create and modify stored APL programs.

Multics APL fully supports all of the above listed terminals, and is usable from any other ASCII terminal as well, although the user must be aware of the typing conventions used to represent some of the APL characters within the framework of the available ASCII graphics.

SECTION 2

COMMUNICATING WITH MULTICS APL

CALLING APL

Normal Multics users must call Multics APL as a command:

```
apl {path} {-control_args}
```

Implemented arguments, their use, and their effects are documented in Appendix B of this manual, and in the *Multics Commands and Active Functions* manual (Order No.: AG92).

All arguments are optional.

APL-only Users

Some users are registered as APL-only users: after login, they are automatically encapsulated in the APL subsystem environment. Therefore, they need not and cannot issue any Multics command line that calls APL.

(Note: This distinction is not trivial. An APL-only user has a process overseer which, among other things, calls a special APL entry point as a subroutine. He is not permitted any direct access to the Multics command environment or storage system. Furthermore, upon leaving APL, he is logged out automatically.)

INPUT PROMPT

After invocation, APL responds by typing six spaces and awaiting input from the user. This not only indicates APL is waiting for information from the user, but also improves the readability of the terminal listing: most of the user-typed lines appear indented by six positions, while most of the APL-generated responses begin at the left margin.

Before typing any input, however, it must be determined how the APL character set is represented on the user's terminal. Since the APL character set differs significantly from the Multics standard character set, normal Multics typing conventions do not apply to communication with APL.

APL CHARACTER SET

In contrast to the 94 graphics of the Multics standard character set, the Multics APL character set has more than 150 graphics. Multics APL graphics are shown in Table 2-1, together with their internal codes, names, and printed representation on both APL and non-APL terminals.

Internal Codes

The internal code assigned to each character is not normally of significance to the APL user. There is no mechanism within the formal APL language to discover or make use of the internal representation of a character. However, there are occasions on which the Multics APL user may need to know the internal code assignments:

1) while in APL, the user may enter any APL character (graphic or nongraphic) with an escape sequence using a 3-digit octal code; for example, the new page (014) control code may be entered and used as character data;

2) in unusual cases, it may be desired that data and/or programs created within APL be used as data by programs external to APL. (This does not apply to APL-only users.)

To simplify code mapping, and to minimize learning effort, the Multics APL code assignments agree with the Multics ASCII code assignments wherever any correspondence of graphics between the two character sets can be found.

Table 2-1. APL Character Set

Printed Representation		Octal		
non-APL	APL	Code	Graphic	Name
(none)	(none)	007	(none)	bell
(none)	(none)	010	(none)	backspace
(none)	(none)	011	(none)	tabulate
(none)	(none)	012	(none)	new line
(none)	(none)	014	(none)	new page
(none)	(none)	015	(none)	carriage return
(none)	(none)	040	(none)	space
!	'.	041	!	exclamation point
"042	¨042	042	(none)	double quote
#	¨043	043	(none)	number sign
$	S|	045	$	dollar sign
"044	¨044	044	(none)	ampersand
%	¨046	046	(none)	per cent
'	'	047	'	quote (apostrophe)
(	(	050	(	open parenthesis
)	)	051	)	close parenthesis
*	*	052	*	star (asterisk)
+	+	053	+	plus
,	,	054	,	comma
-	-	055	-	hyphen (minus, bar)
.	.	056	.	dot (period)
/	/	057	/	slash
0	0	060	0	zero
1	1	061	1	one

Printed Representation		Octal		
non-APL	APL	Code	Graphic	Name
2	2	062	2	two
3	3	063	3	three
4	4	064	4	four
5	5	065	5	five
6	6	066	6	six
7	7	067	7	seven
8	8	070	8	eight
9	9	071	9	nine
:	:	072	:	colon
;	;	073	;	semi-colon
<	<	074	<	less than
=	=	075	=	equals
>	>	076	>	greater than
?	?	077	?	question mark
@	¨100	100	(none)	commercial-at
A_	*A*_	101	$\underline{A}$	capital A
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Z_	*Z*_	132	$\underline{Z}$	capital Z
[	[	133	[	open bracket
\	\	134	\	backslash
]	]	135	]	close bracket
"136	¨136	136	(none)	circumflex
_	_	137	_	underscore

Printed Representation		Octal		
non-APL	APL	Code	Graphic	Name
`	¨140	140	(none)	accent grave
A	*A*	141	*A*	A
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Z	*Z*	172	*Z*	Z
{	{ or [∘	173	{	open brace
¦	|	174	|	vertical bar (stile)
}	} or]∘	175	}	close brace
˜	~	176	~	tilde
(none)	(none)	177	(none)	pad
<_	≤	200	≤	less or equal
>_	≥	201	≥	greater or equal
=/	≠	202	≠	not equal
v	∨	203	∨	or
&	∧	204	∧	and
:-	÷	205	÷	divide
e	∊	206	∊	epsilon
¦^	↑	207	↑	up arrow
¦v	↓	210	↓	down arrow
o	○	211	○	circle
c	⌈	212	⌈	upstile
f	⌊	213	⌊	downstile
d	∆	214	∆	delta
o_	∘	215	∘	null

Printed Representation		Octal		
non-APL	APL	Code	Graphic	Name
q	⎕	216	⎕	quad
n	∩	217	∩	cap
¦_	⊥	220	⊥	base
t	⊤	221	⊤	top
(_	⊂	222	⊂	open shoe
)_	⊃	223	⊃	close shoe
u	∪	224	∪	cup
v˜	∨~	225	⍱	nor
&˜	∧~	226	⍲	nand
0-	○-	227	⊖	circle-hyphen
/-	/-	230	⌿	slash-hyphen
g˜	∇~	231	⍫	del-tilde
*o	*○	232	⍟	star-circle
o¦	○|	233	⌽	circle-vertical bar
o\	○\	234	⍉	circle-backslash
o/	○/	235	⌀	circle-slash
g¦	∇|	236	⍒	del-vertical bar
d¦	∆|	237	⍋	delta-vertical bar
n.	∩∘	240	⍝	lamp
'q	'⎕	241	⍞	quote-quad
b	⊥⊤	242	⌶	I-beam
\-	\-	243	⍀	backslash-hyphen
m	⎕÷	244	⌹	domino
"	¨	245	¨	dieresis
w	⍵	246	⍵	omega

Printed Representation non-APL	Printed Representation APL	Octal Code	Graphic	Name
i	⍳	247	⍳	iota
p	⍴	250	⍴	rho
x	×	251	×	times
a	⍺	252	⍺	alpha
^	¯	253	¯	overscore
g	∇	254	∇	del
<-	←	255	←	left arrow
>-	→	256	→	right arrow
<>	◊ or <>	257	◊	diamond
0_	0_	260	$\underline{0}$	zero-underscore
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
9_	9_	271	$\underline{9}$	nine-underscore
d_	Δ_	272	⍙	delta-underscore
(none)	(none)	273	(none)	mark error
e_	⊥∘	274	⍎	hydrant
f_	⊤∘	275	⍕	thron
(-	⊢ or (-	276	⊢	left tack
)-	⊣ or)-	277	⊣	right tack
(none)	(none)	300	(none)	line feed
(none)	(none)	301	(none)	conditional newline
c¦	⊂|	302	¢	cent sign
,-	,-	303	⍪	comma-hyphen

TERMINAL I/O CONVENTIONS

Overstrikes

Most APL graphics can be typed in one keystroke, regardless of terminal. However, many graphics are formed by overstriking two graphics. (Overstriking more than two different graphics is never either required or allowed.) In columns one and two of Table 2-1, this is indicated by an entry showing its two component graphics.

To simplify operations on character values, APL considers a graphic formed by an overstrike sequence to be a single character internally. Since each internal character must be assigned a unique octal code, and since Multics character operations are limited to characters with 3-digit octal codes, then there is a fixed limit of 512 possible characters. (So far, Multics APL has assigned meaning to 168 of them.) Therefore, there must be an even smaller limit on the number of valid overstruck characters. (Many codes must be assigned to nonoverstruck characters and to terminal-control and carriage-motion characters.) Since there are 4371 possible combinations of the 94 nonoverstruck graphics (taken two at a time) (2!94 in APL; see Section 3), most of these possible overstrikes cannot be valid. In fact, only those overstrikes explicitly defined in Table 2-1 are valid: typing any undefined overstrike results in a *CHAR ERROR* report, and automatic deletion of all input involved in, and to the right of, the invalid overstrike; the input to the left of the invalid overstrike is "replayed" back to the user, allowing completion of the input line.

The "visual fidelity rule" states, in effect, that "you get what you see." This implies that the temporal order of typing input is irrelevant; only the visual order is significant (see "Canonicalization" below). It further implies the following two conventions:

1) Redundant overstriking is allowed, but the redundancy is discarded. For example, overstriking the following 10 graphics results in a single domino character: ⎕÷÷÷⎕⎕÷⎕÷÷. (This is not in conflict with the two-graphic overstrike rule stated above, since only two graphics are involved.)

2) Invisible overstriking is allowed for terminals that produce APL graphics; the apparent character is used by APL. For example, due to the consistent graphic design of the APL lower case alphabetic characters, overstriking the *L* and the *F* produces a graphic which looks exactly like the *E*. APL replaces this overstrike with the letter *E*.

There are many overstrikes of this sort; the following are groups of alphabetics related in this way:

BPR CG EFL IT OQ

Various nonalphabetic groups also exist:

.,:; ÷:-. +- [⌈⌊ ←-→

=≠ !:'. ⍠-. ɛ⌊⌈ ?.

Note: Not all possible overstrikes of components of the above groups are necessarily meaningful: overstriking *B* and *R* is meaningless and is therefore invalid.

Tabulating

Multics APL initially assumes that terminal tab stops are set at intervals of 10 spaces (i.e.: columns 11, 21, 31,...). APL uses tabs to speed up output, and accepts tabs as input. However, all input tabs are immediately translated into the appropriate number of spaces: only "escaped-in" tabs are not translated (see ESCAPE PROCESSING below). If some other tab interval setting is used, APL must be informed via the)*TABS* system command, (see Sections IV and VI). Use of tabs for input and output may be suppressed via:)*TABS* 0 *OR* [] *HT* ← 0.

EBCDIC Terminals

EBCDIC terminals are Selectric-based, and therefore have interchangeable typing elements. Multics APL assumes that these terminals have an appropriate APL typesphere mounted: the second column of Table 2-1 should be used in determining how characters are represented on these terminals.

Note that five graphics in column two give a choice of representation: ¨173 (open brace), ¨175 (close brace), ¨257 (diamond), ¨276 (left tack), and ¨277 (right tack). On EBCDIC terminals, these graphics are represented in the overstruck form: ɛ ɜ ⁎ ⍅ ⍆. On APL/ASCII terminals (see below), they are represented as: {}]⊢⊣. This anomoly is due to the lack of room on Selectric typespheres for these new graphics.

Multics APL supports both BCD and Correspondence terminals, determining from Multics which type is being used. Note, however, that BCD and Correspondence terminals use different APL typespheres: unreadable I/O results from use of the wrong typesphere (BCD uses typesphere #988; Correspondence uses #987).

Ascii Terminals

ASCII terminals with interchangeable graphics have the potential capability of producing APL graphics, such as del ∇, quad ⎕, and upstile ⌈. However, to realize this potential, the terminal must be properly equipped. Specifically, terminals based on the Diablo HyType I printer must have the APL print wheel mounted (Diablo part #38150); terminals based on dot matrix printers must be equipped with an APL ROM and some mechanism for switching to and from APL (this includes CRT based terminals).

ASCII terminals with interchangeable graphics, equipped to produce APL graphics are called APL/ASCII terminals. All other ASCII terminals are called non-APL/ASCII, or simply ASCII.

APL/ASCII terminals can produce the full APL character set, including the five new graphics {}⋄⊢⊣, and therefore the second column of Table 2-1 should be used in determining how characters are represented.

Non-APL/ASCII terminals cannot produce most of the graphics of the APL character set, but reasonable and consistent mappings have been established for Multics APL, and are shown in column one of Table 2-1.

Multics APL assumes that every ASCII terminal is a non-APL/ASCII terminal, unless the -ttp or -terminal_type control argument is given when APL is called; see Appendix B. (For APL-only users, the process overseer makes the decision, supplying an appropriate terminal type identifier.)

Input Line Processing

CANONICALIZATION

As soon as backspaces are allowed in any typed line, it becomes evident that there are many different ways to type a given line. That is, there are many different sequences of keystrokes that produce visually identical results. To reduce confusion and allow greater freedom to the typist, APL <u>canonicalizes</u> each input line as it is read. This means that the characters typed by the user are sorted into their visual order on the page, independently of the temporal order in which they were typed. Hence, the user need not bother to type overstrikes in any specified order.

ERASE, KILL, AND ATTN PROCESSING

Typing errors in Multics APL may be corrected through either the Multics mechanism of erase and kill characters, or by backspacing and pressing the ATTN or BREAK key (as in other APLs). The Multics APL kill character is the alpha ⍺, and the corresponding erase character is the omega ⍵.

When using ASCII graphics, the Multics erase and kill characters may be used (# and @ respectively), or the ASCII mapped equivalents to the Multics APL erase and kill characters may be used (lowercase w and lowercase a respectively).

Kill

The kill character removes the entire line preceding it. That is, the kill character deletes itself, anything overstruck with it, and all characters to the left. Characters to the right of a kill character are not deleted.

Erase

The definition of erase is a little more complicated. If the erase character is overstruck with anything, then everything in that one print position is removed. If the erase character appears alone in a print position, then it and the character in the preceding print position are removed. If there is no character in the preceding position (i.e., it is white space), then the entire white space preceding the erase character is deleted.

Since erase and kill are performed <u>after</u> canonicalization, the spatial positions of the characters on the line determine which characters are removed: the order in which the characters were typed is not significant.

By convention, kill characters are processed before erase characters, but with one important exception: a kill overstruck with an erase results in the kill being erased. Therefore, the only way to erase a kill is to overstrike the kill and erase characters. ATTN may always be used to eliminate a kill character; see ATTN, below.

Note further that erase characters are processed one at a time, working from left to right. Therefore, several erases in succession do not erase each other; rather, they erase successive characters to their left. Furthermore, a nonoverstruck erase character may be effectively nullified by overstriking it with some printing graphic. However, an already overstruck erase character can never be erased or nullified.

ATTN

Pressing the ATTN key removes everything in and to the right of the current column. That is, it removes itself, anything "overstruck" with it, and everything to its right. Characters to the left are not affected.

When ATTN is pressed, APL sends (to the terminal) four characters which flag the use of ATTN: linefeed, "or" v, backspace, and linefeed. This sequence leaves the printer in the same column it was in when ATTN was pressed, except two lines lower, and with the v marking the effectively redirected flow of input.

The effects of using ATTN to edit the input line occur immediately: both the four-character response and the erasure of appropriate characters occur immediately after the ATTN key is pressed. Therefore, it is impossible to erase or nullify an ATTN after it is used.

Since the input line is canonicalized before each ATTN is processed, the spatial positions of the characters on the line determine which characters are removed. And since the remaining characters (after ATTN processing) replace the original line in the input line buffer, they are available for further editing, including the use of erase, kill, ATTN, and overstrikes, or simply appending further input.

An input line is not considered complete -- and therefore is not passed on for erase, kill, or escape processing -- until a carriage return (or a newline, for ASCII or APL/ASCII terminals) is received which is not part of a character constant (i.e., inside a quoted string). (Note that all such embedded carriage returns are translated to newlines. Furthermore, carriage return (or newline) -- embedded or not -- renders the input line up to that point unavailable for further ATTN processing.) Therefore, _all_ ATTN processing occurs before _any_ erase, kill, or escape processing. Furthermore, any or all erase, kill, or escape sequence characters may be removed using ATTN.

ESCAPE PROCESSING

An escape mechanism is provided in order to allow any arbitrary character or sequence of characters to be entered in spite of canonicalization, erase, kill, and ATTN. The escape character is the dieresis ¨ (represented as a double quote " on non-APL terminals).

The escape character is followed by: (1) another escape character, which represents exactly one dieresis as data in the input line, without further escape processing; or, (2) a one-, two-, or three-digit octal number, which represents a single APL character of precisely that internal code; or, (3) a carriage return, which represents exactly nothing -- the two characters are deleted from the input line; or, (4) an erase(or kill) character, resulting in neither character appearing in the input line.

If the character immediately following the escape character does not fit into one of the above rules, then it and the escape character _do not_ appear in the input line.

It is not possible to delete individual characters of an escape sequence: in order to erase a complete escape sequence, the last digit of the three digit octal sequence should be overstruck by the erase character or the ATTN key.

USING APL

When APL has been invoked and awaits input, the user may enter one of three types of input: an expression to be evaluated immediately, a system command, or an invocation of the function editor.

An expression to be evaluated immediately is the most common response. This entry initiates computations. This class of input is discussed in Section 3.

A system command interrogates or adjusts the environment in which computations are performed. Most system commands are attendant to bookkeeping functions: listing names of variables and functions, erasing variables, or leaving APL and returning to Multics (for APL-only users, logging out). System commands are discussed in Section 5.

Finally, the function editor is used to create, modify, or replace stored sequences of unevaluated APL lines -- known as functions -- for later execution. The function editor is discussed in Section 4.

SECTION 3

THE APL LANGUAGE

VALUES

In APL, a value is returned by evaluating any variable or expression. Values are therefore the fundamental entity of APL.

A value is a rectangular array of elements, each of which is a single character or a single number. A value can have any integral number of dimensions, from zero up, and the extent of each dimension can be any integer from zero up.

The number of elements in the value is equal to the product of its dimension extents. Character and numeric elements cannot be mixed within the same value.

Three important characteristics of every APL value are its type, its rank, and its shape.

Type

The type of a value is either character or numeric, depending upon whether its elements are characters or numbers.

APL further classifies numeric values into three subtypes: boolean, integer, and floating-point. These subtypes can be mixed within the same APL value; however, APL classifies the entire value according to the most general subtype present in the value.

For example, a value containing booleans and integers is classified as an integer value, since integers are a superset of -- and therefore are more general than -- booleans. Similarly, a value containing both integers and floating-point numbers is classified as a floating-point value.

Type and subtype distinctions are very important, since most APL operations have type and/or subtype restrictions on arguments.

A value with no elements at all (a so-called <u>empty</u> value) is acceptable to most APL operations. Usually, its <u>type</u> is ignored, since its lack of elements makes it compatible with both character and numeric operations.

<u>Rank</u>

The <u>rank</u> of a value is the number of dimensions it has. A <u>scalar</u> has rank <u>zero</u>, and consists of a single element. A <u>vector</u> has rank one, and consists of a set of linearly ordered <u>elements</u>.

A <u>matrix</u> has rank two, and consists of a set of elements arranged in two <u>orthogonal</u> (perpendicular) dimensions. It has well defined, linearly ordered rows and columns. All rows have the same number of elements; the same applies to columns.

An <u>array</u> has rank three or greater, and consists of a set of elements <u>arranged</u> in three or more orthogonal dimensions. It has well defined, linearly ordered rows, columns, planes, hyperplanes, and so forth. All rows have the same number of elements, the same applies to columns. All planes have the same dimension extents; the same applies to hyperplanes.

Rank is of great practical importance, since many APL operations have constraints on the ranks of their arguments.

<u>Shape</u>

The <u>shape</u> of a value is its set of dimension extents, expressed as an integer vector called the <u>shape</u> <u>vector</u>; it is itself an APL value. A scalar -- having no <u>dimensions</u> -- <u>has</u> an empty shape vector: a vector with no elements.

A vector has a one-element shape vector: an integer indicating the number of elements in the vector value. The shape of a vector is usually referred to as its <u>length</u>.

A matrix has a shape vector with exactly two elements. In general, the rank of any given value equals the length of its shape vector.

SHAPE OPERATOR

The APL shape operator, monadic ⍴, returns the shape vector of its argument value. It is the only practical means of finding the shape of a value, and therefore is a very important and heavily used operator.

If *A* is a 5 by 2 by 4 character array (three dimensional: 5 planes, 2 rows, and 4 columns), then ⍴*A* is an integer vector of three elements with value 5 2 4.

The shape of the shape is also an APL value: it is the rank of the original value. Thus, ⍴⍴*A* would be 3 in the current example. Similarly, for any APL value *B*, ⍴⍴⍴*B* is a one-element integer vector with value 1.

Elements

Each element of any APL value is equivalent to a scalar: an element cannot be 'null' or empty; an element cannot be equivalent to a vector, matrix, or array. Of course, a value can have no elements, in which case the value is empty. An empty value can have any shape or rank, except that at least one of its dimension extents must be zero.

OUTPUT OF VALUES

Two environment parameters directly affect the output of values. They are the page width and the number of digits of printing precision.

The page width is the maximum number of character positions per line that APL fills when typing output. It effectively acts as a right margin beyond which APL does not type. It can be set using the *)WIDTH* system command, orx) the ⎕*PW* system variable.

The number of digits of printing precision is the number of decimal digits that are printed when numeric values are displayed. Numbers are rounded to this precision before printing. The default is 10 decimal digits, but this can be changed using the *)DIGITS* system command, or the ⎕*PP* system variable.

This precision does not affect the accuracy with which internal calculations are carried out; it affects only the final printing of answers.

Scalars

A character scalar is output simply as the single character which it is; it is not placed within quotes or otherwise altered.

A numeric scalar is output in the simplest representation possible in the decimal notation. Positive signs are omitted, negative signs are printed as the overbar ¯. Magnitudes are displayed rounded to conform to the current digits setting.

SCIENTIFIC NOTATION

If its magnitude is very large or very small, APL may display the number in scientific notation; this consists of a digit string, the letter *E* (for "exponent"), and an integer which is the power of ten by which to multiply the digit string to obtain the true number being represented.

For example, Boltzmann's constant -- in joules per degree Kelvin -- is about 1.38 times 10^{23}. APL prints this as 1.38*E*¯23.

In general, the mantissa (1.38 here) will be between 1 and 9.9999... inclusive in magnitude. The exponent can range from -38 to 38 inclusive.

Small Numbers

For a number near zero, APL uses scientific notation if and only if the number is represented internally as being less than or equal to 0.0001 (or 1*E*¯4) in magnitude. This threshold is independent of the)*DIGITS* setting: 0.000097 is printed as 9.7*E*¯5 while in)*DIGITS* 2 or more, and as 1*E*¯4 in)*DIGITS* 1.

(Note: The number 1*E*¯3 is never printed as such; rather, it appears as 0.001. The printing of 1*E*¯3 indicates that the)*DIGITS* setting has caused a number that is less than 1*E*¯3 to be rounded up to it for printing.)

Large Numbers

For a large number, the)*DIGITS* setting defines the threshold that determines the printing format. If the)*DIGITS* setting is referred to as n, then in general, APL uses scientific notation for large numbers if and only if the number is represented internally as being greater in magnitude than 10 to the n power.

For example, in *)DIGITS* 3, 999.9999 is printed as 1000, whereas 1000.00001 is rounded down and printed as 1*E*3.

In general, the degree of precision for any value N is 10 to the -N power. Examples:

)DIGITS setting	numeric value	printed value
1	34.456	3E1
	72.67584	7E1
	19.1234E31	2E32
	22.23423E-10	2E-9
3	34.456	34.456
	72.67854	72.679
	19.1234E31	1.91E32
	22.23423E-10	2.22E-9
5	34.456	34.456
	72.67854	72.67854
	19.1234E31	1.9123E32
	22.23423E-10	2.2342E-10

Note that values printed in scientific notation are printed to the N - 1 degree of decimal precision (where N is the *)DIGITS* setting) while those values whose magnitude is less than or equal to N are printed to the Nth degree of decimal precision.

Vectors

For both character and numeric vectors, an empty vector prints as a single blank line.

CHARACTER

A character vector is output as a character string, with no added spaces or other separators intervening between its elements. Of course, the elements themselves can be special characters: space, backspace, carriage-return, newline, etc. In fact, every character in Table 2-1 is valid as an element of an APL character value.

If the character vector is longer than the page width, then as many elements as possible are printed on the first line; then, the excess elements overflow to the following line or lines, as necessary; each overflow line is indented six spaces.

NUMERIC

In the printing of a numeric vector, each element is set off from the preceding one by a single space. The printing format is determined for each element individually; for example, the following is a correct output while in *)DIGITS* 3: 237 4.6*E*¯17 0.198 3.89*E*3 ¯6.4.

As with character vectors, if one line is insufficient in width to accommodate all the elements, the excess elements are placed on a succeeding line or lines, each indented by six spaces. However, the character string that is the printed representation of a single numeric element is never split between lines.

<u>Matrices and Arrays</u>

Matrices and arrays are printed in a succession of rectangular planes, pairs of which are separated by one or more blank lines. As many planes as necessary are printed to output the entire value. Each row of a plane is printed starting on a new print line; within a plane, no blank lines appear.

To ensure that the output of each matrix or array begins on a fresh print line, APL always sends a conditional newline -- ¨301 -- immediately before starting to print any matrix or array.

As with vectors, if the page width is insufficient to hold the output of an entire row, then the excess elements from each row are printed on the immediately following line or lines, as necessary; these inserted overflow lines are each indented six spaces.

MATRICES

A matrix is printed in one plane, with no leading, imbedded, or trailing blank lines; assuming sufficient page width, it occupies exactly as many print lines as the matrix has rows.

ARRAYS

A 3-dimensional array is printed in as many planes as the array has. Each pair of planes is separated by exactly one blank line. For example, a 5 by 3 by 7 character array is printed in 5 planes, each of which has 3 lines and 7 columns; each plane -- except the first one -- is preceded by a blank line.

Blank lines are inserted only when printing values of rank greater than two -- arrays. They serve merely as a visual aid: they indicate the boundary between adjacent planes, hyperplanes, and so forth. (Since paper is inherently two-dimensional, matrices, vectors, and scalars can be represented without any intervening separators or boundary indicators.)

In order to understand the insertion of blank lines in the output of arrays, it is useful to visualize N-dimensional values as an ordered sequence of identical shape (N-1)-dimensional values. Each (N-1)-dimensional value is itself an ordered sequence of identical shape (N-2)-dimensional values, and so forth.

For example, a 3-dimensional array can be thought of as an ordered sequence of identical shape matrices -- planes of the 3-d array; each matrix is an ordered sequence of identical length vectors; each vector is an ordered sequence of scalars.

With this concept in mind, the general printing format is recursively defined for any N-dimensional value as follows:

> If $N \leq 2$, print the matrix, vector, or scalar as described above and below,
>
> otherwise,
>
> print the value as an ordered sequence of (N-1)-dimensional values, inserting N-2 blank lines between adjacent values.

For example, a 5-dimensional value *A*, such that 2 5 4 6 3 = ρ*A*, is printed as two 4-dimensional values, separated by three blank lines. Each of these 4-dimensional values is printed as five 3-dimensional values, with adjacent 3-dimensional values separated by two blank lines. Each 3-dimensional value is printed as four 6-by-3 matrices (planes), with adjacent matrices separated by one blank line.

CHARACTER

As with character vectors, the output of character matrices and character arrays contains no extra spaces or other separators intervening between the elements of each line of output.

Furthermore, APL keeps track of the cursor position -- the column position of the typing element -- even when printing imbedded carriage-motion characters -- space, backspace, tabulate, carriage-return, linefeed, or newline -- and when printing escape sequences in lieu of undefined characters. Therefore, output is consistent with the)*WIDTH* setting. (The only exception occurs when a tabulate character is output while the physical or electronic tab stops are set at intervals other than the current)*TABS* setting.)

NUMERIC

The output of numeric matrices and arrays is more complicated.

A consequence of this output format is that -- for many cases -- it is impossible to correctly determine the rank or shape of a value merely by examining its printed display. When the shape of a value must be known precisely, the shape operator ⍴ should be used to explicitly extract its shape.

Examples of the output of values follow the discussion of value input below.

INPUT OF VALUES

Character

A scalar or vector character value is input by typing the desired character(s) between a pair of quote characters. If it is desired to represent a quote character itself, the quote must be typed as two quotes. Thus, the input of a scalar character value representing a quote consists of four quotes: two to delimit the value, and two to represent the single quote being entered.

Newline, space, and any other APL character -- including those constructed from overstrikes or escape sequences -- can be entered between quotes. Note that the only carriage-motion characters that can be input directly are newline and space. Due to the "visual fidelity" rule, tabulate, carriage-return, backspace, and linefeed must be input via an appropriate escape sequence.

A character produced by an overstrike or escape sequence is considered a single element internally.

Numeric

A scalar or vector numeric value is input by typing the desired numeric elements(s), separated by one or more spaces or tabs. Numeric values do not require any explicit delimiting character, as do character values. Instead, the delimiting is implicit: anything that cannot be interpreted as a numeric element -- according to the prescribed rules -- is considered to delimit the numeric value.

The rules for forming a valid numeric element for input are as follows:

- Elements are delimited by "white space" -- one or more spaces and/or horizontal tabs -- and by anything which delimits a numeric value. Therefore, "white space" cannot be part of a numeric element.

- Only the following characters are valid:

  0 1 2 3 4 5 6 7 8 9 ¯ . *E*

- A digit string is defined as a contiguous string composed of the decimal digits:

  0 1 2 3 4 5 6 7 8 9

- The first character of the numeric element must not be the letter *E*.

- The element must be of the form, ¯ds1.ds2*E*¯ds3, where:

  - The first (leftmost) overbar ¯ is optional, and is punctuation -- not an operator -- indicating that the element is negative.

  - ds1, ds2, and ds3 are digit strings.

  - ds1 and ds2 can be of any length -- including zero.

  - ds1 and ds2 cannot both be of zero length.

  - The dot . is optional, and is punctuation -- not an operator -- indicating the position of the decimal point.

  - The construct *E*¯ds3 is optional, and is punctuation -- not an operator -- indicating the use of a decimal exponential multiplier. Within this construct: ds3 and *E* are required; ds3 must have either 1 or 2 digits; the overbar ¯ is optional, and is punctuation -- not an operator -- indicating that the decimal exponent is negative.

The way in which a number is typed does not matter; for example, all the following inputs result in the very same internal value:

```
7 007.00 .07E2 7000E¯3 7. 70.0E¯1 7E0.
```

Matrices and Arrays

A value of rank higher than one cannot be input directly. Such a value must be constructed by entering its elements as a vector, and then using the reshape operator -- dyadic ρ -- to reshape it to the desired dimensions, filling in the supplied elements in row-major order.

For example, the input 2 3ρ 1 2 3 4 5 6 is an expression whose value is a 2 by 3 matrix of integers from one to six; the first row is 1 2 3; the second row is 4 5 6.

OPERATORS

Every APL _operator_ has the following properties:

- It is graphically represented by exactly one printable, fixed -- not user-definable, nonalphanumeric symbol.
- Its action is fixed: users cannot "customize" APL operators.
- Exactly one of the following cases is true:
  - a) It takes exactly _one argument_ -- an APL value.
  - b) It takes exactly _two_ arguments.
- It returns exactly one _explicit result_ -- an APL value -- unless an error is detected.
- The result is _passed out_ of the operator and made available for use as an argument to the next operator, pseudo-operator, function, system function, or composite operation.
- No _implicit_ result is produced (e.g.: a change in the workspace environment).

- If an error is detected, all execution terminates, no result is produced, and all arguments and intermediate values are discarded.

An operator taking one argument is said to be monadic and is always written before -- to the left of -- its argument. An operator taking two arguments is said to be dyadic and is always written between its left and right arguments.

Scalar Operators

A scalar operator is one which is defined in terms of its action when given scalar arguments.

With certain broad restrictions, scalar operators can take nonscalar arguments. However, the scalar operation is still defined only in terms of its action on individual elements of its arguments: the elements are effectively taken one at a time, independently of each other.

In other words, a scalar operator applied to nonscalar arguments merely extends its action to each individual element of its arguments.

This is in contrast to a mixed operator, which effectively accepts entire values as its arguments, and which performs some action on whole values at once: an action in which the individual elements of its arguments cannot be considered independently of one another.

Examples of scalar operations include: addition, subtraction, logical AND, and logical OR (dyadic); also: absolute value, reciprocal, and factorial (monadic).

Examples of mixed operations include: reshaping and concatenation (dyadic); also: matrix transposition, matrix inversion, and sorting (monadic).

EXTENSION TO NON-SCALAR ARGUMENTS

Monadic

If a monadic scalar operator is applied to a nonscalar, the result is a value of identical rank and shape, and each element of the result is computed by applying the scalar operator to the corresponding element of the argument.

For example, if *A* is a numeric vector of six elements, then ÷*A* is a numeric vector of six elements, each element being the reciprocal of the corresponding element of the argument vector.

Dyadic

If a dyadic scalar operator is applied to two arguments of identical rank and shape, then the result is a value of the same rank and shape, and each element of the result is computed by applying the operator to the two corresponding elements of the two arguments.

For example, two identical length vectors can be added, element-by-element: 16 ¯2 13 7 29+1 14 ¯19 4 ¯6 gives the result 17 12 ¯6 11 23.

If a dyadic scalar operator is applied to two arguments that fail to match in rank and shape, but exactly one of the arguments consists of just one element, then that single element is considered to be replicated (extended) to the rank and shape of the other argument, and the operator proceeds element-by-element as above.

In other words, the single element participates with each element of the other argument in turn, producing a result identical in rank and shape to the other argument.

For example, if *A* is a 2 by 3 matrix of integers, then 6+*A* is a 2 by 3 matrix of integers, each of which is six greater than its corresponding element in *A*: the single element 6 is applied independently to each of the elements of *A*.

If both arguments consist of just one element -- any dimension extents must equal unity, then the rank of the result is arbitrarily taken as the larger of the two arguments' ranks.

For example, if *A* is a 1 by 1 numeric matrix (rank 2) and *B* is a one-element numeric vector (rank 1), then *A*+*B* is a 1 by 1 numeric matrix containing the appropriate sum.

If the two arguments do not match in rank and shape, and if neither argument consists of just one element, then the operation is in error: clearly, no general, unambiguous correspondence of the type described above can be established between the elements of the two arguments.

As a result, APL will suspend execution and issue a diagnostic message: *RANK ERROR* if the two arguments do not match in rank; *LENGTH ERROR* if they match in rank but some corresponding pair of dimension extents do not match.

General error reporting and possible recovery actions are discussed in detail later in this section.

GENERAL PROPERTIES OF SCALAR OPERATORS

In addition to the above outlined properties that every APL operator has, every scalar operator has the following properties:

- Exactly one of the following is true:

  a) Its arguments must be numeric.

  b) Its arguments can be of any type; the operator is either = or ≠, both of which are dyadic.

- It ignores the type of empty arguments.

- Its result is numeric.

- Its result has exactly the same rank and shape as at least one of its arguments; the rank and shape of the result is completely determined by that of its arguments and thus is independent of the individual data elements contained within its arguments.

- It is a function, in the mathematical sense -- as opposed to a relation: each element of its (possibly extended) arguments maps onto exactly one element of its result.

General Properties of Monadic Scalar Operators

In addition to the above outlined properties that every scalar operator has, every monadic scalar operator has the following properties:

- It takes exactly one argument.

- Its graphic symbol must be the first printable character to the left of its argument.

- Its argument can have any rank and shape.

- Its argument must be numeric.

- It detects only *DOMAIN ERRORS.*

- Its result has exactly the same rank and shape as its argument.

- Each element of its result is computed by applying the operator to the corresponding element of its argument.

General Properties of Dyadic Scalar Operators

In addition to the above outlined properties that every scalar operator has, every dyadic scalar operator has the following properties:

- It takes exactly two arguments.

- Its graphic symbol must be the only printable character between its two arguments.

- Its two arguments must be <u>scalar conformable</u>, which is defined as follows:

  o Exactly one of its two arguments can have any rank and shape.

  o Exactly one case from the following exhaustive set of mutually exclusive cases must be satisfied:

    1) Neither argument has exactly one element; the arguments must have identical rank and shape.

    2) Exactly one argument has exactly one element; that argument can have any rank, and the other argument can have any rank and shape.

    3) Each argument has exactly one element; each argument can have any rank.

- It detects only the following errors, in the order given: *RANK*, *LENGTH*, *DOMAIN*, and *COMPATIBILITY*.

- The rank and shape of its result are determined by which one of the above scalar conformability cases is satisfied:

  1) Its result has exactly the same rank and shape as its arguments.

  2) Its result has exactly the same rank and shape as the nonsingle-element argument.

3) Its result has exactly the same rank and shape as the argument with the higher rank.

- The individual elements of its result are computed using a method determined by which one of the above scalar conformability cases is satisfied:

  1) Its result is computed element-by-element by applying the operator in a scalar fashion to the pair of elements that occupy the corresponding position in the arguments.

  2) Its result is computed by first considering the single element to be replicated to the rank and shape of the nonsingle-element argument, and then using rule (1) above by substituting the now extended argument for the single-element argument.

  3) Its result is computed by treating the two arguments as scalars.

- Its result is scalar conformable to both of its arguments.

ADD, SUBTRACT, MULTIPLY, DIVIDE + - × ÷

When used dyadically, + - × and ÷ represent the arithmetic operations of addition, subtraction, multiplication, and division.

Unlike some programming languages that truncate quotients of integers to an integer, APL retains the fractional part of a quotient as accurately as the hardware permits -- approximately 19 decimal digits.

A *DOMAIN ERROR* occurs when dividing by zero, except for 0÷0, which is defined to equal 1. A *NONCE ERROR* occurs when the result of an operation exceeds the capacity of the hardware to represent numbers -- the largest magnitude representable is 1.7014118346046923173*E*38. Nonce errors are due to limitations of the implementation, not to some misuse of the APL language itself.

The divide operator uses integer fuzz to determine whether or not its left and/or right arguments are "effectively" -- that is, "fuzz equal to" -- zero.

PLUS, NEGATIVE + -

Monadic + leaves its numeric argument unchanged. A *DOMAIN ERROR* occurs if a character argument is given.

Monadic - represents negation; that is, algebraic change of sign of its argument.

SIGNUM ×

Monadic × represents the mathematical signum operation; that is, it returns a 1 if its argument is greater than zero, 0 if its argument is zero, and ¯1 if its argument is less than zero.

Signum uses integer fuzz (value of *⎕CT*) to determine whether or not its argument elements are "fuzz equal to" zero.

RECIPROCAL ÷

Monadic ÷ returns the reciprocal of its argument. A *DOMAIN ERROR* occurs if an element of its argument is within the <u>integer tolerance</u> of zero.

POWER, LOGARITHM * ⍟

is expressed in APL *B*N*. is expressed as *B⍟N*. Note that the base is the left argument for both operators.

If the base is omitted (monadic usage), then the base of the natural logarithm -- e ≅ 2.718281828459045... -- is used. Thus, *1 is e itself; **X* is the same as e^x; and *⍟X* is the same as ln x.

There is no square-root or cube-root operator in APL; the power operator is used to perform these operations. For example, the square root of A can be expressed as *A*0.5*.

Since APL does not handle complex numbers, any attempt to extract an even root of a negative number results in a *DOMAIN ERROR*.

The indeterminate case 0^0 defined to equal 1.

RESIDUE |

Dyadic | represents the modulo operation: *B|N* is read "*N* modulo *B*" or "the *B* residue of *N*." It is defined as the remainder left after *B* is divided the maximal integral number of times into *N*. (Note the order of the arguments: the left argument is the divisor and the right argument is the dividend.)

More precisely, if *N* is not 0, then an integer quotient *Q* is chosen so that the remainder *N*-(*Q*×*B*) is the smallest possible nonnegative remainder -- that is, greater than or equal to 0, but strictly less than the absolute value of *B*; this remainder is the value of *B*|*N*.

If *B* is 0, then *N* itself is the value of *B*|*N*.

MAGNITUDE |

Monadic | represents the absolute value of its right argument: the algebraic sign of each element is changed to positive if it was negative. Thus, the result is effectively the unsigned magnitude of its argument.

Monadic ! represents the factorial function. For positive, integer arguments !*N* is defined as the product of all positive integers up to *N*, where *N* is no greater than 33.82635 . For negative, integer arguments the factorial function is singular and results in a *DOMAIN ERROR*. For zero arguments, the result is defined to be 1. For non-integer arguments, !*A* represents the gamma function of *A*-1.

The gamma function is computed over the range 0 to 1, to an accuracy of 20 decimal places. Results for arguments outside this range are computed using a recurrence relation.

Note that the factorial function is written before its argument, not after it as in conventional mathematical notation. All APL monadic functions precede their argument.

Dyadic ! represents the binomial coefficients function. For non-negative, integer arguments *K*!*N* represents the number of ways that *K* different elements can be chosen from a collection of *N* objects. For negative, or non-negative arguments is defined in terms of the Gamma function, and generalized as before.

The cases of the binomial coefficents relation to binomial coefficients by the following identity:

```
BETA (A,B) ←→ ÷B×←A-1→!A+1
```

The mathematical notation for the binomial coefficients function is related to the APL notation as follows.

$$\left\{ \begin{matrix} K \\ N \end{matrix} \right\} \quad \leftrightarrow \quad K!N$$

Table 3-1. Binomial Coefficients Function Special Cases

A	B	B-A	*A!B*
0	0	0	`(!B)÷(!A)!B-A`
0	0	1	0
0	1	0	*DOMAIN ERROR*
0	1	1	`(¯1*A)=A!A-B+1`
1	0	0	0
1	0	1	*IMPOSSIBLE CASE*
1	1	0	`(¯1*B×A)=(|B+1)!|A+1`
1	1	1	0

In this table, a 1 indicates that the value of *A*, *B*, or *B-A* is a negative integer, and a 0 indicates that it is not.

The right-most column gives the expression that is used to compute the result. The result of *K!N* is also equivalent to the *K*th coefficient in the binomial expansion of (*X*+1)**N*.

MAXIMUM AND MINIMUM ⌈ ⌊

Dyadic ⌈ and ⌊ represent the maximum and minimum operations, respectively. They are defined only for numeric arguments: characters have no collating sequence in APL.

CEILING AND FLOOR ⌈ ⌊

Monadic ⌈ and ⌊ represent the ceiling and floor operations, respectively. Ceiling is defined as the algebraically smallest integer greater than or equal to its argument; floor is defined as the algebraically largest integer less than or equal to its argument.

A number is considered equal to an integer if it is within a certain tolerance of that integer. This tolerance is called integer fuzz. Fuzz is discussed later in this section.

ROLL *?*

While dyadic *?* is a mixed operator -- deal, monadic *?* is a scalar operator -- the random number generator -- named roll.

Its argument -- *A* -- must be a positive integer; its result is an integer chosen randomly and uniformly from the set of integers ⍳*A*.

(As explained later, the set ⍳*A* is a vector of *A* integers -- either 1 2 3 ... *A*, or else 0 1 2 ... *A*-1 -- depending upon whether the index origin is set to 1 or 0, respectively. (The index origin can be changed with the)*ORIGIN* system command or the ⎕*IO* system variable.))

Generating Algorithm

The random number algorithm used by Multics APL is a multiplicative congruential generator with period 34359738368. In this algorithm, the seed used to produce each random number is a function of the seed used to produce the precious one. In a clear workspace, the starting seed is derived from the calendar clock, so that the sequences of random numbers generated are unpredictable from session to session.

If it is desirable to work with a reproducible sequence of random numbers, the user should explicitly initialize the seed with the ⎕*RL* system variable. The seed can be set to any integral value from 1 to 34359738367. The seed is properly remembered and restored by the)*SAVE* and)*LOAD* system commands.

COMPARISON OPERATORS < ≤ = ≠ ≥ >

The APL comparison operators are < ≤ = ≠ ≥ and >. They represent the mathematical relations of less-than, less-than-or-equal-to, equal-to, not-equal-to, greater-than-or-equal-to, and greater-than, respectively. The comparison operators are all dyadic, and they all return the boolean value 1 to signify "true", or the boolean value 0 to signify "false".

Arguments of < ≤ ≥ and > must be numeric -- otherwise, a *DOMAIN ERROR* occurs.

Arguments of = and ≠ can be numeric or character or both. A number is considered not equal to a character; hence, in a mixed-type comparison, = always returns 0 and ≠ always returns 1.

Two numbers are considered equal if they are within a certain tolerance of each other. This tolerance is called fuzz. Fuzz is discussed later in this section.

LOGICAL OPERATORS ~ ∧ ∨ ⍲ ⍱

~ ∧ ∨ ⍲ and ⍱ represent the logical operations NOT, AND, OR, NAND, and NOR respectively. The NOT operator ~ is monadic; the other four are dyadic. Both the arguments and the results of the logical operators are restricted to the two boolean values 1 and 0, which signify "true" and "false" respectively.

~*A* is 1 if and only if *A* is 0.

A∧*B* is 1 if and only if both *A* and *B* are 1.

A∨*B* is 0 if and only if both *A* and *B* are 0.

A⍲*B* is 0 if and only if both *A* and *B* are 1.

A⍱*B* is 1 if and only if both *A* and *B* are 0.

By virtue of their actions on arguments of 0 and 1, the six comparison operators introduced above can also be used as dyadic logical operators, with = representing EQUIVALENCE, ≠ EXCLUSIVE OR, ≤ IMPLIES, and ≥ IS IMPLIED BY. This gives APL the complete set of all ten nontrivial dyadic logical operations.

A=*B* is 1 if and only if *A* and *B* are both 0 or both 1.

A≤*B* is 1 unless *A* is 1 and *B* is 0.

A>*B* is 0 unless *A* is 1 and *B* is 0.

A≥*B* is 1 unless *A* is 0 and *B* is 1.

A<*B* is 0 unless *A* is 0 and *B* is 1.

CIRCULAR ○

Dyadic circle ○ is used to generate the common trigonometric and hyperbolic functions of its right argument. The left argument determines which function is generated.

Angular values are expressed in radians.

¯7○*A*	←→	arctanh *A*
¯6○*A*	←→	arccosh *A*
¯5○*A*	←→	arcsinh *A*
¯4○*A*	←→	(¯1+*A*×*A*)*0.5
¯3○*A*	←→	arctan *A*
¯2○*A*	←→	arccos *A*
¯1○*A*	←→	arcsin *A*
0○*A*	←→	(1-*A*×*A*)*0.5
1○*A*	←→	sin *A*
2○*A*	←→	cos *A*
3○*A*	←→	tan *A*
4○*A*	←→	(1+*A*×*A*)*0.5
5○*A*	←→	sinh *A*
6○*A*	←→	cosh *A*
7○*A*	←→	tanh *A*

Any other left argument of ○ is a *DOMAIN ERROR*.

PI TIMES ○

Monadic circle ○ multiplies its argument by (an approximation of) the transcendental number pi. For example, ○*A* is *A*×3.14159265358979... The value of pi used by APL is accurate to 1 part in 2^{63}

<u>Mixed</u> <u>Operators</u>

A <u>mixed</u> <u>operator</u> is one that must consider an argument as a whole, rather than acting independently on its constituent elements. Each mixed operator has its own rules about the rank and shape of arguments it accepts. Like the scalar operators, some mixed operator symbols can be used either monadically or dyadically, with some change in meaning of the operation performed.

A few of the operator descriptions in this section make use of subscript notation before it is formally introduced. That is, *V*[*I*] is used to refer to the *I*th element of the vector *V*, and *M*[*I*;*J*] is used to refer to the *I*,*J*th element of the matrix *M*. The subscripting capability of APL is actually far more powerful than these simple uses suggest, and is discussed at length under Lists later in this section.

SHAPE ρ

Monadic rho ρ is an operator whose return value is the dimension vector, or shape of its argument. The type and element values of the argument are ignored. The result of the shape operator is always an integer vector.

The shape of a scalar (which has no dimensions) is an empty vector (a vector with length 0; i.e., containing no elements). The shape of a vector is a vector of length 1 (because the argument has one dimension) whose element indicates the length of the argument. The shape of a matrix (2 dimensions) is a vector of length 2, whose elements are the extents of the two dimensions of the argument matrix. In general, the shape of any value is a vector of length equal to the rank, or number of dimensions, of the value.

The shape operator can be applied to its own result to produce the shape of the shape. Since the length of the shape is the rank of the original argument, this is a way of obtaining the rank of any value.

RESHAPE ρ

Dyadic rho ρ is the reshape operator. It forms a sequence of elements into a specified shape. The left argument of reshape must be a shape vector (a vector of nonnegative integers). The elements of the right argument are used to fill up a value of the shape specified by the left argument. The shape of the right argument is ignored.

If the result requires fewer elements than the right argument provides, the excess elements are simply not used. If the result requires more elements than the right argument provides, then the elements of the right argument are repeated over and over, as many times as are necessary to fill up the result.

The elements are extracted and packed in row-major order. That is, the first elements treated are those of the first row of the first plane, followed by the second row of the first plane, and so on, through the last row of the last plane.

If the result is to have any elements at all, then the right argument of reshape must have at least one element; otherwise, a *LENGTH ERROR* occurs.

RAVEL ,

Monadic comma , is the ravel operator. Ravel returns its argument as a vector, by retaining all of its elements but ignoring its rank and shape.

CATENATE , ⍪

The operation implied by the form *A*,[*I*]*B* is dependent upon the subtype of the value of *I*: if *I* is an integer, then the operator is catenate; if *I* is a noninteger, it is laminate.

Catenate joins two APL values along an existing coordinate: its two arguments are combined to form its result by placing them "next to" each other, along a coordinate specified by the origin-dependent coordinate index *I* -- if given, or along the last dimension of the higher-ranked argument -- if the form *A*,*B* is used.

The conformability requirements of catenate are quite complex, yet are as unrestrictive as is reasonably possible, and are in practice more easily mastered than those of some other, seemingly simpler mixed operators.

Since catenate is the only APL operator that performs the vital operation of joining values along an existing coordinate, it is a heavily used operator in almost any application, and its conformability requirements must therefore be thoroughly mastered.

Although these conformability requirements are discussed in general later in this description, it is also useful -- for greater clarity -- to present them in a more detailed, case-by-case framework, as follows.

In the following discussions, it is assumed that the coordinate index for the operation is available in the variable *I*. If the form *A*,[*I*]*B* is used, *I* is explicitly stated. If the form *A*,*B* is used, then *I* is taken to be (⍴⍴*A*)⌈⍴⍴*B* in)*ORIGIN* 1 (¯1+(⍴⍴*A*)⌈⍴⍴*B* in)*ORIGIN* 0). If the form *A*⍪ is used, *I* is taken to be 1.

Case 1: Two multielement arguments.

If both *A* and *B* are multielement values, then the conformability requirements are:

DOMAIN	*A* and *B* must be of the same type;
INDEX	*I* -- if specified explicitly -- must be in the set ⍳(⍴⍴*A*)⌈⍴⍴*B*;
LENGTH	If *A* and *B* have the same rank, then their shapes must match, except that their *I*th dimension extents need not be equal;
LENGTH	If *A* and *B* have ranks that differ by exactly one, then their shapes must match, except that the *I*th dimension extent of the higher-rank argument is disregarded (for the purposes of this comparison);
RANK	If *A* and *B* have ranks that differ by more than 1, then a *RANK ERROR* is reported.

The rank of the result equals that of the higher-rank argument.

If *A* and *B* have the same rank, then the shape of the result matches that of each argument, except that its *I*th dimension extent is the sum of the *I*th dimension extents of the arguments.

If *A* and *B* have ranks that differ by 1, then the shape of the result matches that of the higher-rank argument, except that its *I*th dimension extent is 1 greater than that of the higher-rank argument.

Case 2: One multielement argument; one single-element argument.

If one argument is a single-element value, while the other argument is a multielement value, then the conformability requirements are:

DOMAIN	*A* and *B* must be of the same type;
INDEX	*I* -- if specified explicitly -- must be in the set ⍳⍴⍴*MULTI_ELEMENT_ARG*;

The rank of the result equals that of the multielement argument.

The shape of the result matches that of the multielement argument, except that its *I*th dimension extent is 1 greater than that of the multielement argument.

Case 3: Two single-element arguments.

If both *A* and *B* are single-element values, then the conformability requirements are:

DOMAIN	*A* and *B* must be of the same type;
INDEX	*I* -- if specified explicitly -- must be in the set ⍳(⍴⍴*A*)⌈⍴⍴*B*.

The rank of the result equals that of the higher-rank argument.

The shape of the result is such that all dimension extents equal 1, except the *I*th, which equals 2.

Case 4: One single-element argument; one empty argument.

If one argument is a single-element value, while the other is an empty value, then the conformability requirements are:

INDEX	*I* -- is specified explicitly -- must be in the set ⍳⍴⍴*EMPTY_ARGUMENT*.

The rank of the result equals that of the empty argument.

The shape of the result equals that of the empty argument, except that the *I*th dimension extent of the result is one greater than that of the empty argument.

The type of the result depends upon the number of elements in the result, and may depend upon the argument order:

if the result is nonempty, then the type of the result matches that of the single-element argument;

if the result is empty, then the type of the result matches that of the right argument.

Case 5: One multielement argument, one empty argument.

If one argument is a multielement value, while the other is an empty value, then the conformability requirements are:

INDEX	*I* -- if specified explicitly -- must be in the set: ⍳(⍴⍴*A*)⌈⍴⍴*B*;
LENGTH	If *A* and *B* have the same rank, then their shapes must match, except that their *Ith* dimension extents need not -- and will not -- be equal; or the empty argument must be a zero-extent value.
LENGTH	If the multielement argument has rank exactly one greater than that of the empty argument, then the empty argument must be a zero-extent value;
LENGTH	If the empty argument has rank exactly one greater than that of the multielement argument, then their shapes must match, except that the *Ith* dimension extent of the empty argument is disregarded (skipped over); or, the empty argument must be a zero-extent value.
RANK	If *A* and *B* have ranks that differ by more than one, then a *RANK ERROR* is reported.

The rank of the result equals (⍴⍴*A*)⌈⍴⍴*B*.

If (⍴⍴*A*)=⍴⍴*B*, then the shape of the result equals that of the multielement argument.

If, of the two arguments, the empty argument has the higher rank, then the shape of the result is: ((*I*-1)↑⍴*MULTI_ELEMENT_ARG*),1,(*I*-1)↓⍴*MULTI_ELEMENT_ARG*.

If, of the two arguments, the multielement argument has the higher rank, then the shape of the result equals that of the multielement argument, except that the *Ith* dimension extent of the result is 1 greater than that of the multielement argument. (This has the effect of "creating data:" the result has more elements than do the two arguments. The extra elements reflect the type of the result: they are zeros if numeric, or spaces if character.)

Case 6: Two empty arguments.

If *A* and *B* are both empty values, then the conformability requirements are:

INDEX	*I* -- if specified explicitly -- must be in the set: ⍳(⍴⍴*A*)⌈⍴⍴*B*
LENGTH	If *A* and *B* have the same rank, then their shapes must match, except that their *Ith* dimension extents need not be equal; or, at least one argument must be a zero-extent value;
LENGTH	If *A* and *B* have ranks that differ by exactly one, then their shapes must match, except that the *Ith* dimension extent of the higher-rank argument is disregarded; or, at least one argument must be a zero-extent value.
RANK	If *A* and *B* have ranks that differ by more than one, then a *RANK ERROR* is reported.

The type of the result matches that of the right argument.

The rank of the result equals (⍴⍴*A*)⌈⍴⍴*B*.

The shape of the result depends upon the type of conformability achieved:

If *A* and *B* have the same rank, then

If the shapes of *A* and *B* -- excluding their *Ith* dimension extents -- match, then the shape of the result matches that of each argument, except that the *Ith* dimension extent of the result is the sum of the *Ith* dimension extents of the arguments.

If at least one of the arguments is a zero-extent value, then the shape of the result matches that of the other argument -- which may or may not also be a zero-extent value.

If *A* and *B* have ranks that differ by exactly one, then

If the shapes of *A* and *B* -- excluding the *Ith* dimension extent of the higher-rank argument -- match, then the shape of the result matches that of the higher-rank argument, except that the *Ith* dimension extent of the result is one greater than that of the higher-rank argument.

If at least one of the arguments is a zero-extent value, then:

if the higher-rank argument is a zero-extent value, then the shape of the result matches that of the lower-rank argument, except that a dimension extent of 1 is inserted into the *Ith* position:

(ρ*RESULT*)=((*I*-1)↑ρ*LOWER_RANK_ARG*),1,(*I*-1)↓ρ*LOWER_RANK_ARG*).

if the higher-rank argument is not a zero-extent value (but the other argument is), then the shape of the result matches that of the higher-rank argument, except that the *Ith* dimension extent of the result is one greater than that of the higher-rank argument. (Note that this may cause the result to be nonempty.)

General Rules for Catenate

TYPE:

Conformability: If both arguments are nonempty, then they must be of the same type.

If at least one argument is empty, then type is ignored in determining conformability.

Result: If both arguments are empty, or if the result is empty, then the type of the result matches the type of the right argument.

If at least one argument is nonempty, and if the result is nonempty, then the type of the result matches the type of the nonempty argument.

RANK:

Conformability: If neither argument is a single-element value, then their ranks may differ by either 0 or 1.

If at least one argument is a single-element value, then rank is not considered in conformability.

Result:

If neither argument is a single-element value, or if both arguments are single-element values, then the rank of the result matches that of the higher-rank argument. (ρρ*R*)=(ρρ*A*)⌈(ρρ*B*)

If exactly one argument is a single-element value, then the rank of the result matches that of the nonsingle-element argument.

SHAPE:

Conformability:

After meeting type and rank requirements -- if any, two arguments are catenate conformable:

> if at least one argument is a single-element value; or
>
> if at least one argument is a zero-extent value; or
>
> if their ranks match, then if their shapes -- excluding their *I*th dimension extents -- match; or
>
> if their ranks differ by 1, then if their shapes -- excluding the *I*th dimension extent of the higher-rank argument -- match.

Result:

If the argument ranks match, then the *I*th dimension extent of the result equals the sum of the *I*th dimension extents of the arguments.

If the argument ranks differ by 1, or if at least one argument is a one-element value, then the *I*th dimension extent of the result is exactly one greater than that of the higher-rank argument.

The other (non-*I*th) dimension extents of the result "match" those of at least one of the arguments:

> if the argument ranks match, then the non-*I*th dimension extents of the result match the non-*I*th dimension of at least one of the arguments;

if the argument ranks differ by 1, then the non-*I*th dimension extents of the result match the non-*I*th dimension extents of the higher-rank argument, and/or match all dimension extents of the lower-rank argument;

if the argument ranks differ by more than 1, then at least one argument is a single-element value, and therefore:

> if exactly one argument is a single-element value, then the non-*I*th dimension extents of the result match the non-*I*th dimension extents of the nonsingle-elements argument;
>
> if both arguments are single-element values, then the non-*I*th dimension extents of the result all equal 1, and therefore match the non-*I*th dimension extents of the higher-rank argument.

The following rules define which argument(s) determine, supply, and therefore have dimension extents that "match" the non-*I*th dimension extents of the result:

if at least one argument is a multielement value, then each multielement argument does;

if exactly one argument is a single-element value, then the other -- nonsingle-element -- argument does;

if both arguments are single-element values, then the higher-rank argument does,

if both arguments are empty values, then:

> if at least one argument is a nonzero-extent value, then each nonzero-extent argument does,
>
> if both arguments are zero-extent values, then both arguments do.

(These rules define a hierarchy of precedence as follows:

multielement

$\text{empty} \left\{ \begin{array}{l} \text{nonzero-extent values} \\ \text{zero-extent value} \end{array} \right.$

single element

<u>Notes:</u>

With one exception, catenate does not discard data -- the result usually has all of the elements contained in both arguments.

The exception: one single-element argument, one empty argument; if (ρ*EMPTY_ARG*)[*I*]≠0 or if the empty argument has more than 1 dimension extent of zero (that is, (+/0ρ*EMPTY_ARG*)>1), then the result is empty. However, if (ρ*EMPTY_ARG*)[*I*]=0 and if 1=+/0=ρ*EMPTY_ARG*, then the result is not empty, and is composed solely of the element supplied by the single-element argument.

With one exception, catenate does not create data -- the result usually has no elements that are not found in the arguments.

The exception: argument rank difference of 1, lower-rank argument is a zero-extent value; the <u>*I*th</u> dimension extent of the result is 1 greater than that of the higher-rank argument (All other dimension extents match those of higher-rank argument.); if this change causes the result to have more elements than has the higher-rank argument, then additional elements are needed. (In general, this is true unless the higher-rank argument is empty and has a zero dimension extent that is <u>not</u> its <u>*I*th</u> one.) The additional elements are zeros if the type of the result is numeric, or spaces if character.

LAMINATE *A*,[*I*]*B*

If *I* is a noninteger, numeric, one-element value, then *A*,[*I*]*B* represents <u>laminate</u>.

<u>Laminate</u> joins two APL values along a new coordinate: its two arguments are combined to form its result by placing them parallel to each other, creating a new coordinate whose dimension extent is 2.

The placement of the new coordinate -- relative to the original coordinates -- is specified by the origin-dependent coordinate index *I* -- if given. (If *I* is not given, then the operator is laminate if and only if both arguments are scalars, in which case *I* is taken to equal .5; otherwise, the operator is catenate (above).)

However, since *I* is not an integer, it does not refer to an existing coordinate. Rather, it specifies the coordinate(s) next to which -- or between which -- the new coordinate will be placed.

For example (origin 1), for *A*,[1.5]*B*, the new coordinate is the 2nd coordinate of the result: ⌈*I*. If *A* and *B* are equal length vectors, then the result is a ρ*A* by 2 matrix. If *A* and *B* are identically shaped matrices, then the new coordinate is placed between the 1st and 2nd (⌊*I* and ⌈*I*) coordinates of the arguments; the result is a (ρ*A*)[1] by 2 by (ρ*A*)[2] array.

The conformability requirements of laminate are much less complex than those of catenate, but are still about as general and unrestrictive as is reasonably possible.

Like catenate, laminate is the only APL operator that performs the vital operation of joining values along a new coordinate. Thus, it is a commonly used operator whose conformability requirements should be learned thoroughly.

<u>Case 1</u>: Two multielement arguments.

If both arguments are multielement values, then the conformability requirements are:

DOMAIN	*A* and *B* must have the same type;
RANK	*A* and *B* must have the same rank;
INDEX	*I* must be less than 1+ρρ*A* (*ORIGIN* 1) or ρρ*A* (*ORIGIN* 0);
INDEX	*I* must be greater than 0 (*ORIGIN* 1) or ¯1 (*ORIGIN* 0);
LENGTH	*A* and *B* must have the same shape.

The rank of the result is 1 greater than that of each argument.

The shape of the result matches that of each argument, except that a dimension extent of 2 is inserted, becoming the ⌈*I*<u>th</u> dimension extent of the result.

<u>Case 2</u>: One multielement argument; one single-element argument.

If one argument is a single-element value, while the other argument is a multielement value, then the conformability requirements are:

DOMAIN	*A* and *B* must have the same type;
INDEX	*I* must be less than 1+ρρ*MULTI_ELEMENT_ARG* (*ORIGIN* 1) or ρρ*MULTI_ELEMENT_ARG* (*ORIGIN* 0);
INDEX	*I* -- if specified explicitly -- must be greater than 0 (*ORIGIN* 1) or ¯1 (*ORIGIN* 0).

The rank of the result is 1 greater than that of the multielement argument.

The shape of the result matches that of the multielement argument, except that a dimension extent of 2 is inserted, becoming the ⌈*I*<u>th</u> dimension extent of the result.

<u>Case 3</u>: Two single-element arguments.

If both arguments are single-element values, then the conformability requirements are:

DOMAIN	*A* and *B* must have the same type;
INDEX	*I* -- if specified explicitly -- must be less than 1+ρρ*HIGHER_RANK_ARG* (*ORIGIN* 1) or ρρ*HIGHER_RANK_ARG* (*ORIGIN* 0);
INDEX	*I* -- if specified explicitly -- must be greater than 0 (*ORIGIN* 1) or ¯1 (*ORIGIN* 0).

The rank of the result is 1 greater than that of the higher-rank argument.

The shape of the result is such that all dimension extents equal 1, except the ⌈*I*th, which equals 2.

Case 4: One single-element argument; one empty argument.

If one argument is a single-element value, while the other argument is an empty value, then the conformability requirements are:

INDEX	*I* must be less than 1+ρρ*EMPTY_ARG* (*ORIGIN* 1) or ρρ*EMPTY_ARG* (*ORIGIN* 0);
INDEX	*I* must be greater than 0 (*ORIGIN* 1) or ¯1 (*ORIGIN* 0).

The type of the result matches that of the right argument.

The result is empty.

The rank of the result is 1 greater than that of the empty argument.

The shape of the result matches that of the empty argument, except that a dimension extent of 2 is inserted, becoming the ⌈*I*th dimension extent of the result.

Case 5: One multielement argument; one empty argument.

RANK	*A* and *B* must have the same rank;
INDEX	*I* must be less than 1+ρρ*A* (*ORIGIN* 1) or ρρ*A* (*ORIGIN* 0);
INDEX	*I* must be greater than 0 (*ORIGIN* 1) or ¯1 (*ORIGIN* 0);
LENGTH	The empty argument must be a zero-extent value.

The rank of the result is 1 greater than that of the multielement argument.

The shape of the result matches that of the multielement argument, except that a dimension extent of 2 is inserted, becoming the ⌈*I*th dimension extent of the result.

Case 6: Two empty arguments.

If both arguments are empty values, then the conformability requirements are:

RANK	*A* and *B* must have the same rank;
INDEX	*I* must be less than 1+ρρ*A* (*ORIGIN* 1) or ρρ*A* (*ORIGIN* 0);
INDEX	*I* must be greater than 0 (*ORIGIN* 1) or ¯1 (*ORIGIN* 0);
LENGTH	*A* and *B* must have the same shape, or at least one must be a zero-extent value.

The rank of the result is 1 greater than that of each argument.

The type of the result matches that of the right argument.

The result is empty.

If at least one argument is a nonzero-extent value, then the shape of the result matches that of each nonzero-extent argument, except that a dimension extent of 2 is inserted, becoming the ⌈*I*th dimension extent of the result.

If both arguments are zero-extent values, then the shape of the result matches that of each argument, except that a dimension extent of 2 is inserted, becoming the ⌈*I*th dimension extent of the result.

General Rules For Laminate

TYPE:

Conformability	If both arguments are nonempty, then they must have the same type.
	If at least one argument is empty, then type is ignored in determining conformability.
Result:	If both arguments are empty, or if the result is empty, then the type of the result matches that of the right argument.

If at least one argument is nonempty, and if the result is nonempty, then the type of the result matches that of the nonempty argument.

RANK:

Conformability: If neither argument is a single-element value, then their ranks must be equal.

If at least one argument is a single-element value, then rank is not considered in determining conformability.

Result: If neither argument is a single-element value, then the rank of the result is 1 greater than that of each argument.

If both arguments are single-element values, then the rank of the result is 1 greater than that of the higher-rank argument.

If exactly one argument is a single-element value, then the rank of the result is 1 greater than that of the nonsingle-element argument.

SHAPE:

Conformability: After meeting type and rank requirements, -- if any, two arguments are laminate conformable:

> if at least one argument is a single-element value; or
>
> if at least one argument is a zero-extent value; or
>
> if their ranks match, then if their shapes match.

Result: The ⌈*I*th dimension extent of the result equals 2.

The other (non-⌈*I*th dimension extents of the result match those of at least one of the arguments.

> if at least one argument is a multielement value, then each multielement argument;
>
> if exactly one argument is a single-element value, then the other -- nonsingle-element -- argument;

if both arguments are single-element values, then the higher-rank argument;

if both arguments are empty values, then:

> if at least one argument is a nonzero-extent value, then each nonzero-extent argument;
>
> if both arguments are zero-extent values, then both arguments.

(These rules define a hierarchy of precedence:)

$$\begin{array}{r} \text{multielement} \\ \text{empty} \\ \text{single element} \end{array} \left\{ \begin{array}{l} \text{nonzero-extent} \\ \\ \text{zero-extent} \end{array} \right.$$

Notes:

With one exception, laminate does not discard data -- the result usually has all of the elements contained in both arguments.

> The exception: one single-element argument, one empty argument; the result will be empty, and therefore does not contain the element from the single-element argument.

With one exception, laminate does not create data -- the result usually has no elements that are not found in the arguments.

> The exception: one multielement argument, one empty (zero-extent) argument; the result has exactly twice the number of elements that the two arguments have, and therefore additional, arbitrarily chosen elements are needed; these elements are consistent with the result type: zeros if numeric, spaces if character.

INDEX GENERATOR ⍳

Monadic iota ⍳ is the index generator operator. Its argument must be a one-element nonnegative integer value. The result of ⍳*A* is a vector of integers of length *A*, the first element of which is the index origin (either 0 or 1), and succeeding elements of which are each one greater than the preceding element.

The index origin can be changed with the)*ORIGIN* system command or the ⎕*IO* system variable.

INDEX OF ⍳

Dyadic iota *A*⍳*B* represents the index of the first occurrence of *B* in the vector *A*. The left argument of index of must be a vector, or a *RANK ERROR* occurs. The right argument can be a value of any shape; its elements are considered independently of one another. The shape of the result is the same as that of the right argument.

As each element is selected from *B*, it is compared to the successive elements of *A*, starting with the first and proceeding until a match is found. If a match is found, then the answer is the index of the element that matched. If no match is found, then an index one greater than the last (highest) index of *A* is returned.

The indices returned by index of follow the index origin. If the index origin is 0, then the first element of *A* has the index 0, then next is indexed 1, and so on. If the index origin is 1, then the first element of *A* is indexed 1, the next 2, and so on.

Two numeric elements are considered equal if they are within a certain tolerance of each other. This tolerance is called the <u>comparison tolerance</u>, and is discussed later in this section.

If more than one element of *A* matches the element of *B* being considered, the index of the earliest is returned.

If no element of *A* matches, then an index one greater than the last index of *A* is returned; e.g., if *A* has seven elements, then 7 is returned in origin 0 (because the elements of *A* are numbered from 0 to 6), but 8 is returned in origin 1 (because the elements are then numbered from 1 to 7).

TAKE, DROP ↑ ↓

Take ↑ and drop ↓ are both dyadic operators that accept a vector of integers *V* as left argument and any value *A* as right argument. ⍴*V* must equal ⍴⍴*A* (except that a scalar *V* is automatically replicated to the rank of *A*).

The result of take *V*↑*A* is to take, for each dimension *I* in the rank of *A*, the first (if *V*[*I*]>0), or the last (if *V*[*I*]<0) |*V*[*I*] elements of that dimension, discarding the other elements.

The result of drop *V*↓*A* is to drop or discard the first (if *V*[*I*]>0) or last (if *V*[*I*]<0) |*V*[*I*] element of each coordinate *I*, retaining the others.

For both take and drop, |*V*[*I*] need not be less than (⍴*A*)[*I*].

For take, if (|*V*[*I*])>(⍴*A*)[*I*] for some *I*, then either zeros or spaces -- depending upon the type of the right argument -- are used to fill out the result to the required dimensions.

For drop, if (|*V*[*I*])≥(⍴*A*)[*I*] for some *I*, then all elements along the *I*th coordinate are dropped. (It is meaningless to speak of dropping more elements than actually exist.) That is, what is actually performed is: ((×*V*)×(|*V*)⌊⍴*A*)↓*A*. This makes the extent of the *I*th dimension of the result equal zero, and therefore the result is empty.

For take, the result always has shape (⍴⍴*A*)⍴|*V*, which, for non-scalar *V* (0≠⍴⍴*V*), is simply |*V*.

For drop, the result always has shape (⍴*A*)-(|*V*)⌊⍴*A*, which, for (*V*≤⍴*A*)∧*V*≥0 (that is, nonnegative and within the dimension extents of *A*), is simply (⍴*A*)-*V*.

GRADE UP, GRADE DOWN ⍋ ⍒

Grade up ⍋ and grade down ⍒ are the APL sorting operators. They are both monadic, and accept any numeric array as argument (characters have no collating sequence in APL -- hence they cannot be sorted.).

The result of ⍋ or ⍒ is a permutation array (a value whose elements are indices), identical in shape to *A*, that orders the elements of *A* to be monotonically nondecreasing or nonincreasing along the last dimension of *A*. This is, when the result of the grade operator is used to subscript its argument, the result is found to be sorted along the last dimension.

The sort preserves the original order of equal elements. Neither grade up nor grade down uses the comparison tolerance when comparing elements.

If *A* is a vector, then *A*[⍋*A*] is the elements of *A* sorted into increasing order. If *A* is a matrix, then ⍋*A* is a permutation matrix each row of which orders each row of *A* into ascending order, so *A*[*I*;(⍋*A*)[*I*;]] is the Ith row of *A* sorted.

If a one-element nonnegative integer value in brackets follows the grade operator, as ⍋[*I*]*A*, then the value *I* is taken as the coordinate index upon which to sort (instead of the last coordinate). The coordinate index and the indices returned by the grade operators follow the index origin. Thus, in 0-origin indexing, if *A* is a matrix, then ⍋[1]*A* is the same as ⍋*A*, while ⍋[0]*A* is the permutation matrix which orders the columns of *A* into increasing order.

REVERSE ⌽ ⊖

Monadic ⌽ reverses the elements along the last dimension of its argument, while monadic ⊖ reverses along the first dimension of its argument. Like the grade operators, reverse accepts a bracketed coordinate index; ⌽[*I*]*A* and ⊖[*I*]*A* reverses the elements of *A* along the Ith dimension; where *I* is a one-element nonnegative integer value. The coordinate index follows the setting of the index origin.

ROTATE ⌽ ⊖

Dyadic ⌽ and ⊖ represent the rotate operators: ⌽ rotates the elements of its right argument along the last dimension while ⊖ rotates along the first dimension. ⌽[*I*] and ⊖[*I*] rotates along the Ith dimension (coordinate index follows the index origin).

The left argument of ⌽ and ⊖ specifies the amount of rotation as follows: in *A*⌽[*I*]*B* or *A*⊖[*I*]*B*, *A* must be an integer value with rank one lower than that of *B*, or a 1-element integer value of any rank; each integer specifies the number of positions to the "left" that the elements of each corresponding "vector" of *B* along the Ith dimension are to be rotated.

Elements rotated off the end of a value re-enter it on the other end.

Zero is a valid rotation (which results in no change), as are negative numbers (which result in rotation to the "right"), as well as very large numbers (which may have the effect of rotating the "vector" through its starting position several times -- the interpreter avoids performing the superfluous complete cycles).

If a one-element integer value is given for *A*, then it is replicated to the required shape; i.e., all "vectors" of *B* along the Ith coordinate are rotated by the same scalar amount.

"Left" and "right" here are used figuratively: they refer to the direction of rotation when rotating along the last coordinate. For example, if B is a matrix, then 1⌽B causes all rows of B to be rotated to the left 1 position, whereas 1⊖B causes all columns of B to be rotated up 1 position. Thus, positive rotation is always in the direction of lower indices for that coordinate; that is, positive rotation moves elements towards the first elements of that dimension (towards an axis).

Similarly, "vector" is used loosely above to refer merely to a set of linearly ordered elements, not to a complete and separate value.

Conformability requirements are as follows: 1=X/⍴A or

A) For A⌽B: (⍴A)=¯1↓⍴B;

B) For A⊖B: (⍴A)=1↓⍴B;

C) For A⌽[I]B or A⊖[I]B: (⍴A)=((I-1)↑⍴A),I↓⍴B assuming that I∊⍳⍴⍴B.

That is for any B, A can be a single-element value; otherwise, ⍴A must the same as ⍴B, except that ⍴A does not include the dimension extent of the coordinate of B along which the rotation takes place. Note that if A and B are rotate conformable, then they are also catenate conformable.

TRANSPOSE ⍉

Monadic ⍉ is the ordinary transpose operator. It reverses the coordinate numbering of all coordinates of its argument: (⍴⍉A)=⍉⍴A.

Clearly, ⍉A has no effect if the rank of A is less than 2.

In dyadic transpose V⍉A, V must be a nonnegative integer vector of length equal to the rank of A, so that V[I] corresponds to the Ith dimension of A. Then, dimension I of A becomes dimension V[I] of the result.

The dimension indices in V follow the index origin.

It is not necessary that the integers in *V* be different: if two or more integers of *V* are equal, then that dimension of the result is composed of elements taken from the major diagonal crossing the dimensions of *A* that map into it (If the involved dimensions of *A* are not identical in extent, then the diagonal ends at the edge of the shortest dimension.). For example, 1 1⍉*A* is the ordinary major diagonal of the matrix *A*.

It is required, however, that all dimensions that will finally appear in the result be specified somewhere in the vector *V*. That is, the vector *V* must consist of the numbers from the index origin (which is the number of the first dimension) through the highest element of *V*, with some possibly repeated, but none missing. Or, stated in APL, every member of ⍳⌈/*V* must be present at least once in *V*. (Or: (⍳⌈/*V*)∊*V*.)

Note that: (⍴*V*⍉*A*)[*V*]=⍴*A*, and (⍴*V*⍉*A*)=(⍴*A*)[⍋*V*] if there are no repetitions in *V*, otherwise, (⍴*V*⍉*A*)[*V*]≤⍴*A*.

Monadic transpose is related to dyadic by: (⍉*A*)=(⌽⍳⍴⍴*A*)⍉*A*.

COMPRESS / ⌿

Compress is a dyadic operator. In *V*/*A*, *V* must be a boolean vector of length equal to the extent of the last dimension of *A*. The result is obtained by selecting (retaining) those elements along the last dimension of *A* that correspond to a 1 in the vector *V*, and omitting those elements that correspond to a 0.

That is, elements are retained whose last index matches the index of a 1 in *V*; all other elements are "compressed out."

Thus, the result has the same rank as *A*, and has the same dimension extents except for the last, along which it has been compressed.

In *V*⌿*A*, the operation is applied along the first coordinate: elements are retained whose first index matches the index of a 1 in *V*; other elements are compressed out.

A bracketed coordinate index can be used to explicitly specify the coordinate along which compression takes place: *V*⌿[*I*]*A* and *V*/[*I*]*A* are equivalent. The coordinate index follows the index origin.

V can be a one-element boolean value of any rank: if that element is 1, then *A* is retained in its entirety; if it is 0, then the appropriate coordinate of *A* is completely compressed out, and the result is an empty value. (All dimension extents remain unchanged, except that of the coordinate along which the compression took place, which becomes zero.)

EXPAND \ ⍀

Like compress, expand is a dyadic operator requiring a boolean vector as left argument. However, for expand, the number of 1's in the left argument must equal that dimension extent of the right argument that corresponds to the coordinate along which expand is applied.

The result of *V*\A (or *V*⍀*A* or *V*⍀[*I*]*A* or *V*\\[*I*]*A*) is obtained by inserting in *A* either zeros (if *A* is numeric) or spaces (if *A* is character) in positions defined by *V*. 1's in *V* correspond to elements of *A*; 0's in *V* correspond to inserted elements.

The insertion is applied along the explicitly or implicitly specified coordinate; the coordinate index follows the index origin.

As with compress, the result has the same rank as *A*, and has the same dimension extents, except for that of the coordinate that was expanded.

Note that compress is the inverse of expand: *A*=*V*/*V*\A.

MEMBERSHIP ϵ

The result of *A*ϵ*B* is a boolean value identical in shape to *A*, with 1's corresponding to those elements of *A* that are found to occur somewhere (anywhere) within *B*, and 0's for those elements of *A* that are not found in *B*. The shape of *B* is irrelevant: *B* merely represents a collection of elements, and ϵ determines which elements of *A* are members of the collection and which are not.

A and *B* can be of any type, rank, and shape, and can differ in any or all of these properties. Of course, if *A* and *B* are not of the same type, the result will be all 0's: (⍴*A*)⍴0=*A*ϵ*B*.

Note that no errors are detected by membership.

ENCODE ⊤

A⊤*B* encodes each element of *B* into its positional representation in any one or more number systems, the radices being specified by the numeric value *A*. Each element of *A* represents the radix applicable to the corresponding position in the representation being generated. If the rank of *A* is 2 or greater, then each row of *A* is interpreted as a set of radices for a single number system; each element of *B* is encoded into all of the number systems represented by *A*: all combinations of numbers and number systems are used.

The result is a value whose shape is the catenation of those of *A* and *B*.

DECODE ⊥

A⊥*B* is the inverse of encode. It accepts a numeric value *A* defining the radices of the positions in one or more number systems, and a value *B* containing digits representing one or more numbers in those systems. The result has rank 2 less than the sum of those of *A* and *B*: (⍴⍴*A*⊥*B*)=¯2+(⍴⍴*A*)+(⍴⍴*B*).

The shape of *A* must equal the first dimension extent of *B*, unless *A* is a one-element value, in which case *A* is reshaped to the appropriate length.

To be meaningful, *B* must be of the form produced by encode.

DEAL ?

Though monadic ? is a scalar operator, dyadic *A*?*B* is a mixed operator. *A* and *B* must be one-element integer values such that: (0≤*A*)∧*A*≤*B*. The result is a vector of *A* elements selected randomly and without replacement from the set ⍳*B*. The effect is that of shuffling a deck of *B* cards, and then dealing *A* of them.

The set ⍳*B* consists of either the integers from 0 to *B*-1, or from 1 to *B*, depending upon whether the index origin is set to 0 or to 1 respectively.

Details of the random number generation algorithm used by the deal operator can be found under "Roll", earlier in this section.

MATRIX INVERSE ⌹

Monadic ⌹ is matrix inverse. In ⌹*A* the argument *A* must be a numeric matrix or one-element value. If a matrix, it must have at least as many rows as columns; otherwise a *LENGTH ERROR* results.

If *A* is a one-element value, then (⌹*A*)=÷*A*; that is, the result is the ordinary reciprocal of *A*.

However, if *A* is a multielement value, then the result has the shape (⍴⌹*A*)=⌽⍴*A*; that is, the result has the same shape as the transpose of *A*. The elements of ⌹*A* are chosen to least-squares best-fit the ordinary matrix product of *A* and ⌹*A* to the identity matrix of order 1↑⍴*A* (the number of rows in *A*). That is, if *X* is the result of ⌹*A*, then the elements of *X* are chosen to minimize +/,((*A*-.×*X*)-*I*)*2 where *I* is the identity matrix.

If *A* is a square matrix, then ⌹*A* is the ordinary matrix inverse of *A*. If *A* is over-square (more rows than columns), then *A* is not exactly invertible, and ⌹*A* is the least-squares best inverse. If *A* is under-square, then ⌹*A* results in a *LENGTH ERROR*.

If *A* is a singular square matrix, then ⌹*A* yields a *DOMAIN ERROR*.

MATRIX DIVIDE ⌹

Dyadic ⌹ is matrix divide. The result *X* of *A*⌹*B* is chosen to least-square best-fit the matrix product of *B* and *X* to *A*. More precisely, the elements of *X* are chosen to minimize +/,((*V*+.×*X*)-*A*)*2.

A and *B* must be numeric values.

The shape and conformability requirements of matrix divide are as follows:

1) If *B* is a one-element value, then: *B* can have any rank; *A* can have any rank and shape; the result has the same rank and shape as *A*; the result elements are computed using division, *A*÷*B*.

or

2) If *B* is a multielement value, then:

a) *B* must be a matrix, 2=⍴⍴*B*;
b) *A* must be a matrix or a vector, (⍴⍴*A*)∊1 2;
c) The first dimension extent of *A* and *B* must be equal, (1↑⍴*A*)=1↑⍴*B*;
d) *B* must have at least as many rows as columns, (1↑⍴*B*)≥1↓⍴*B*;
e) Violation of (a) or (b) yields a *RANK ERROR*;
f) Violation of (c) or (d) yields a *LENGTH ERROR*;
g) The result has the shape (⍴*A*⌹*B*)=(1↓⍴*B*),1↓⍴*A*;
h) the result is the ×/1↓⍴*A* sets of least-squares best solutions in 1↓⍴*B* unknowns to the ×/1↓⍴*A* sets of 1↑⍴*B* linear equations in 1↓⍴*B* unknowns.

I-BEAM ⌶

I-beam ⌶ is a monadic operator that accepts as argument a one-element integer value whose element is chosen from a small set. The result is the value of some system dependent parameter, which particular one being selected by the argument.

All times are in sixtieths of a second; all results are scalars, integers, except ⌶27. The results are defined as follows:

⌶19 The real time since this instance of APL was invoked.
⌶20 The time of day.
⌶21 The CPU time used since this instance of APL was invoked.
⌶22 The amount of workspace remaining available to be used, in units of 9-bit bytes (i.e., four times the number of words).

This number reflects the fact that a Multics APL workspace can be many segments in size. However, since any single APL value must fit wholly within one segment, it is possible for some APL expressions to cause errors even when I-beam 22 is returning large values. For example, it is impossible to create a 2,000,000 character item, even in a workspace with millions of characters of space available.

⌶23 The number of Multics users currently logged in.
⌶24 The time of day that this instance of APL was invoked.
⌶25 The date, as a 6-digit integer, MMDDYY.
⌶26 The first element of I-beam 27 (or 0 of I-beam 27 is empty).
⌶27 The vector of statement numbers in the state indicator, most recent first. An element of 0 corresponds to pending evaluated input (⎕) entries in the state.

FORMAT ⍕

Monadic format ⍕ converts numeric arguments to character form, and returns character arguments unchanged. The result has the same appearance, when displayed, as that of the argument. The result is formatted using the same rules as numeric output, but this result is explicitly available, instead of being printed.

The rank of the result is a vector if the argument is a scalar. Otherwise, the result has the same rank as the argument. The length of each dimension of the result, except the last, is the same as the corresponding dimension of the argument. The length of the last dimension is equal to the number of columns in the printed represenation. Each element of the argument generally takes several columns to print.

The result never has a trailing column of blanks. If the result is a vector, it will not have any leading blanks, either. Each column of numbers is formatted independently, so that each column in the result occupies as few characters as possible. The width of each column is chosen so that only one blank separates adjacent columns.

The value of ⎕*PP* is used to determine the number of digits printed. The printing width, ⎕*PW*, is considered to be infinite.

The result produced by monadic format may vary from one release of APL to another, so users needing precise control should use dyadic format.

Dyadic format ⍕ converts numeric arguments to character form under the control of the left argument, and does not accept character arguments. In the general case, the left argument contains one pair of values ("format pair") for each column in the right argument. Each format pair controls the spacing, type, and precision of one column of the result.

The right argument to format is the numeric value to be converted. It may have any rank and shape. A scalar is considered to have one column, a vector has as many columns as elements, and an array has as many columns as the length of its last dimension. The left argument to format can be a scalar integer, a pair of integers, or as many pairs of integers as columns in the right argument.

The rank of the result is a vector if the argument is a scalar. Otherwise, the result has the same rank as the argument. The length of each dimension of the result, except the last is the same as the corresponding dimension of the argument. The length of the last dimension depends on the values specified by the format pair.

Each format pair controls the conversion of one column of values from the right argument into one <u>field</u> of successive columns of characters in the result. If only one format pair is specified, it applies to all columns.

The first element of the format pair specifies the field width. This is the total number of characters each value occupies. All values are right-aligned in their fields. If a field width of zero is specified, a default width is chosen such that exactly one blank separates the longest number from the neighboring column.

If only one format pair is specified, a field width of zero causes the default width to be the same for all columns, otherwise the default width is computed only for the specific column.

The field width can be from 0 to 255, inclusive.

The second element of a format pair specifies the type and precision of the formatted value. The sign of the second element specifies the type (negative for exponential, or scientific, form; zero for integral form; positive for fixed decimal form). The absolute value of the second element specifies the precision. This is the number of significant digits in scientific form and the number of decimal places in fixed decimal form. If a single element is specified as the left argument to format, it is taken as the precision, and a default field width is used.

The precision can range from 1 to 19 for scientific form, and from 1 to 57 for fixed decimal form. A 0 indicates integral form.

There is no requirement that a blank be left between fields if the field width is specified explicitly -- thus boolean values may be tightly packed, for example. If a formatted value will not fit in the specified field width, a domain error occurs.

Notes

Values are formatted by converting from binary to decimal, rounding the decimal value to either the specified number of digits, to an integer, or to the specified number of decimal places, and then converting the decimal value to character form. Decimal values are precise to 19 digits before rounding. Since small negative numbers can round to zero, the sign of the original value can be lost.

Both monadic and dyadic format reserve three character positions in exponential form for the sign of the exponent and two digits, even though only one of them may be needed. Trailing blanks are added, as necessary, so that columns of values are aligned by the decimal point. No decimal point is produced if only one significant digit is requested in exponential form.

Format produces a leading zero in fixed decimal form if the absolute value of the decimal value is less than one. Monadic format suppresses trailing zeros, and any trailing decimal point, in fixed decimal form. Dyadic format never suppresses trailing zeros.

COMPOSITE OPERATIONS

Composite operations use scalar operators to perform very frequently used operations that would otherwise require a host of complex and time-consuming user-supplied functions. In a few keystrokes, they define operations that would require 100 to 1000 keystrokes to define otherwise. And, they can perform these operations 100 times faster than an equivalent APL function.

APL has four composite operations: reduction, scan, outer product, and inner product. Syntactically, they all behave like operators, reduction and scan being monadic, and outer and inner product being dyadic. Furthermore, they all use scalar operators in both their definition and printed representation. However, they do not behave like scalar operators; they behave like mixed operators.

For all composite operations, the following are true:

- It is syntactically equivalent to an operator.
- It is defined only with scalar operators.
- One or two scalar operators are present in its printed representation; the same number are used in its definition.

- Its elemental domain is determined by those of the scalar operators used in its definition.

- Its rank, shape, and conformability requirements are determined by the operation (reduction, outer product, inner product), not by the operators.

REDUCTION ⊙/ ⊙⌿

Reduction is a composite operation consisting of a slash / preceded by the symbol of any standard APL dyadic scalar operator. For example, plus-reduction is +/ and maximum reduction is ⌈/. (Scalar operators are described under "Scalar Operators", earlier in this section.)

When it is necessary to discuss reduction in general, it will be shown as ⊙/ with the understanding that the ⊙ symbol, which has no APL meaning, stands for the symbol of any dyadic scalar operator.

Reduction behaves like a monadic mixed operator: it accepts a single argument, a right argument. The argument can have any rank and shape.

When applied to a vector argument, the result of ⊙/*V* is the same as placing the scalar operator ⊙ between each adjacent pair of elements of *V*. For example, if *V* is a four-element numeric vector, then +/*V* is the same as *V*[1]+*V*+[2]+*V*[3]+*V*[4]. (The subscripts are in 1-origin indexing.) Thus, the plus-reducing of *V* is the sum of the elements of *V*, returned as a scalar.

If the argument of reduction is an empty vector, the result is the identity element for the scalar operator involved, if it has one; otherwise, it is a *DOMAIN ERROR*. Identity elements for the dyadic scalar operators are shown in Table 3-2.

Table 3-2. Identity Elements for Dyadic Scalar Operators

Operator		Identity
Name	Symbol	⊙/*ZO*
add	+	0
subtract	-	0 (right)
multiply	×	1
divide	÷	1 (right)
power	⋆	1 (right)
logarithm	⍟	none
maximum	⌈	¯1.7014118346046923*E*38
minimum	⌊	1.7014118346046923*E*38
residue	∣	0 (left)
combinations	!	1 (left)
circular	○	none
and	∧	1
or	∨	0
nand	⍲	none
nor	⍱	none
less	<	0 (left,boolean)
less-equal	≤	1 (left,boolean)
equals	=	1 (boolean)
not-equal	≠	0 (boolean)
greater-equal	≥	1 (right,boolean)
greater	>	0 (right,boolean)

If the argument of reduction is a scalar, it is treated as a one-element vector. The result of reducing a one-element vector is always simply the single element itself, returned as a scalar, if in the domain of the reducing scalar operator. (+/'*X*' yields a *DOMAIN ERROR*.)

If the argument of reduction is a matrix or array, then the reduction is along the "vectors" that form the last dimension of the value. The result has rank one less than the argument, and has the same shape as the argument, except for the disappearance of the reduced-over last dimension.

Reduction can be performed along coordinates other than the last: ⊙⌿*A* signifies ⊙ reduction along the first coordinate of *A*, ⊙/[*I*]*A* and ⊙⌿[*I*]*A* signify reduction along the *I*th coordinate of *A*, where *I* is a one-element value whose element is among the set ⍳⍴⍴*A*. (Coordinate index follows the index origin.)

In general, the result has rank one less than that of the argument, and has the same shape, except for the disappearance of the reduced-over dimension.

The exact order in which the repeated scalar operations of a reduction are performed is sometimes of consequence -- it is not in the case of plus-reduction -- but is in the case of minus-reduction.

For example, if V is a four-element vector, -/V gives V[1]-(V[2]-(V[3]-V[4])) which is different in value from ((V[1]-V[2])-V[3])-V[4].

The rule is that the operations are performed in right-to-left order; i.e., the first operation performed is the rightmost one, and the result of that operation becomes the right argument of the next operation to the left, and so on. As discussed under "Right-to-Left Rule", later in this section, this is the same interpretation given to V[1]-V[2]-V[3]-V[4] if it were typed directly.

SCAN ●\ ●⍀

<u>Scan</u> is a composite operator consisting of a backslash \ preceded by the symbol of any standard APL dyadic scalar operator. For example, plus-scan is +\ and maximum-scan is ⌈\.

As with reduction, the form ●\ indicates that the discussion can be generalized to any dyadic scalar operator scan.

Like reduction, scan behaves like a monadic mixed operator, accepting a single, right argument. Also like reduction, the argument can have any rank and shape. <u>Unlike</u> reduction, scan preserves the structure of its argument; that <u>is</u>, the result of a scan has the same rank and shape as its argument, without exception.

When applied to a vector argument V, the result R of ●\V is defined as follows (origin 1): R[I]=●/I↑V for each I∊⍳⍴V. That is, R[1]=V[1], R[2]=●/2↑V, R[3]=●/3↑V, ..., and R[⍴V]=●/V. For example, 1 3 6 10 15 21 28 = +\⍳7. Thus the plus-scan of V is the vector of cumulative sums of the elements of V.

Unlike reduction, if the argument of scan is an empty vector, the result is an empty vector; no identity elements are involved.

Like reduction, if the argument of scan is a scalar, it is treated as a one element vector, except that the result is a scalar, not a one-element vector.

Like reduction, if the argument of scan is a matrix or array, then the scan is along the "vectors" that form the last dimension of the value. The result always has the same rank and shape as the argument.

Like reduction, scan can be performed along coordinates other than the last: ⊙⍀*A* signifies ⊙ scan along the first coordinate of *A*; ⊙\[*I*]*A* and ⊙⍀[*I*]*A* signify scan along the *I*th coordinate of *A*, where *I* is a one-element value whose element is among the set ⍳⍴⍴*A*.

Note the following relationships (*A* is boolean):

```
(~<\~A)=≤\A      (~≤\~A)=<\A
(~∧\~A)=∨\A      (~∨\~A)=∧\A

(×\A)=∧\A        (⌊\A)=∧\A
(⌈\A)=∨\A        (!\A)=<\A
```

All scan relationships hold for reduction.

OUTER PRODUCT ∘.⊙

Outer product is a composite operation consisting of the symbols ∘ and . followed by the symbol of any standard APL dyadic scalar operator. When outer product is discussed in general, it is shown as ∘.⊙.

Outer product behaves like a dyadic mixed operator, accepting two arguments, right and left. Each argument may have any rank and shape.

The elemental domain of outer product (i.e. valid type and element values) depends upon the particular scalar operator in use.

The result of outer product always has rank equal to the sum of the ranks of its arguments, and has shape equal to the catenation of the shapes of its arguments. That is, *A*∘.⊙*B* has rank (⍴⍴*A*)+⍴⍴*B*, and shape (⍴*A*),⍴*B*.

The elements of the result of *A*∘.⊙*B* are computed by applying the operator ⊙ to every *AB* pairwise combination of the individual elements of *A* and *B*, taken in row-major order.

Inner product is a composite operation built up out of any standard APL dyadic scalar operator, followed by a dot ., followed by another dyadic scalar operator. When inner product is being discussed, in general, it is shown as ⊙.⊕.

Inner product behaves like a dyadic mixed operator, accepting two arguments, right and left. In *A*⊙.⊕*B*, the last dimension extent of *A* must equal the first dimension extent of *B*.

If *A* and *B* are both vectors, then the result of *A*⊙.⊕*B* is a scalar whose value is ⊙/*A*⊕*B*. This is, the elements of *A* and *B* are passed as arguments to the ⊕ dyadic scalar operator, and the result vector is reduced with the ⊙ operator to form a scalar result.

More generally, if *A* and *B* are of higher rank, then each "vector" forming the last dimension of *A* (there may be many of them, as determined by the preceding dimensions of *A*) is paired with each "vector" forming the first dimension of *B* (again, there may be many of them, as determined by the succeeding dimensions of *B*) to form a single element of the result, just as in the vector-vector case. (The two vectors are passed as arguments to the ⊕ operator, and the result reduced with the ⊙ operator.)

Every possible *AB* pairwise combination of the "vectors" of *A* and *B* is formed in this way, and yields one element of the final result.

Note that the elemental domain is determined only by that of the ⊕ operator. However, the range of the ⊕ operator must be in the domain of the ⊙ operator. For example, *'ABCD'*=.=*'AACC'* works, but *'ABCD'*÷.=*'AACC'* yields a *DOMAIN ERROR*.

The result of *A*⊙.⊕*B* has shape equal to the catenation of the shapes of its arguments, except that the last dimension extent of *A*, and the first dimension extent of *B*, are missing (these two dimensions are lost in the reduction process).

EXPRESSIONS

An expression is any valid combination of APL symbols which, when executed, produces exactly one explicit result: an APL value.

Expressions may be of any length, and may produce any number of implicit results.

Right-to-Left Rule

In every APL expression, each operation (operator, composite operation, function, system function, and pseudo-operator) takes -- for its right argument -- the value produced by the entire subexpression to its right, and takes for its left argument -- if it requires one -- the value immediately to its left.

Thus, in every APL expression, the first operation to be performed is the right-most one; the result of that operation becomes the right argument for the next operation to the left; its result is passed to the next operation to the left, and so on.

Operation precedence is therefore positional, rather than attributive (as in algebra, and most other programming languages). This may seem unusual at first, but the large number of APL operators would render almost any attributive precedence scheme unworkable. Thus, positional precedence is a natural choice. The selection of right-to-left ordering is due to monadic operations taking of right arguments rather than left arguments. (Left-to-right ordering could have been chosen as well, so long as all monadic operations were defined to take only left arguments.)

Subexpressions

A subexpression is an expression that is part of a larger expression.

Note that a subexpression need not preserve the meaning of its containing expression, and therefore need not be delimited.

For example, the expression 1+2,3⍴4+⍳2 contains the following subexpressions,

```
2,3
1+2
3⍴4
1+2,3
2,3⍴4
```

and more, all of which are inconsistent with the intent and meaning of their containing expression, since they violate its syntax. For example, 3⍴4 violates the syntax of the original expression because the 4 is the left argument of the rightmost add + in the original expression, yet is used as the right argument of reshape ⍴ in this subexpression; the intended right argument to reshape ⍴ is 4+⍳2. Similarly, 1+2 has the 2 as right argument to add +, whereas the 2 is the left argument to catenate , in the containing expression; the intended right argument to this add + is the implicit subexpression 2,3⍴4+⍳2.

Implicit Subexpressions

An implicit subexpression is a subexpression that is delimited implicitly -- that is, due to APL syntax -- rather than explicitly -- such as with parentheses -- and therefore preserves the meaning of its containing expression.

Every APL expression consisting of at least one operation contains at least one implicit subexpression.

For example, the expression 1+2,3⍴4+⍳2 contains only the following implicit subexpressions, in order of increasing length:

```
2
⍳2
4+⍳2
3⍴4+⍳2
2,3⍴4+⍳2
```

The following are not implicit subexpressions of this expression, since their argument associations are inconsistent with those defined by the syntax of the original expression:

```
3⍴4
2,3
1+2
1+2,3
1+2,3⍴4
```

Explicit Expression Delimiters ();[]⍝:⋄

Eight characters explicitly delimit APL expressions when outside of character constants: ();[]⍝:⋄. Each has one or more special meanings associated with it, which are explained below.

Two "characters" implicitly delimit APL expressions: bol ("beginning of line") and eol ("end of line").

Explicit Subexpressions

An explicit subexpression is a subexpression that is explicitly delimited by one of the following eleven pairs of explicit expression delimiters;

```
()
[]
(;
;;
;)
[;
;]
;⍝
;⋄
:;
⋄;
```

or by one of the two pairs bol ; or ;eol, where "bol" denotes "beginning of line" and "eol" denotes "end of line."

Every explicit subexpression preserves the meaning of its containing expression. (This is because any delimited subexpression preserves the meaning of its containing expression.)

Note that the following nine pairs of expression delimiters -- none of which are in the above list -- can delimit a valid APL expression, but not a subexpression:

```
:⍝
:◇
:eol
◇⍝
◇◇
◇eol
bol⍝
bol◇
bol
eol
```

No other pairwise permutation of expression delimiters can delimit an expression or subexpression.

Parenthesized Expressions and Subexpressions

Any APL expression or implicit subexpression can be enclosed in parentheses nested to any depth. However, the parentheses will have no effect whatever. (If the reason for this is not immediately obvious, review the definitions of expressions, the right-to-left rule, and implicit subexpressions.)

Parentheses have operational significance only when they define (delimit) subexpressions that do not qualify as implicit subexpressions. Subexpressions delimited by parentheses are explicit subexpressions called parenthesized subexpressions.

Each parenthesized subexpression is evaluated separately, and its result replaces it in the evaluation of the remainder of the expression. A given parenthesized subexpression is evaluated just before the execution of the operation that takes the result of that parenthesized subexpression as an argument, and no sooner.

Lists

A list is a sequence of any number (including zero) of APL expressions, separated by semicolons. Lists have three uses in APL: indexing, mixed output, and multi-value argument passing to external functions and system functions.

Indexing

Indexing is an operation that allows the detailed selection of an arbitrary number of elements, or blocks of elements from any non-scalar APL value. In algebra and most other programming languages, this is called subscripting. However, since there is no means for entering a subscript on most terminals, and since the element selection is made by specifying element indices, indexing is an appropriate term for this operation.

To index a given value, the value is immediately followed by a bracketed list of expressions that produce nonnegative integer values as results. The number of expressions in the list must equal the rank of the indexed value. Therefore, the number of semicolons in the list must be one less than the rank of the indexed value.

Clearly, indexing a vector requires a list with only one expression, and no semicolons; that is, a vector is indexed by a bracketed expression. Indexing a matrix, however, requires a list of two expressions, with exactly one semicolon.

The elements of the result value of each expression of the index list are interpreted as indices for the coordinate of the indexed value that corresponds to that expression, of elements of the indexed value. This is, the elements of the result of the first expression specify first coordinate indices of elements of the indexed value; the elements of the second expression result specify second coordinate indices; and so forth.

For example, if *A* is a 6 by 10 matrix, then *A*[1 3 4;2 7] selects elements of *A* that have a first coordinate index of 1 or 3 or 4, and a second coordinate index of 2 or 7. That is, selected elements must be in the first or third or fourth rows, and must be in the second or seventh columns. Clearly, there are six such elements.

Note that in the example, the numbers of elements in the two expression results are not equal. Furthermore, the selection effectively uses all possible pairings of one element from each of the two expressions. This is true in general. Therefore, the number of selected elements is the product of the numbers of elements of the selecting expression results. (This is not the only possible scheme. For example, if it were required that all expression results have the same number of elements, then these elements could be paired (in the matrix case) sequentially to enable the sparse selection of elements. For example, *A*[1 3 4;2 4 7] would -- in this system -- select three elements: *A*[1;2], *A*[3;4], and *A*[4;7]. However, although this type of indexing can be useful, it is not nearly as useful as the scheme selected for use in APL.)

The shape of the result of indexing is determined by the shapes of the results of the expressions in the index list, not by the shape of the indexed value: the shape of the result is the catenation of the shapes of the results of the indexing expressions. Clearly, the rank of the result is the sum of the ranks of the results of the indexing expressions.

For example, if all indexing expressions produce scalars, then the result is a scalar. However, if all indexing expressions produce one-element vectors, then the result is a one-element value with rank equal to that of the indexed value. (Ignoring the rank of one-element results of indexing expressions is likely to produce erroneous later results: *RANK ERRORS* due to superfluous dimensions of unity extent. Use of ''⍴ on one-element indexing expression results will yield the usually desired scalar.)

All indices used in indexing follow the index origin.

All indices must be among the set ⍳(⍴*A*)[*I*], where *I* specifies the coordinate being indexed and *A* is the indexed value. Any index not in this set is invalid: use of invalid indices yields an *INDEX ERROR*.

A special feature of APL indexing that does not fit into the rules above is the elided expression: any expression in the index list can be omitted. The default expression, used in place of elided expressions, is effectively ⍳(⍴*A*)[*I*], where *A* is the indexed value and *I* is the coordinate index of the coordinate being indexed by that expression. This has the effect of selecting all indices along the coordinate of the elided expression. For example, if *A* is the 6 by 10 matrix used above, then *A*[1;] selects the first row of *A*, returning it as a 10-element vector; *A*[;3] selects the third column of *A*, returning it as a 6-element vector; *A*[2 5;] selects the second and fifth rows of *A*, returning them as a 2 by 10 matrix; and *A*[;] selects all elements of *A*, returning them as a 6 by 10 matrix identical to *A*.

Note that for the above example, *A*[] yields a *RANK ERROR*, since *A* is a matrix, and the index list is suitable only for a vector. Thus, when eliding index list expressions, do not elide any semicolons.

Mixed Output

A list of implicitly defined expressions delimited by pairs of semicolons produces mixed output when evaluated:

```
;⍳5;' IS A VECTOR CONSISTING OF ' ; 5 ; ' ELEMENTS ';
```

produces the following output:

```
1 2 3 4 5 IS A VECTOR CONSISTING OF 5 ELEMENTS
```

Mixed output is the output (printing) of the explicit result of each expression in turn, one after another, with no additional spaces or newlines inserted between them. All spaces and newlines that are part of the normal printed representation of each value are printed as usual. Since some of the expressions may produce character values and others numeric values, mixed output is one way to form output with mixed type in one line.

The explicit results of the list expressions are printed starting with that of the left-most expression in the list, and proceeding to the right. This may at first seem inconsistent with APL's usual right-to-left ordering, but note that the order in which the expressions are evaluated is explicitly undefined in APL: only the left-to-right order in which the explicit results of those expressions are printed is defined. Furthermore, left-to-right is the way terminal printers print, and is therefore appropriate for mixed output.

Any mixed output list entry that produces no explicit result -- and is therefore not an expression -- is an error, as mixed output requires a value from each expression in the list. Evaluation of such a mixed output list yields a *VALUE ERROR*.

However, elided list entries are valid: they are equivalent to a list expression of ⍳0. That is, elided entries are completely ignored and do not affect the printed output.

Note that for the purposes of defining mixed output, a mixed output list must contain at least two expressions: a one-expression "list" delimited by one of the above pairs is not a list, it is an expression.

Argument Lists

A list that is delimited by parentheses -- a parenthesized list -- can be used as an argument to two special types of functions discussed in more detail later in this manual: external functions, and system functions.

An external function is a user-defined executable entity written in PL/I, compiled, and made available in APL via the)*DFN*,)*MFN*, or)*ZFN* system commands (See Section 5.). It can have any syntax that an APL function or operator can have; its syntax can also be dependent upon how it is called: it can therefore have both a monadic and dyadic definition; either of its arguments can be a parenthesized list.

A system function is an executable entity similar to an external function, but with two differences: it is a permanent part of APL (Users cannot create one; they are always available to the user, just as operators are.); its "name" begins with a quad ⎕.

```
A←⎕CS
```

or

```
A←'I3' ⎕FMT (⍳12;10+⍳12;100+⍳12;1000+⍳12).
```

Comments

Comments can be appended to any complete line of APL by using the lamp ⍝ as the left-most character of the comment. APL ignores everything to the right of any lamp ⍝ that is not imbedded in a character constant.

If outside of any character constant, the lamp ⍝ is equivalent to an eol: it can serve as a right delimiter of implicit subexpression, expression, mixed output list, statement, function line, and any diamond line.

Labels

A label is a part of a function line that effectively "names" the line. Any reference to the label produces as an explicit result the line number of the line on which the label is defined. A label is defined by placing its name, followed by a colon, immediately to the left of the definition of the function line to be "labeled".

This is the only valid use of the colon in APL syntax.

If outside any character constant, and if to its left is a valid APL name (discussed later), the colon is equivalent to a bol, except that it has the side-effect of defining a label: a colon can serve as a left delimiter of an expression, mixed output list, statement, or diamond line.

Labels are discussed in more detail later in this section.

Statements

A statement is any valid sequence of APL operations that is delimited by one of the following nine pairs of delimiters:

```
:⍝
:◇
:eol
◇⍝
◇◇
◇eol
bol⍝
bol◇
bol eol
```

An operation is anything that can take one or more arguments. Operations include: operators, pseudo-operators, composite operations, indexing, mixed output, functions, system functions, and external functions.

An operation need not produce an explicit result.

A statement is a syntactically closed construct: it is entirely self-contained, and therefore cannot act as an argument to any construct, nor can it take an argument from any construct.

A statement's delimiters are not part of the statement.

A statement need not produce an explicit result: if its lowest precedence operation -- that is, the one that is executed last -- does not produce an explicit result, then the statement itself does not produce an explicit result.

Only the lowest precedence operation is eligible to not produce an explicit result: if an operation with higher precedence fails to produce an explicit result then a *VALUE ERROR*, *SYNTAX ERROR*, *CONTEXT ERROR*, or *USAGE ERROR* occurs, depending upon the cause.

The statement is the most general APL construct that contains only operations.

The delimiter pairs listed above imply that a statement cannot be a portion of a "larger" statement. This is in contrast to an expression, which can be a portion of a larger expression.

Closed Expressions

A closed expression is an expression which is a statement: its explicit result is not taken as an argument of any operation.

Diamond Lines

A diamond line is a sequence of any number (including zero) of statements separated by diamonds ◇.

A diamond line is evaluated by the sequential evaluation of its constituent statements, working from left-to-right. The statements are treated completely separately and independently: neither arguments nor results are shared or passed amongst the statements; each statement must have valid APL syntax.

This is the only valid use of the diamond in APL syntax.

If outside any character constant, the diamond in many ways can act similarly to both eol and bol: it can serve as a right delimiter of an implicit subexpression, an expression, any mixed output list, and any statement, and simultaneously can serve as a left delimiter of an expression, mixed output list, or any statement. Any diamond which is inside a character constant is merely a character, not a syntactic entity.

With respect to labels, all statements of a diamond line are considered to be part of the same function line.

Since a statement need not produce an explicit result, any statement of a diamond line can be "vacuous", containing only white space.

A one-statement diamond line is generally called a line.

OBJECTS

An object is a variable, a function, or a group.

Names

A name is used to establish a reference to an object: every object has exactly one name; every name refers to exactly one object.

A name consists of an alphabetic character followed by any number -- including zero -- of alphabetic or numeric characters. For the purposes of this rule, the alphabetic characters are defined as:

∆ABCDEFGHIJKLMNOPQRSTUVWXYZ and
⍙A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲ ;

and the *numeric characters* are defined as:

0123456789 0123456789 .

Note that the underscore _ cannot be the first character of a name.

All characters of a name are significant. Names can be of any length.

Syntax of Names

At least one nonalpha-numeric character must appear between each pair of adjacent -- yet distinct -- names, numbers, or names and numbers, in order to unambiguously separate them. This separator character can be a space or tabulate, or the symbol of a dyadic APL operator, or an explicit expression or value delimiter, or a punctuation character.

All of these separators -- except space and tabulate -- have special meanings in APL, and therefore cannot be used indiscriminately. However, since space and tabulate serve only as separators, they can be inserted whenever necessary or convenient.

Except for the abovementioned special cases, and the additional special case of *distinguished names* (discussed later), the inclusion of spaces and tabulates -- "white space" -- in APL expressions is completely optional.

Identifiers

An *identifier* is a name that refers to -- or is intended to refer to -- a *variable* or a *function* that either currently exists or can exist in the current environment. That is, an identifier is any name that is *not* currently in use as the name of a defined group.

Variable Names

A *variable name* is an identifier that refers to -- or is intended to refer to -- a variable that either currently exists or can exist in the current environment. That is, a variable name is an identifier that is *not* currently in use as the name of a defined (existing) function.

Function Names

A function name is an identifier that refers to -- or is intended to refer to -- a function that either currently exists or can exist in the current environment. That is, a function name is an identifier that is not currently in use as the name of an existing function.

Group Names

A group name is a name that refers to -- or is intended to refer to -- a group that either currently exists or can exist in the current environment. That is, a group name is a name that is not currently in use as the name of an existing variable nor a defined (existing) function.

Variables

A variable is an identifier that currently references exactly one value. That is, a variable is an identifier to which some one value is currently assigned. In order to qualify as a variable, a binding between identifier and value must currently exist.

More simply, a variable is an APL value that has a name. Unlike most programming languages, APL requires no declarations of names: a variable is created by merely assigning a value to an identifier (discussed later).

Variables have no type, subtype, rank, or shape restrictions: any identifier can be assigned any APL value; when a new value is assigned to an existing variable, its previous value is discarded.

The importance of a variable lies in the fact that it is the only mechanism by which a value can be indefinitely and conveniently stored: unless stored in a variable -- or in a less versatile and more cumbersome file, every value produced in the evaluation of an APL expression is discarded by the time its evaluation completes.

A value cannot be assigned to the name of an existing group or function: only existing variables can be redefined in this way; this protection hinders accidental erasure of functions and accidental dispersal of groups.

Syntactically, variables behave as do constants and parenthesized expressions: in evaluation, the identifier is effectively replaced by the value to which it refers.

Any "by-value" reference to an identifier not yet assigned a value -- nor defined as a niladic function returning an explicit result (discussed below) -- yields a *VALUE ERROR*.

Functions

A function is an identifier that currently references exactly one previously defined function definition: a sequence of function lines -- called the function body -- and its associated header (discussed later). In order to qualify as a function, a binding between identifier and function definition must currently exist.

More simply, a function is a stored APL program that has a name. It consists of any integral number (not including zero) of unevaluated, sequentially ordered function lines (discussed later), plus a header that defines the function's syntax and specifies the identifiers to be used temporarily and internally during its execution.

A function can be manually created and altered using the APL function editor (discussed in Section 4); or, it can be created and altered by another function -- with the help of APL system functions.

Syntactically, functions can behave in many different ways, depending upon the specification in the header: functions can mimic operators, or variables, or neither. In evaluation, a monadic or dyadic function -- defined to produce an explicit result -- takes one or two arguments respectively, and produces exactly one explicit result, thereby mimicking monadic and dyadic operators, respectively; similarly, a niladic function -- defined to return an explicit result -- takes no arguments, and simply returns an explicit result that effectively replaces the identifier in the expression thereby mimicking variables; finally, functions can behave unlike operators or variables: a niladic, monadic, or dyadic function -- defined not to return an explicit result takes zero, one, or two arguments respectively, and produces no explicit result (If such a function is to be useful as more than just a cpu-time consumer and/or "bit bucket", then it should produce either an implicit result -- such as a change in environment, or a printed output.).

Groups

A group is a named collection of objects. Grouping the objects allows them to be copied and erased as a unit, without repetitively typing their individual names.

A group can contain any integral number (except zero) of any type of object, including other groups. An object can be in more than one group at a time, since grouping is only a "bookkeeping" convenience, and has no effect whatever on the values of grouped variables or on the definition or execution of grouped functions.

Groups have no significance other than in the)*COPY*,)*ERASE*,)*PCOPY*,)*GROUP*,)*GRPS*, and)*GRP* system commands.

Groups have no syntax, since a group name can never be a valid part of any APL expression.

Groups -- and their associated system commands -- are discussed more fully in Section 5.

Note that by the above definitions, a "variable name" is not the same as a "name of a variable": the "Name of a variable" is literally the name of an existing variable, whereas a "variable name" is a name which either is, or can be, the name of a variable, given the current environment. This distinction applies in an analogous manner to "function name" versus "name of a function" and to "group name" versus "name of a group."

THE ASSIGNMENT PSEUDO-OPERATOR ←

Left arrow ← is the symbol for the dyadic assignment pseudo-operator. A variable name or an indexed name of a variable must appear as its left argument: this argument is evaluated by-reference, not by-value. A value must be its right argument.

The assignment pseudo-operator creates a binding between its left and right arguments.

For the example *VAR_NAME←VALUE*, if '*VAR_NAME*' is not the name of an existing variable, then APL creates a variable named '*VAR_NAME*', assigning to it the value *VALUE*; if *VAR_NAME* is the name of an existing variable, then APL discards its old value and assigns to it the new value *VALUE*.

Indexed Assignment

For the example *NAME_OF_VAR*[index_list]←*VALUE*, *NAME_OF_VAR* must be the name of an existing variable, index_list must be a valid index list for the variable *NAME_OF_VAR* (that is, consistent with the rank and shape of *NAME_OF_VAR*, and *VALUE* must either be a one-element value of any rank, or must have the same rank and shape as *NAME_OF_VAR*[index_list], with the exception that unity dimension extents are ignored when making the rank and shape tests.

The elements of *NAME_OF_VAR* selected by the indices given in index_list are replaced by the corresponding elements of *VALUE* (If *VALUE* is a one-element value, then that element replaces all selected elements of *NAME_OF_VAR*.).

Elements of *NAME_OF_VAR* not selected by the indices of index_list are unchanged.

The type of *NAME_OF_VAR* must match that of *VALUE*: mixed-type values are not permitted.

Neither the rank nor the shape of the variable *NAME_OF_VAR* can be changed using indexed assignment: this operation merely changes the value of existing elements.

The order in which the selected elements of *NAME_OF_VAR* are replaced is explicitly undefined: if some particular element is selected several times and successively replaced by unequal elements of *VALUE*, the final value of that element is undefined and may differ among APL implementations.

Results

Like operators, the assignment pseudo-operator produces an explicit result: its right argument is passed to the left to be used as an argument to successive operations.

Therefore, except for the very important implicit result of either creating a variable or changing the value of an existing one, assignment has no effect on the evaluation of its containing expression.

Output of Explicit Results

APL prints every value that is not taken as an argument to some operation. For example, if *A* and *B* are variables, then the expression *A* causes the value of *A* to be left unclaimed by any operation whatever: thus, it is printed; similarly, *A*+*B* produces an unclaimed explicit result, which is therefore printed.

Assignment is the one exception to this rule: its explicit result is not printed, even when no operation takes it up as an argument.

Mixed output is the one exception to the rule that the explicit result of every expression in a mixed output list is printed, without exception.

Assignment is called a pseudo-operator because it violates the defined properties of operators; specifically, it violates them in the following respects:

- o one argument must be a reference, not a value
- o it produces an implicit result

THE I/O PSEUDO-VARIABLES ⎕⍞

Quad ⎕ and quote-quad ⍞ are syntactically equivalent to variables, with the single exception of indexed assignments -- an operation not allowed with ⎕ ⍞.

Operationally, ⎕ and ⍞ behave as do variables, except that instead of a value being stored in, or fetched from, the semi-permanent "memory" of a variable, the value is output to, or input from, the user's terminal.

In all other respects, any ⎕ or ⍞ appearing in an expression -- outside of any character constants -- behaves exactly as does a variable.

Evaluated Input ⎕

When not the left argument of assignment, a quad ⎕ indicates a request for evaluated input. When an evaluated input request is encountered during execution of an expression, APL temporarily sets aside evaluation of the remainder of the expression, leaving the execution environment unchanged; APL then prints on the user's terminal the evaluated input prompt ⎕:, followed by a newline and six spaces; APL waits for the user's input, which can be any expression; APL then evaluates this expression in the current execution environment; the explicit result of this input expression effectively replaces the quad ⎕ in the original (source) expression; finally, with the requested input value successfully obtained, APL resumes execution of the original expression at the point of temporary interruption.

If what is input in response to the evaluated input prompt is a statement that produces no explicit result -- if the response given to it is not an expression, then APL prints a *VALUE ERROR* message, then reprompts -- again awaiting input of an expression: APL cannot proceed until the request has been satisfied.

If the input is a statement that yields an error message while in execution at the level of that statement, then APL reprompts -- again awaiting input of a (hopefully) error-free expression.

If the input is a system command other than)*OFF*,)*QUIT*,)*CONTINUE*,)*CLEAR*,)*LOAD*,)*COPY*, or)*EXEC*, then the command is performed, and APL reprompts for evaluated input.

If the input is one of the above system commands, but not)*COPY* or)*EXEC*, then the command is performed; however, due to the action of the command, APL obviously cannot possibly reprompt.

If the input is either)*COPY* or)*EXEC*, then the command is performed; however, due to the potential action of the command, APL may not be able to reprompt.

If the input is an invocation of the function editor, then editing can be performed normally; after exit from the function editor, APL reprompts for evaluated input.

Execution of an expression containing a request for evaluated input never resumes until the requested input value has been obtained.

Other than the rather drastic use of the above five system commands, there are two mechanisms for escaping from a request for evaluated input: the potentially drastic and destructive escape →, and the generally much safer strong interrupt.

Typing a right arrow alone as the response to an evaluated input request causes APL to unwind its execution stack back to the last point of suspension; that is, execution of the current expression, its container (usually a function), that container's caller, that caller's caller, its caller, etc. back to the point at which there are no further callers, is abandoned, with all direct evidence and results of this execution sequence being discarded. (For full Multics users, the escape is essentially equivalent to the Multics release command with no argument. The)*RESET* system command is comparable to the Multics command, release -all.)

The strong interrupt is input by pressing the ATTN key twice in a row. While in evaluated input mode, any strong interrupt causes APL to permanently interrupt execution at the point of the request for evaluated input. Furthermore, APL discards the temporary results obtained from the partial evaluation of the expression. Execution cannot be restarted at the point of interruption, as these results are needed and no longer available. Thus, any resumption of execution must reevaluate at least the entire expression. However, the container of the expression, and all of its callers, are unaffected: the execution stack is unchanged by strong interrupt.

Escapes and interrupts are discussed in detail later in this section.

Character Input ⍞

When not the left argument of assignment, a quote-quad ⍞ indicates a request for character input. When such a request is encountered during expression evaluation, APL temporarily sets aside evaluation of the remainder of the expression; APL then waits for the user's input, which can be any string of characters -- APL does not prompt for character input; APL interprets the input as an arbitrary character vector -- no evaluation or modification is performed; this character vector is the value which effectively replaces the quote-quad ⍞ in the original expression; finally, with the requested input character vector successfully obtained, APL resumes execution of the original expression at the point of temporary interruption.

No error can directly result from the use of character input since no evaluation takes place.

Once a character has typed, there is only one mechanism for escaping for a character input request, and its action is identical to that of a strong interrupt issued in evaluated input mode (Interrupts cannot be issued while in character input mode -- all ATTN's are interpreted as editing the input.): type the three letters *O U T* all overstruck, as *OUT*.

The newline character that terminates the input line is not included in the character vector value of quote-quad input.

Therefore, if no input is given -- only the terminating newline is input, an empty character vector -- equivalent to '' -- is its value.

Any input produces a vector, even if it is a single character: a single character is not returned as a scalar.

Output ⎕ ⍞

Quad ⎕ or quote-quad ⍞, when the left argument of the assignment pseudo-operator, output the value to the user's terminal, rather than to a variable.

Values are printed completely normally, except that the newline which normally terminates the output of values is not printed if quote-quad output is used. (This is the only difference between ⎕ output and ⍞ output.)

THE EXECUTE PSEUDO-OPERATOR ⍎

The execute pseudo-operator takes one argument: a character scalar, vector or single-element array. The argument must be a single diamond line (Note that this precludes labels but not comments). The effect of execute is to evaluate the diamond line in the current environment, performing all implicit and explicit operations specified. Execute can have one explicit result: the explicit result of the rightmost statement. The only case in which execute has no explicit result is when its argument is null or blank.

Execute is useful in several ways. It can be used to convert characters to numbers. It can be used to build APL statements under program control, and then have them evaluated.

Execute is a pseudo-operator because it can have no explicit result.

Execute can be used in a subexpression like any other operator; if it does not return an explicit result a value error occurs.

There is no practical limit to the length of the argument to execute, nor to the depth of recursion of its use.

All *SYNTAX ERROR*s are detected before evaluation of the argument. All other errors are detected during evaluation, as is normal in Multics APL. The verbosity of messages printed when an error is detected during evaluation of execute can be controlled by the)*ERRS* system command.

The explicit result of execute may not be meaningful or useful. If execute is the left-most operation in a statement this result is unclaimed, and hence is printed. To suppress this printing, the result of execute can be assigned to a variable name or to a system variable that ignores assignment (⎕*AI*, for example).

Execute should be used to perform actions than cannot be performed without it, or are very difficult without it; not to perform actions that can just as easily be done another way. This is because the argument to execute must be converted to an internal form each time execute is evaluated, rather than just once, (per)*LOAD* or)*COPY*) as is the case for functions. There is a small, but nontrivial, cost associated with this conversion that can add up if execute is used excessively or unwisely.

PORNOGRAPHY: DEPENDENCE UPON UNDEFINED EVALUATION ORDER

In APL, pornography is defined informally as the dependence upon undefined evaluation order for the successful or correct evaluation of an APL statement.

Two evaluation orderings are explicitly undefined in APL:

- that of the various complete expressions of a list; and
- that of the arguments of any operation.

Any APL function or statement that depends upon assumptions in either of these undefined cases for its correct or error-free evaluation is incorrect APL: any such incorrect APL code cannot be expected to operate consistently when evaluated by different implementations of APL, or even when evaluated by future versions of any given implementation; as these cases are explicitly undefined, an implementation may choose any evaluation order it desires, and is free to change it at any time.

The various complete expressions that form the entries of a list are always evaluated separately and independently: no portion of any expression participates as an argument of any operation in the expression of any other entry in that list.

The semicolons of a list effectively act as "barriers" to the scope of the right-to-left rule, which holds only within single expressions and single explicit and implicit subexpressions.

Therefore, the order in which the separate and independent list entry expressions are evaluated is explicitly undefined.

```
D←A[I+3;I←I+1]
```

Two list expressions reference a variable whose value is changed in one of those expressions. The value *I*+3 is undefined, as it may reference the old <u>or</u> the new value of *I*.

```
'THIRTY = ';N←3;N←0
THIRTY = 30
```

The value of *N* after evaluation of this statement
is undefined.

```
⎕←'ONE';⎕←'TWO'
?
?
ONETWO
```

This statement yields three lines of output, the third
definitely being the string *ONETWO*, that being the mixed
output "result." Whether the other two lines are printed
in the order *ONE TWO* or *TWO ONE* is undefined.

The right-to-left rule defines precisely the order in which operations are performed in an expression, but it leaves undefined the order in which the arguments and coordinate index of a single argument-taking entity are evaluated.

```
(I←2)⌽[I←1]3 4⍴⍳I←12
9 10 11 12
1 2 3 4
5 6 7 8
```

The explicit result is definitely 2⌽[1]3 4⍴⍳12, but
the value of *I* after evaluation is undefined.

```
(A←3)×A
```

Clearly, *A*←3 is evaluated before α×ω. However, the value of ω is undefined: it may reference either the old or new *A*.

```
A×A←3
```

This expression is well defined: 3×3. However, this type of code is obscure and is therefore not recommended.

```
⎕÷⎕
⎕:
17
2
```

Two separate requests for evaluated input occur.
However, which acts as which argument of α÷ω is undefined.
⎕:

`⌈/[''⍴I←(1+⍳1),?2 7 3⍴10]I`	The coordinate index is well-defined: it is 1+ index_origin. However, the value being reduced -- ⍵ in ⌈/[⍺]⍵ -- is undefined: it may reference either the old or new *I*.
`⌈/[''⍴I←]I(1+⍳1),?2 7 3⍴10`	This expression is well-defined (except for the randomness of ?). However, it is obscure. `I←(1+⍳1),?2 7 3⍴10 ◇ ⌈/[''⍴I]I` is well-defined and transparent.

In summary, dependence upon undefined evaluation order occurs only in statements containing assignments to variables referenced elsewhere in that same statement. For the purposes of this statement, any reference to the I/O pseudo-variables must be considered a form of assignment, as must calls to functions that perform relatively global assignments or that reference the I/O pseudo-variables.

Not every statement containing such assignments and references is necessarily undefined, but such statements -- if at all complex -- tend to be quite obscure: such code should be avoided whenever reasonable by splitting the offending statement into several statements.

Obviously, however, it is unnecessary to split `A←A+1` into `T←A+1◇A←T`, but even such seemingly simple statements as `A×A←3` should be split into `A←3◇A×A`.

The following two expressions are taken from a suite of functions recently published in an APL newsletter. Both are well-defined, yet unnecessarily obscure. Both cases are then repeated after being split into well-defined, transparent expressions.

Example 1

```
 E←ρD←¯1↓(C,0)-0,C←(+/~C)↓⍋C←(~≠\''''  =B)∧A  =B←,B,A
```

when *A* and *B* are previously defined.

This monstrosity should be split into the following sequence of expressions:

```
      B←,B,A
      C←(~≠\''''=B)∧A=B
      C←(+/~C)↓⍋C
      D←¯1↓(C,0)-0,C
      E←ρD
```

Example 2

```
 Z[B←(C∧X∊D)/⍳ρX;]←(2 4ρ' Y9  X9 ')[(C←(~≠\''''=X)∧A≤ρD)/A←(D←'ωα')⍳X;]
```

where *Z* and *X* are previously defined.

Similarly, this eyesore should be split into the following expressions:

```
      D←'ωα'
      A←D⍳X
      C←(~≠\''''=X)∧A≤ρD
      B←(C∧X∊D)/⍳ρX
      Z[B;]←(2 4ρ' Y9 X9 ')[C/A;]
```

Split up, these examples are relatively easy to follow, unlike their source expressions. Unfortunately, expressions such as these examples are easy to dredge up from APL publications -- including many APL user's manuals, and have therefore contributed to the undeservedly negative reputation which APL has in some circles.

WORKSPACES

A workspace is a portion of computer memory in which APL stores everything it needs to remember during a session, and in which the user stores an arbitrary -- but usually logically related -- collection of objects. The workspace is the largest logical entity in APL: it is the most general self-contained entity that can be referenced and used as a unit.

Every APL workspace contains: (1) the symbol table, listing the name and storage location of each variable, function, and group; (2) the value of each variable, and the definition of each function and of each group; (3) the state indicator; (4) the value stack, in which temporary and intermediate values are stored by APL; (5) an assortment of dynamic, user-modifiable workspace parameters; and (6) an assortment of static, non-user-modifiable workspace attributes.

The Active Workspace

All computing in APL takes place in a special workspace, called the active workspace.

When APL is first entered, The active workspace is clear: the symbol table has no entries; there are no values, nor function or group definitions; the state indicator is empty; the value stack is empty; the workspace parameters are set to default values; the workspace attributes are undefined or set to dummy values.

APL has a system command that saves a copy of the contents of the active workspace. Another command loads a copy of a previously saved workspace into the active workspace. This permits an APL session to be interrupted and saved, and then resumed at a later date with no loss of information. This also permits a user to maintain any number of saved workspaces, each applicable to some separate task, and to take them up in turn as desired.

The active workspace is implemented by Multics APL as at least four separate segments. Every saved Multics APL workspace is stored as either a single segment, or a multisegment file, as necessary.

SECTION 4

FUNCTIONS

FUNCTIONS

Functions are stored APL programs. They are generally created and modified by the APL function editor -- described later in this section -- although other mechanisms are also available.

More formally, a function is a stored ordered sequence of unevaluated function lines -- called the function body -- preceded by a special line called the header, which defines the syntax, name, and local identifiers of the function.

A function line is a diamond line which may be preceded by a label.

Arguments

Every APL function takes a fixed number of arguments -- either zero, one, or two. Functions are thus characterized as niladic, monadic, or dyadic, respectively.

Results

Every APL function has another fixed property: the ability -- or lack thereof -- to return an explicit result.

A function defined to be unable to return an explicit result cannot -- and will not -- ever return an explicit result. A function defined to be able to return an explicit result may or may not return one, depending upon the arguments given and/or the code in the function body.

As discussed earlier under "Names", a niladic function that returns an explicit result syntactically behaves exactly as does a variable: it takes no arguments, and returns an explicit value. Furthermore, a niladic function defined to be able to return an explicit result syntactically behaves almost exactly as does a variable name: it takes no arguments, and may or may not return an explicit result value; the difference is that a call to a niladic function that returns no explicit result can stand alone in an APL statement without error, whereas a similar reference to a variable name always yields a *VALUE ERROR*.

Similarly, a monadic or dyadic function that returns an explicit result syntactically behaves exactly as does a monadic or dyadic operator, respectively: it takes one or two arguments, respectively, and returns an explicit result.

Regardless of its definition, upon returning to its caller, a function call that has not produced an explicit result yields a *VALUE ERROR* if the context of the function call demands an explicit result.

Operationally and environmentally, functions do <u>not</u> behave as do variables, variable names, or operators: a function may produce implicit results that affect the environment or are printed; a function may execute for an infinite period of time without returning; a function may reference files and variables other than its arguments; a function may call itself or other functions; etc.

<u>Local Identifiers</u>

A <u>local identifier</u> is an identifier that is currently <u>localized</u> by a function call and that <u>masks</u> -- but does not alter or destroy -- any object which that name may refer to in the environment external to this function call.

An identifier is <u>localized</u> by a call to a function that includes the identifier in its header in any position except that of the function name.

A local identifier <u>masks</u> the existence of any object in the environment external to this function call whose name matches the local identifier.

Masking is a change in the execution environment: masked objects can no longer be referenced.

Masking is an immediate consequence of the localization process: it occurs at the time of the function call, before any function lines are executed; it occurs if and only if an identifier to be localized matches the name of an existing object.

Neither identifier localization nor masking causes any object in any environment to be created, modified, or destroyed: only name-object referencing rules are affected, and only for local identifiers and masked objects.

The execution environment reverts to its original state upon return from the localizing function call: masking is removed, and local identifiers and their referents -- _local objects_ created by the localizing function or its callees -- are discarded.

Local identifiers are truly intended "for internal use only:" local objects with any names -- except that of the localizing function -- can be safely and freely created, modified, and erased without alteration or destruction of any nonlocal object of the same name; masked objects are therefore completely safe, as they are isolated and unavailable by any mechanism.

Local Objects

A _local object_ is an object whose name is currently a local identifier.

Only variables and functions can be local: groups are ineligible.

Local objects behave -- in normal execution -- exactly as do their _global_ counterparts: all definitions and rules of APL expression and function evaluation apply as well to local objects. However, some system commands and system functions treat local objects differently than global objects.

Global Identifiers

A _global identifier_ is an identifier that is _not_ currently localized by any function call. That is, the current execution environment is in such a state that the identifier is not localized.

This does not preclude the possibility that a future function call may localize this identifier: it merely indicates that no function call is currently doing so.

Global Objects

A global object is an object whose name is not currently a local identifier.

Any object can be a global object: variables, functions, and groups are equally eligible.

Global objects are the basis for the comparison of the behavior of other objects. Therefore, by definition, they always behave "normally."

Immediately Local Identifiers and Objects

An immediately local identifier is an identifier that is localized by the function call that is on the top of the execution stack; that is, it is local to the most recent function call.

An immediately local object is an object whose name is an immediately local identifier.

The Function Header

The function header, or simply header, defines the syntax, name, and local identifiers of the function.

Headers have two parts: the syntax definition, and the local identifier list.

Syntax Definition

The syntax definition is essentially in the form of a prototype call on the function: it defines the name of the function, the number of arguments it takes, and its ability -- or lack therefore -- to return an explicit result; furthermore, it specifies the local identifiers to be used to copy its arguments into the function, and its result (if any) out of the function.

Clearly, there are six possible forms for the syntax definition -- they are listed below:

```
  FN_NAME (niladic, no result)
  FN_NAME R (monadic, no result)
L FN_NAME R (dyadic, no result)
```

```
  ER←FN_NAME (niladic, possible result)
  ER←FN_NAME R (monadic, possible result)
ER←L FN_NAME R (dyadic, possible result)
```

In these examples, *FN_NAME* is the function name, *R* is the local identifier to which the right argument is automatically assigned when the function is called, *L* is the analogous local identifier for the left argument, and *ER* is the local identifier which is assumed to refer to the value that is returned as the explicit result when the function returns.

ER, *L*, *FN_NAME*, and *R* must all be distinct: any attempt to define a function whose header's identifiers are not all distinct will fail, yielding a *DEFN ERROR*.

ER, *L*, and *R* are local identifiers. However, they are treated specially, as indicated briefly above. Their treatment and significance is explained below.

Argument Identifiers

L and *R* are called the argument identifiers. When a function is called, its arguments -- having been successfully evaluated in the calling expression -- must be made available to it. In APL, this mechanism is quite simple: the function's arguments are automatically assigned to their corresponding argument identifiers.

Thus, before any function line is executed, zero, one, or two local variables are automatically created by APL, corresponding to niladic, monadic, and dyadic functions, respectively.

Result Identifier

ER is the result identifier. When a function returns, if its result identifier refers to an APL value, then that value is returned as the function's explicit result. Otherwise -- and if the function has no result identifier, no explicit result is returned.

Local Identifier List

The local identifier list is an optional list of additional identifiers to be localized when the function is called: each list entry is a single identifier to be localized.

Not all local identifiers will necessarily be found in the local identifier list: result and argument identifiers -- if any, must not be placed in this list; and any labels -- which are implemented as local functions, must also not be placed in this list. Any attempt to violate this rule will fail, yielding a *DEFN ERROR*.

The local identifier list -- if it has any entries at all -- is placed to the right of -- and separated by a semicolon from, the syntax definition.

If the local identifier list has no entries, then no semicolon is placed to the right of the syntax definition.

Elided list entries are not permitted: an identifier separates each pair of adjacent semicolons; the header must not end with a semicolon.

R←NM LOCATE ADDR;R_NM;I;T;CI	This header specifies a dyadic function named *LOCATE*, whose argument identifiers are *NM* and *ADDR*, whose result identifier is *R*, and which additionally localizes *R_NM*, *I*, *T*, and *CI*. Reference to any other identifiers will refer to objects that are not local to this function.
PRINT X;R;RR	This header specifies a monadic function named *PRINT*, which cannot return an explicit result, whose argument identifier is *X*, and which additionally localizes *R* and *RR*.
ER←L XRHO R	This header specifies a dyadic function named *XRHO*, whose argument

identifiers are *L* and *R*, whose result identifier is *ER*, and which does not localize any additional identifiers.

The Function Body

A function body is a sequence of any integral number -- except zero -- of function lines.

A function body must have at least one function line: otherwise, the function can do nothing but consume arguments.

Line Numbers

A line number is the positive integer that is associated with a single function line of a given function.

Every function line has a single corresponding, unique positive integer line number.

Line numbers -- and their corresponding function lines -- are sorted in ascending order in the function body.

The first function line in the function body has the line number 1. Each successive function line has a line number exactly one greater than its predecessor.

The largest line number of the function equals the number of function lines in the function body.

Execution Flow

Unless otherwise directed by a successful branch (see below), APL executes the function lines in the function body in succession according to their line numbers: the function line whose line number equals 1 is executed first followed by the function line numbered 2, followed by line 3, then line 4, then 5, etc. until no more function lines remain to be executed -- that is, the next line number exceeds the largest one in this function; then, the function returns.

This execution flow is realized by the use of the statement counter: each statement in a function is associated with a unique, positive integer statement number -- the first statement of the first function line is statement number 1, the second statement of the first line is statement 2, ..., and the last statement of the last function line is statement n, where n is the total number of statements in the function; the statement counter contains the statement number of the next statement to be executed.

After completing a statement, APL fetches a copy of the next statement to be executed, as given by the statement counter. Then, APL increments the statement counter by 1. Finally, APL executes the new statement.

Thus, if nothing meddles with the value of the statement counter, execution proceeds "straight through" the function, as described above.

However, the sole purpose of the branch pseudo-operator is to change -- under user control -- the value of the statement counter.

The Branch Pseudo-Operator →

The branch pseudo-operator allows the dynamic or static specification of an arbitrary function line execution sequence.

Ostensibly, branch is equivalent to the familiar "go to" of many other programming languages. However, its implementation allows much more sophistication than this implies.

Branch is a monadic pseudo-operator. It produces no explicit result. It must therefore be the left-most graphic of its containing statement.

The argument of branch must be an integer vector or scalar -- otherwise, a *RANK* or *DOMAIN ERROR* occurs.

The implicit result -- if any -- is to change the value of the statement counter to the statement number of the first statement of the function line whose line number is specified in the argument:

- If the argument is an empty vector, then the statement counter is not changed -- the branch is unsuccessful, and execution flow proceeds normally;
- If the argument is a scalar or non-empty vector, then the statement counter is changed to the statement number of the first statement of the function line whose line number is the

first element of the argument -- the branch is successful, and execution flow is redirected.

If the first element of the argument is an integer, but is not a valid line number for this function, the branch sets the statement counter to zero.

Just before attempting to fetch the next statement, APL checks the validity of the statement counter: if it is zero, or is greater than the number of statements in this function, then APL terminates execution of this function by initiating function return.

Note that for the purposes of branching and general execution flow, APL need not keep track of the line number of the function line from which the currently executing statement comes. However, various user interfaces use this line number; so for the convenience of these interfaces, APL does maintain this information. (The implementation of this execution flow system requires only the following: function lines must be broken down into separate statements; an internal vector containing -- for each function line -- the number of its first statement (This vector is indexed by the target line number of successful branches to give the needed target statement number); the total number of statements in this function.)

Note that if the argument of branch is a multielement vector, only its first element has significance: all other elements are ignored and discarded.

Since the argument of the jump pseudo-operator can be the explicit result of an arbitrary APL expression, the target line number can be calculated -- as opposed to being a constant. Furthermore, this enables branching to be conditional as well as unconditional.

Thus, the branch pseudo-operator can be used to perform four types of branching:

```
unconditional-constant →3;
unconditional-computed →3+5×ρρARG;
conditional-constant →((K≤ρARG)∧K≠0)ρ14;
conditional-computed →(N≤ρρTEMP)ρ7 11 15 19[1+ρρTEMP]
```

APL does not have any construct directly analogous to the DO or FOR-NEXT constructs of other programming languages -- its array handling capabilities vastly reduce the need for such constructs. However, highly efficient looping algorithms are easily coded in APL, and often are far more efficient than their brute force, storage-heavy, non-looping, "elegant" counterparts. (As in most fields, beauty -- in this case, "elegance" -- is in the eye of the beholder.) Dramatic examples of some proper -- and improper -- uses of branching are given later.

The target function line of any successful branch is always within the function containing the branch: there is no mechanism for branching from one function into another.

Function Return

APL initiates function return whenever it discovers that its statement counter contains an invalid statement number for the function currently in execution.

Function return is the process by which APL returns from an executing function: first, APL checks to see if the result identifier -- if any -- refers to a value; if so, it copies this value into its execution stack -- to make it available to the calling expression; then, APL restores the local execution environment to its state just before this function call -- that is, APL reverses the localization of its local identifiers -- unmasking any masked objects, and discards all local functions and values; finally, APL -- having copied in any explicit result -- resumes evaluation of the calling expression.

Labels

It should be obvious by now that the use of a line number -- computed or constant -- as the argument of a branch is at best of dubious value, and can easily result in erroneous code. (Consider the problem of inserting or deleting function lines in an existing function definition: associations between line numbers and function lines automatically change. Therefore, care must be taken to modify any line number constants and computations that are affected by these changes. Failure to correctly perform these modifications results in erroneous code.)

To satisfy the need for a simpler and less error-prone mechanism for referencing and branching to specific function lines -- rather than whatever function line happens to be associated with a given line number, APL provides the capability of defining function line labels.

A label or function line label is a local identifier that refers to a local niladic function which returns an explicit result: the scalar integer equal to the line number of the function line on which the label is defined in this invocation of the function.

A label is defined by placing its name to the left of -- and separated by a colon from -- a diamond line of function definition.

A label is automatically created by APL when its localizing function is first entered: the label name is localized, and an appropriate function definition is generated and placed in the local environment.

Every label in a function must have a different name -- otherwise, its result would be undefined. In fact, each local identifier of a given function must be distinct from all other local identifiers and from the name of the localizing function.

Label names must not be included anywhere in the header -- they can only be defined by their presence to the left of a diamond line.

(Recall that a function line is merely a diamond line which may or may not -- that is, is syntactically permitted to -- have a label defined on it.)

Since labels are implemented as niladic local functions which always return an explicit result, they behave just as do local variables, except that their name cannot be the left argument of an assignment -- that is, their result value cannot be changed via an assignment.

In fact, the value returned by a label function cannot be changed by any mechanism, since its function definition is locked, and therefore cannot be edited or otherwise modified by the user.

Recursion

Since each call to a function relocalizes its local identifiers, APL functions can be fully recursive.

It is generally best if recursive functions do not produce implicit results -- implicit results are not localizable, and are therefore difficult to control correctly in recursive functions.

Implicit Results

An implicit result is any change made to the workspace or execution environment that remains in effect after completion of execution of its change-producing construct.

For example, the assignment and branch pseudo-operators produce implicit results: assignment produces a variable or a change in the value of a variable -- changes in the execution environment that remain in effect after assignment is complete; branch may change the statement counter -- a change in the execution environment that remains in effect after this pseudo-operator has completed.

Similarly, a function produces an implicit result if it creates or modifies a global variable or function, as these objects and/or changes remain in effect after the function returns.

Any APL function can be defined to produce any number of implicit results. In fact, any expression can do the same.

However, it is usually undesirable for functions to produce implicit results, as their hidden nature obscures the fact of their existence or occurrence. The semi-permanent change of some workspace or execution environment parameter by a mechanism that obscures this change is usually of little utility, and may easily cause substantial and potentially irreparable destruction of data or other information.

It is recommended that functions -- whenever possible -- produce only explicit results. Any temporarily needed variables or functions should be created locally -- not globally: as local objects, they mask existing objects, and are discarded during function return -- as global objects, they would destroy existing like-named objects, and would remain in the workspace independently of function return -- often cluttering and confusing the workspace and its symbol table with forgotten and useless objects.

Similarly, functions generally should not globally modify workspace parameters, such as the index origin and digits -- these may be locally changed by appropriate use of localized system variables.

Some functions may be required to produce implicit results, as this may be the only mechanism available to perform the desired action.

For example, a function that edits or creates functions must produce one implicit result: the creation or replacement of a global function.

Similarly, a function that selectively erases local or global objects must obviously produce implicit results.

However, except when necessitated by APL's inability to produce the desired results explicitly, functions should not produce implicit results.

Scalar Functions

A *scalar function* is a function that behaves exactly as do scalar operators, except with respect to argument and result types and to symbolic representation.

A function is a scalar function if: it is monadic or dyadic, and capable of returning an explicit result; it references all arguments but only its arguments and result identifier; it uses only scalar operators and the assignment pseudo-operator; every statement is an expression ending in an assignment to the result identifier; every constant is a scalar; assignments are made only to the result identifier; at least one statement references all arguments by value; every statement references by value either all arguments, or the result value and at least one argument.

Scalar functions that do not conform to the above rules can be written, but verifying their scalar behavior is necessarily more difficult.

Scalar functions can be very powerful in appropriate applications -- such as those that are purely mathematical -- as they generalize (extend) to n-dimensional values in the same convenient and natural way as do scalar operators.

Trace Pseudo-Variables *T*∆name of fn

A trace pseudo-variable is a pseudo-variable whose name is composed of the string *T*∆ immediately followed by the name of a function, and that can be used -- with certain restrictions -- just as can any normal variable, except that its proper use produces a side effect -- an implicit result for which this construct was designed -- tracing the execution of its namesake function.

A trace pseudo-variable name can be the left argument of the assignment pseudo-operator -- its right argument can be any APL value.

Similarly, a trace pseudo-variable can be referenced by value -- returning the value most recently assigned to it.

Unlike variables, trace pseudo-variables cannot be listed using any system command, and similarly cannot directly be erased using any system command.

Furthermore, a trace pseudo-variable cannot be created unless the character string following the *T*∆ in its name is the name of an existing function.

Analogously, the trace pseudo-variable is automatically erased when its namesake function is erased.

A trace pseudo-variable name is localized just as are normal identifiers, with two exceptions: a trace pseudo-variable name must not appear anywhere in a header syntax definition; a trace pseudo-variable name must not be used as a label name. However, any number of trace pseudo-variable names may appear in the header local identifier list.

Localizing a trace pseudo-variable name masks any existing identically-named trace pseudo-variable. However, its namesake function is not masked.

Localizing an identifier may mask a function; if so, the masked function's trace pseudo-variable is also masked -- but its name is not localized.

Regardless of any localizations and masking, a trace pseudo-variable name cannot be referenced in any context unless its namesake function name refers -- in the current environment -- to an existing function.

A trace pseudo-variable is used to trace the execution of its namesake function.

The execution of a function is traced by printing -- for each traced function line -- the function name, immediately followed by the bracketed line number of that function line, followed by the explicit result -- if any -- of each statement in that function line; the outputs for the various statements are separated by a string of seven characters: newline, ◇, and five spaces -- which is printed between the execution of each adjacent pair of statements, regardless of whether or not any given statement produces an explicit result.

A statement ending in a branch does not produce an explicit result. However, for the purposes of tracing execution flow, a successful branch is denoted in the trace output for that statement as →n, where n is the target line number. Similarly, an escape is denoted by →.

The function lines to be traced are specified by line number in the trace pseudo-variable for that function. Before executing each function line, APL uses the membership operator to find out if that function line is to be traced: n∊*T*Δname_of_function, where n is the line number of the function line about to be executed, and *T*Δname_of_function is the appropriate trace pseudo-variable.

Since the membership operator is used for this determination, no rank, shape, type, or subtype restrictions on the value of a trace pseudo-variable need be imposed.

For example, if *MAX_RANK* is a seven-line function, and *TΔMAX_RANK*←2 4⍴¯3 92.617 0 15 4 2.9 1 42, then its function lines numbered 1 and 4 are traced when *MAX_RANK* is executed.

Thus, tracing of a given function can be "turned off" by setting its trace pseudo-variable appropriately: assigning ⍳0 or ¯1 or 0 to the trace pseudo-variable works equally well.

Tracing is used almost exclusively for exploring and/or diagnosing complex or allegedly errant functions. It has little -- if any -- practical use outside of these contexts.

A locked function cannot be traced: an attempt to create a trace pseudo-variable for a locked function yields a *USAGE ERROR*; an existing trace pseudo-variable of an unlocked function is erased when that function is locked.

Locked Functions

A locked function is a function whose definition cannot be displayed or modified -- and whose execution cannot be traced -- by any APL user, including the writer or owner of that function.

When first created, most functions are unlocked. However, any function can be locked at any time -- even at the time of its creation -- using the APL function editor. Furthermore, any function can be locked using the ⎕*LOCK* system function or the)*LOCK* system command.

Locking a function does not affect its execution.

A locked function cannot be edited or displayed in any way, whether through the APL function editor, or through a system function: its function definition is totally unavailable to any and all APL users through any mechanism available in APL.

THE STATE INDICATOR

The state indicator -- or SI -- is the APL execution stack: it contains all information needed to control and facilitate execution flow.

The SI is composed of a push down stack of stack frames, each of which is associated with a single call to a function, or to the execute pseudo-operator ⍎, or to evaluated input ⎕.

The information that APL records in each stack frame is that which would be needed by APL to correctly resume execution upon return from a further call to a function, to execute ⍎, or to evaluated input ⎕.

For example, if a function *FOO* has a statement *A*←(2+*B*)×*DEC* 3,*ARG* with variables *B* and *ARG*, and a call to the function *DEC*, then APL needs to remember the statement number -- and point within that statement -- at which the call to *DEC* occurs. Otherwise, upon returning from *DEC*, APL would not know at what point to resume execution of *FOO*.

Stack frames associated with function calls also contain the information needed to "undo" any localization and masking caused by that function call. Clearly, this type of information does not exist for calls to execute ⍎ or to requests for evaluated input ⎕.

Execution Termination

ATTN; WEAK & STRONG INTERRUPTS

The user can interrupt APL execution by pressing the ATTN -- or BREAK -- key. (Note that this use of ATTN is entirely different from its use in editing input lines: this use of ATTN causes APL to discard pending output, to stop evaluation of statements, and finally to read input from the terminal.)

However, APL has two different types of interrupt: the weak interrupt, and the strong interrupt.

Weak Interrupt

The weak interrupt is signalled when APL receives a single ATTN. A weak interrupt causes APL to immediately discard all pending output, but to continue evaluation to the end of the current statement, at which point execution stops.

APL does not send any message to acknowledge its receipt of a weak interrupt.

Strong Interrupt

The strong interrupt is signalled when APL receives two distinct ATTNs in a row -- with no input between them. A strong interrupt causes APL to immediately discard all pending output, and to abort evaluation abruptly -- without regard to how clean or restartable the break is.

APL then types *INTERRUPT* to acknowledge the receipt of the strong interrupt; finally, APL prints out the interrupted source line, placing the error marker ∧ under the point of interruption. Thus, the execution history of the interrupted line is fully documented.

Of course, a strong interrupt can only be signalled after having signalled a weak interrupt. That is, when APL receives the first ATTN, it discards pending output but continues execution of the current statement. When the second ATTN is received, the weak interrupt becomes a strong interrupt: pending output is again discarded; execution halts immediately; and the *INTERRUPT* message and flagged source line are output to the terminal.

ERROR HANDLING

When an error is detected during the execution of a statement, APL aborts further evaluation, types a message naming the error, and prints a copy of the source line containing the error, placing the error marker -- the caret ∧ -- under a character appropriate to the error.

If the offending line is a line read from the terminal -- as either desk calculator or evaluated input -- then no change is made to the SI or any other internal APL construct: APL merely discards the line and reprompts for the appropriate input.

Suspension

However, if the line in error is a line from the body of a function, then execution of this function call is temporarily suspended: lines for evaluation cease to be drawn from the function body; instead, APL reverts to reading lines from the terminal.

Following a suspension, arbitrary APL lines can be input; they execute completely normally, except that they now execute in the environment prevailing at the instant of suspension. Therefore, all objects -- both local and global -- that were available to the now-suspended function just prior to its suspension are now equally available to the user -- to be displayed and/or changed. Since the full power of APL is available for manipulating these objects, APL is truly its own debugging language.

Implicit Results

Of course, any implicit results produced by evaluation of APL code prior to the error cannot and will not be undone by APL. Note that such implicit results include -- but are not limited to -- I/O activity, assignments to variables, pseudo-variables, and system variables, and erasure and creation of objects via system functions.

Syntax and Context Errors

Unlike many other APL implementations, Multics APL detects and reports all true syntax errors <u>before</u> beginning evaluation of a terminal input line. Furthermore, Multics APL also detects and reports all such errors occurring anywhere in a function just <u>before</u> beginning execution of the function.

Therefore, any error whose report is *SYNTAX ERROR* has been found <u>before</u> <u>any</u> evaluation of its containing entity has taken place.

However, since function definitions are subject to dynamic change -- under program control, APL cannot know in advance if the syntax of a function call is correct -- that is, consistent with the function syntax definition. Therefore, errors of this sort -- which are named *CONTEXT ERROR*s in Multics APL -- can only be detected at the time of the function call: the line containing the function call must be partially evaluated before a *CONTEXT ERROR* can be found and reported. (Context errors are so named because whether or not a statement contains such an error is dependent upon the context of the function call at the time of the function call -- it is dependent solely upon the syntax definition of the called function as currently defined at the instant of the function call. Therefore, such an error is not a true syntax error -- as it is not intrinsically an error -- but rather is a context-dependent error, or simply a context error.)

Note that *SYNTAX ERROR*s are the <u>only</u> runtime errors in Multics APL that are detected and reported before evaluation begins.

STOP PSEUDO-VARIABLES *S*∆name_of_fn

A <u>stop</u> <u>pseudo-variable</u> is a pseudo-variable whose name is composed of the string *S*∆ immediately followed by the name of a function, and which can be used -- with certain restrictions -- just as can any normal variable, except that its proper use produces a side effect -- an implicit result for which this construct was designed -- which stops the execution of its namesake function.

With the sole and crucial exception of this implicit result, the properties, behavior, and rules governing the use of stop pseudo-variables exactly match those of trace pseudo-variables.

Execution of the namesake function of a stop pseudo-variable is stopped just before APL begins to execute any line of that function whose line number is found in that stop pseudo-variable. Before executing each function line, APL uses the membership operator to find out if that function line number is specified in the appropriate stop pseudo-variable: n∊*S*∆name_of_fn, where n is the line number of the function line about to be executed, and *S*∆name_of_fn is the appropriate stop pseudo-variable. If this test yields a ¯1 -- "true" -- then the function execution is suspended, and APL prints: name_of_fn[n]. Otherwise, execution proceeds normally.

Like trace, stop pseudo-variables are used almost exclusively for exploring and/or diagnosing complex or allegedly errant functions. They have little -- if any -- practical use outside of these contexts.

Trace and stop pseudo-variables can be used singly, or in any combination, as appropriate. They do not in any way interact or interfere with each other.

LOCKED FUNCTIONS

When APL detects an error in a locked function, it reports the error just as it does those in unlocked functions, with an important exception: the offending source line is not printed. However, the error name, the name of the errant function, and the line number of the offending line are all printed as usual. Suspension proceeds completely normally.

ATTN may be used -- just as with unlocked functions -- to interrupt execution of any locked function. The function call is suspended completely normally. However, as with error reporting, APL does not print the source line of any locked function whose execution is interrupted in mid-line by a strong interrupt.

Stop and trace pseudo-variables cannot be used with locked functions: any attempt to create one yields an error.

Apart from the restrictions and exceptions previously mentioned, locked functions behave exactly as do unlocked functions.

HALTED FUNCTION CALLS

A *halted function call* is any function call that has stopped executing due to any of the above causes -- an interrupt, error, or stop pseudo-variable. A function call is not halted if it has stopped executing due to normal function return, nor if it is waiting return from one of the following, which is in execution: a function call, ⎕ or ⍞ input or execute ⍎. However, if the function call is awaiting return from something which is itself halted -- a function call, ⎕ input, or ⍎ -- then the function call is halted.

Suspended Function Calls

A *suspended function call* is a halted function call that is not awaiting return from anything.

The execution environment is unchanged by a suspension. That is, the values of local and global variables, of system functions and local and global system variables, of trace and stop pseudo-variables, and of labels, and the definitions of local and global functions, and of groups, are completely unaffected by the fact that the function call that was in execution at the current level has become suspended. In fact, the *only* change is that instead of reading diamond lines to be executed from the function definition, APL reverts to reading them from the user's terminal. Thus, the user "sees" the same execution environment that the now suspended function call executed in just before suspension.

Pendent Function Calls

A *pendent function call* is a function call that is awaiting return from one of the following: a function call, ⎕ input, or ⍎. A pendent function call may or may not be halted: the execution status of its "callee" is irrelevant.

A pendent function call is *not* "in execution".

A given function call always falls into exactly one of the three mutually exclusive categories: in execution, suspended, or pendent.

A pendent function call has a special property: it contains a partially evaluated *pendent statement* containing the call which rendered that function call pendent.

Exploring the SI

APL has several facilities for listing the contents of the SI. The most commonly used are the two system commands)*SI* and)*SIV* and the system function ⎕*SI*. Less useful -- except in special applications -- are the I-beam functions *I*27 and *I*26, and the system function ⎕*LC*.

THE)*SI* SYSTEM COMMAND

The)*SI* system command causes APL to print out a listing -- with one line per stack frame -- of the names of the entities in whose behalf the stack frames exist, together with -- in the case of function calls -- a bracketed line number that indicates: for pendent function calls, the function line on which the pendency exists; or, for suspended function calls that were interrupted between function lines, the function line that was to be executed next; or, for suspended function calls that were interrupted in mid-line via a strong interrupt, the function line that was interrupted and only partially evaluated.

So, for a stack frame created by a function call, the)*SI* list entry appears as: name_of_fn[n]. Similarly, the)*SI* entry of a request for evaluated input is simply a quad ⎕, and that of a call to execute is a hydrant ⍎.

To help distinguish between pendent and non-pendent -- that is, suspended or in execution -- function calls, APL flags every)*SI* entry of a non-pendent function call with a star *. (Note that since execute ⍎ and evaluated input requests ⎕ cannot be suspended -- they can be either in execution or pendent -- no ambiguity can exist as to what state a given such call is in. Therefore, no flagging of)*SI* entries for such calls is necessary.)

THE)*SIV* SYSTEM COMMAND

The)*SIV* system command produces the same listing that)*SIV* does, except that each function call entry is followed -- on the same line -- by a list of all local identifiers localized by that function call.

THE ⎕*SI* SYSTEM FUNCTION

The ⎕*SI* system function returns -- as an explicit result -- a character matrix representation of that which)*SI* would print. That is, the result of ⎕*SI* -- if displayed immediately -- would match the printed output produced by)*SI*.

*I*27, *I*26 AND ⎕*LC*

*I*27 and ⎕*LC* are identical, each returning an integer vector explicit result whose elements correspond one-to-one with)*SI* entries as follows: if the entry is ⍎ or ⎕, then the corresponding element in the result of *I*27 and ⎕*LC* is a zero; if the entry is a function call, then the corresponding element in the result of *I*27 and ⎕*LC* is the line number that is bracketed in the)*SI* entry.

*I*26 returns an integer scalar explicit result that is the first element of *I*27 and ⎕*LC*, or zero if they are empty.

Clearing the SI

It is usually undesirable to leave suspensions -- and any associated pendencies -- in the SI any longer than is necessary to explore and/or diagnose problems in the functions involved in the suspension. Suspensions may require large amounts of storage, especially if local objects are large, or if pendent function or execute calls are nested very deeply; furthermore, unneeded)*SI* entries clutter and confuse the)*SI* listing.

Eliminating, or clearing unneeded suspensions is accomplished by the escape → mechanism.

THE ESCAPE →

The escape → -- a branch symbol with no argument -- causes APL to unwind exactly one suspension: the most recently created one. It clears the SI back to the point of the previous suspension, undoing all localization and masking caused by the most recently suspended function call, and by all of its pendent function calls.

The)*RESET* system command on other system can be mimicked by entering -- one at a time -- as many escapes as there are suspensions.

Restarting a Suspended Function Call

Following the suspension of a function call, the branch pseudo-operator can be used to restart function execution on any line of the most recently suspended function.

Of course, if execution termination was caused by an error, then this is generally useful only after suitable remedies have been effected to prevent recurrence of the error. Such action might include manual entry and execution of appropriate APL statements, and/or corrective editing of the suspended function itself.

The user must also be careful to restart on the correct line -- whether the line before, during, or after which the interruption occurred needs to be executed depends upon the intricacies of the function and upon precisely how, why, and where the interruption occurred.

Function line labels of the suspended function call are defined and accessible, so they can be used -- if convenient -- in restarting that function call.

As always in APL, it is possible to branch only to a line of the topmost function call in the SI -- that is, of the most recently invoked function call.

<u>SI</u> <u>Damage</u>

<u>SI</u> <u>damage</u> occurs when the environment is changed in such a way that it becomes inconsistent with the SI.

For example, the SI is damaged if a function is erased that has pendent or suspended function calls in the SI. Similarly, SI damage occurs if the header of a function with a suspended call is changed via the function editor.

SI damage <u>cannot</u> be caused by any APL function or system function. SI damage can <u>only</u> be caused by: improper editing -- via the APL function editor -- of functions with suspended function calls in the SI; or, erasure -- via the APL)*ERASE* or)*COPY* system commands -- of global functions with suspended or pendent function calls in the SI; or, errant or malicious external functions, although APL cannot -- and will not -- detect or report such damage.

APL notifies the user that SI damage has occurred by printing the message: *SI DAMAGE*.

SI damage is irreversible. However, although sometimes quite complex, it is always possible to recover from SI damage, resuming and successfully completing the original -- or even modified -- computations and function call sequence.

EFFECTS

SI damage does not cause APL to make any further change to the environment. All local and global objects -- except the erased or improperly modified function -- and all localizations and masking, remain unchanged in any respect, regardless of the cause of the SI damage. (Significantly, APL correctly remembers which is the result identifier -- if any -- and this returns completely normally any value assigned to it, when and if the function call returns.)

However, APL prevents further execution in the damaged function call: any attempt to restart a damaged function call, whether by means of a branch into a damaged suspended function call - via →*N*, where *N*>*0* -- or by a return to a damaged pendent function call, fails, yielding an *SI ERROR*.

Note that a function return -- via →*N*, where *N*≤*0* -- or an escape →, is completely acceptable, and proceeds normally.

SI damage is recorded in the)*SI* and)*SIV* listings, and in the explicit result of ⎕*SI* in the following way: the name_of_fn[n] portion of each damaged function call entry is replaced with six spaces. Any stars * that flag non-pendent function calls remain unchanged, and local identifier lists produced by)*SIV* remain unchanged.

Note that SI damage extends to all previous calls to the erased or destructively modified function : every existing suspended and pendent function call to that function is damaged, not merely that most recent such call.

EDITING HALTED FUNCTIONS

Functions that have pendent function calls cannot be edited, because at least one of its lines is only partially evaluated, and is therefore saved in the SI. No meaningful association could be established between this saved information and an edited function definition: APL could not know where or how to resume execution of these partially evaluated lines.

Consequently, if it is necessary to edit a function with pendent calls, these pendencies must first be cleared from the SI using escapes →.

However, functions that have only suspended function calls can be edited and restarted successfully, with the following restrictions:

- The header cannot be changed in any way.
- Label names cannot be changed.
- Labels cannot be added or deleted.

Any failure to observe these rules elicits a warning from APL when the user attempts to leave the editor. A second request to leave the editor is honored, but the violation of the above rules causes SI damage to all calls to the just modified function.

If the above restrictions are observed, no warning or SI damage occurs when leaving the editor. Furthermore, the new function definition takes effect immediately, for both future and suspended function calls.

Labels are automatically redefined -- if necessary -- just before leaving the function editor. This is necessary if the line number of the line on which the label appears differs between the old and the new function definitions. (Note, however, that non-branching references to labels made prior to editing cannot and will not be corrected by APL, and may therefore be invalid. For example: *L*2:*A*←3+5×*L*2; after editing, *L*2 is redefined correctly, but *A* reflects the old *L*2 definition, and therefore causes incorrect operation of the function.)

EDITING A FUNCTION

APL provides a function editing capability. Adding, deleting, or retyping lines can be performed on complete lines. Detailed editing of individual lines, such as deleting, inserting, and changing individual characters on a line, can also be performed. APL must be in definition mode to perform function editing.

The various function editing techniques are described below. Assume the following function has already been defined as:

```
     ∇X←AVG
[1] SUM←(+/NUM)
[2] ∇
```

This function may be edited as follows:

1. To add a new line, type ∇*FUNCTION_NAME*

```
          ∇AVG
     [2]
```

   Notice that APL is now in definition mode, waiting for you to enter line 2, the next available line in the function. Type in the line and terminate definition mode:

```
     [2] X←SUM÷ρNUM
     [3] ∇
```

   or

```
     [2] X←÷ρNUM∇
```

2. To list a function and return to immediate mode, type:

```
          ∇AVG[⎕]∇
```

   To list a function and remain in definition mode, type:

```
          ∇AVG[⎕]
          ∇X←AVG
     [1] SUM←(+/NUM)
     [2] X←SUM÷ρNUM
          ∇
     [3]
```

   To list a function when the interpreter is already in definition mode type:

```
     [3] [⎕]
          ∇X←AVG
     [1] SUM←(+/NUM)
     [2] X←SUM÷ρNUM
          ∇
     [3]
```

   To list a single line, type:

```
     [3] [2⎕]
     [2] X←SUM÷ρNUM
     [2]
```

   To list all of the lines from line *K* on, type [⎕*K*]. For example:

```
     [2] [⎕1]
     [1] SUM←(+/NUM)
```

```
[2] X←SUM÷ρNUM
[3]
```

3. To delete a line, type, in square brackets, a delta (∆), followed by the line number that is to be deleted.

```
[3] [∆2]
[3]
```

4. To replace a complete line, type, in square brackets, the number of the line that is to be replaced. When prompted by the appropriate line number, type in the new line.

```
[2] [1]
[1] X←(+/NUM))÷ρ
[2]
```

   A line can also be replaced by typing the line number to be replaced in brackets, followed immediately by the new text.

5. To insert a line between the other lines, use fractional line numbers. For example, to insert a line between the header (considered to be line 0) and line 1, type some fraction between 0 and 1 (say 0.5) in brackets. When APL prints the line number [0.5], type in the new line.

```
[2] [0.5]
[0.5] 'AVERAGE'
[0.6] ∇
```

   Lines may also be inserted by typing the fractional line number in brackets, followed immediately by the text for that line.

   When a function is closed, APL automatically resequences the function line numbers. The edited function now looks like this:

```
 ∆AVG[⎕]∇
 ∇X←AVG
[1] 'AVERAGE'
[2] X←(+/NUM)÷ρ
 ∇←∇
```

EDITING A LINE

Characters within a function line (including line 0, the header line) can be replaced, deleted, and inserted by typing [*M*⎕*N*]; where *M* is the line number to be edited, and *N* is the approximate position in the line for editing to begin. After [*M*⎕*N*] is typed, the line to be edited is displayed, the carriage is returned, and *N* positions are spaced over. Editing of the line may then be performed.

1. To delete any number of characters, type in a / beneath the characters to be deleted:

```
[3] [2⎕15]
[2] X←(+/NUM))÷ρ
               /
[2] X←(+/NUM)÷ρ
              NM
[3] [2⎕]
[2] X←(+/NUM)÷ρNM
[2]
```

   This line is now displayed again, and the carriage waits at the end of the line so that additional characters may be added to the line.

2. To insert a character or characters between two adjacent characters j and k, type a digit below k to indicate the number of characters which are to be inserted to the left of k:

```
[3] [3⎕8]
[2] X←(+/NUM)÷ρNM
               1
```

   The line is displayed with the appropriate number of blanks inserted. The carriage then waits at the leftmost blank for insertion:

```
[2] X←(+/NUM)÷ρN M
                U
```

   The letters *A* through *Z* can be used in place of the digit to insert 5 blanks for an *A*, 10 blanks for a *B*, etc.

3. To replace a character with another character or characters, combine the two above methods and type /*N*, where *N* is a digit or letter to indicate the number of characters to replace the character:

```
[3] [1⎕13]
[1] 'AVERAGE'
           /4
[1] 'AVERAG    '
           E =
```

4. To add to the end of a line, the editing command [*M*⎕0] can be used. Line *M* will be printed, and new material may be added. This is convenient for adding comments to the end of a line.

```
      [2] [0⎕0]
           X←AVG
                NUM
      [1] [1⎕0]
      [1] 'AVERAGE = '
                   ⍝COMMENT
```

The edited function now looks like:

```
      [1] [⎕]
       ∇X←AVG NUM
      [1] 'AVERAGE = '⍝COMMENT
      [2] X←(+/NUM)÷⍴NUM
           ∇
      [3] ∇
```

SECTION 5

SYSTEM COMMANDS

System commands are special lines typed by the user to adjust or control the operation of APL. They are distinguished from expressions by always beginning with a right parenthesis -- no expression could ever begin that way.

The right parenthesis is followed by the name of the particular system command, which is then followed by arguments, separated by spaces. The arguments required by each command vary, and are discussed under the individual command descriptions below.

System commands can be issued whenever APL is awaiting desk calculator or evaluated input. They cannot be issued from a function call, nor are they recognized while APL is awaiting character input.

Most system commands are innocuous enough to be accepted while in the function editor. However, certain system commands are explicitly disallowed while in the editor -- any attempt to issue these commands is rejected:

>)*CLEAR*,)*CONTINUE*,)*COPY*,)*ERASE*,)*LOAD*,
>)*OFF*,)*QUIT*,)*SAVE*, and)*V1COPY*.

A system command is performed as soon as it is issued; then APL requests again the input it was awaiting before encountering the command -- unless the action of the command is such that the input is no longer needed.

ENVIRONMENT PARAMETERS

APL is partially controlled by a set of environment parameters. These include the workspace parameters, and a set of session parameters.

APL offers system commands and/or system variables to set each parameter, or to find out its current setting.

For system commands, a parameter is set by issuing the command name, followed by the desired setting, as an argument. The command name alone causes APL to print the current setting, without change.

For system variables -- discussed more in a later section -- a parameter is set by assigning the desired setting to it, just as with a regular variable.

Workspace Parameters

Every workspace contains a set of dynamic, user-definable workspace parameters, used for specific, fixed purposes by APL. These are: the index origin, the number of digits of printing precision, the comparison tolerance, the integer tolerance, the latent expression, the workspace identification, and the random number seed. Most of these workspace parameters have associated with them a system command and a system variable to set and/or find its value.

The workspace parameters are also affected by the *)CLEAR* and *)LOAD* system commands. The *)CLEAR* command resets all parameters to their default values, while the *)LOAD* command sets them to the values recorded in the saved workspace.

Session Parameters

Every APL session has an associated set of dynamic, user-definable session parameters, each of which is used by APL for specific, fixed purposes. The session parameters are: the page width, the horizontal tab setting, the error mode, and the compatibility mode.

Each of these session parameters has a system command to set or inquire about its value.

None of these parameters are affected by changing or clearing the active workspace.

The)*ORIGIN* System Commands

The)*ORIGIN* system command takes as its argument either the constant 0 or the constant 1. The command establishes its argument as the new value of the index origin and types out the old value.

The index origin of a clear workspace is 1 by default.

The index origin of a workspace is saved and restored by the)*SAVE* and)*LOAD* system commands.

The index origin determines whether numbers from 0 to *N*-1 or from 1 to *N* are used to number coordinates and elements for various operators:

```
A[A;A;...;A]            interpretation of subscripts;
⍳S          V⍳A         result of index operators;
⍋A          ⍒A          result of grade operators;
?A          S?S         result of roll and deal;
⍋[S]A       ⍒[S]A       interpretation of coordinate
⌽[S]A       V⌽[S]A      numbers.
V/[S]A      V\[S]A
⊙/[S]A
```

The)*WIDTH* System Command

The)*WIDTH* system command takes as its argument an integer constant from 30 to 130. The integer supplied is established as the new page width to be observed by the APL output routines, and the previous value of the page width is typed out.

The page width of a workspace determines the maximum number of characters that the APL output routines place on a line before deciding the line is full and overflowing to the next line.

Page width is a session parameter whose default is taken to be the line length in use when APL is entered.

The)*DIGITS* System Command

The)*DIGITS* system command takes as its argument an integer constant from 1 to 19. The integer supplied is established as the new number of digits of printing precision; the old value is typed out.

The digits setting of a clear workspace is 10.

The digits setting is saved and restored by the)*SAVE* and)*LOAD* system commands.

The digits setting determines only how values are formatted for printing. It does not affect the stored values themselves, nor does it affect their calculation. All values in Multics APL are calculated to 63 bits of precision (approximately 19 decimal digits). As a value is printed, it is rounded to the desired number of significant digits when it is converted to printable characters.

The)*ERRS* System Command

The)*ERRS* system command establishes whether APL error messages are printed in their long or brief form. The long form gives additional information on the cause of the error, while the brief form is very short, giving only the error name.

The argument is the mode that the errors are to be printed in: that is, *LONG* or *BRIEF*. The error mode is a session parameter.

The)*TABS* System Command

The)*TABS* system command takes as its argument an integer constant from 0 to 130. The argument is established as the new tab setting to be used to speed up output. For properly formatted output, the physical or electronic tab stops must be set at uniform intervals that match the)*TABS* setting.

)*TABS* 0 disables use of tabs in output.

The tabs setting is a session parameter.

The)*CHECK* System Command

The)*CHECK* system command accepts one argument; *ON* or *OFF*. This command establishes whether or not APL checks and reports usage of APL constructs whose definitions have been changed incompatibly. APL reports such usage as a *COMPATIBILITY ERROR* only when the compatibility mode is turned on via)*CHECK ON*.

)*CHECK OFF* disables this checking.

The compatibility mode is a session parameter.

The)*HUH* System Command

The)*HUH* system command is used to print the long form of the most recent error message. It is used to obtain more information about the problem when running in)*ERRORS BRIEF* mode.

SYMBOL TABLE

The symbol table is an area of the workspace set aside for remembering names and the objects to which they refer. A number of system commands exist for inspecting and manipulating the symbol table.

The)*VARS* System Command

The)*VARS* system command is used to print out a list of the names of all currently accessible -- non-masked -- variables in the workspace, both local and global. The list is produced in alphabetic order, and is printed in as many columns across the page as the page width allows.

The)*VARS* system command can be issued without any arguments, in which case it lists all variables. It can also be issued with a name as its argument, in which case it lists only names that match or follow the argument name in alphabetic order.

The name supplied as the argument need not name an object in the workspace: it is used only in an alphabetic comparison to decide which names to print.

The)*FNS* System Command

The)*FNS* system command is used to print a list of the names of all currently accessible -- non-masked -- global and local functions defined in the active workspace. Like)*VARS*, the)*FNS* command prints its list in alphabetic order, and a valid APL name -- possible representing a nonexistent object -- is accepted as an optional argument indicating where to begin the list.

The)*GROUP* System Command

The)*GROUP* system command is used to gather objects into a group, to append more objects to a group, or to disband a group.

The first argument of the)*GROUP* command must be the name of the group upon which the command is to operate. If the purpose of the)*GROUP* command is to create the group, then the group need not yet exist at the time the command is issued; otherwise, it is an error if the first argument is not the name of an existing group.

If no further arguments beyond the group name are supplied, then)*GROUP* disbands the group. Disbanding a group has no effect upon its members; the only consequence is that they are no longer considered to be in a group. This should be carefully contrasted with erasing a group, which also erases its members.

If some names of objects follow the group name argument, then)*GROUP* establishes a group of the designated name having the indicated members. Any object can be a member of a group: a variable, a function, or a group.

Only one object cannot be made a member of a group: the group itself. The inclusion of the group's own name in the list of members has a special meaning to the)*GROUP* command: all the previous members of the group are to be retained in the new group, along with the new members. This appends new members to an existing group.

The)*GRP* System Command

The)*GRP* system command lists the names of the members of a group. It takes as its argument the name of a group.

The)*GRPS* System Command

The)*GRPS* system command lists the names of all groups defined in the active workspace. Like the)*VARS* and)*FNS* commands, the)*GRPS* command prints its list in alphabetic order, and accepts a starting name as an optional argument.

The)*ERASE* System Command

The)*ERASE* command is used to delete objects from the active workspace. Its arguments are any number of names of objects to be deleted. The objects can be global variables, function, or groups.

When an object is erased, it is completely removed from the workspace and discarded. No record of its previous existence remains. Its name and the storage it occupied become available for other uses.

Local objects cannot be erased by the)*ERASE* command. However, they are automatically erased when the function to which they are local returns. Also ⎕*EX* erases local objects.

If an argument to)*ERASE* is the name of a group, then the group is disbanded, and all members of that group are erased. However, groups that are members of the erased group are disbanded, but their members in turn are not affected. Thus, erasure of groups containing groups is not fully recursive; only the direct members of the erased group are erased.

The)*SYMBOLS* System Command

The Multics APL)*SYMBOLS* command does not accept arguments, and causes no change to the workspace environment. It simply reports the total number of symbol table entries currently used.

The)*SI* System Command

The)*SI* system command is used to inspect the state indicator of the APL processor. The state indicator is an area of the workspace set aside to record the state of functions currently invoked.

The meaning and workings of the state indicator are fully explained in Section 4, but, briefly, the state indicator acts as a stack. As one APL statement invokes another (either as a function or as evaluated input), the information pertaining to the partially evaluated invoking line is stacked in the state indicator. The APL processor is then free to evaluate the invoked lines, knowing that when it finishes it can return to complete the evaluation of the invoking line by restoring the saved state of evaluation from the state indicator stack. Since the invoked statements can further invoke other statements, many partial evaluations may need to stack successively in the state indicator. As their respective evaluations complete, the stack is popped back in parallel.

A statement whose execution is stopped temporarily because the processor must execute another statement (that it invokes) is said to be pendent. Thus, the use of the state indicator discussed so far is to remember all pendent statements.

Another item of information remembered in the state indicator is function suspensions. When the execution of a function produces an error report, statements cease to be drawn from the function definition and are instead read from the user's terminal until the function is explicitly restarted. During this interval, the function is said to be suspended. An entry in the state indicator for a suspension differs from that for a pendent statement in that no partially evaluated statement is remembered, and also in that a suspension marks a place where the user obtained control and was able to type new statements.

In the listing printed by the)*SI* command there is one line per entry on the state indicator stack. The stack is printed in the order of most recent item first to least recent last; thus, the first line printed corresponds to the most recent entry made into the state indicator. Each line shows the name of the function in execution (or the symbol ⎕ if the entry refers to an evaluated input line), the statement number upon which execution resumes (the pendent statement itself for pendent entries; the statement following the error for suspended entries), and finally an asterisk if the entry represents a suspension (lines without an asterisk correspond to pendent entries).

The)*SIV* System Command

The)*SIV* system command performs the same function as the)*SI* system command, except that each line of the display shows, in addition to everything shown for the)*SI* command, all identifiers local to the particular invocation.

Since a reference to an identifier is satisfied by the most recently created object of that name, the referent of any given identifier is easily found by scanning the)*SIV* list downward. The first instance of the sought identifier is the satisfying referent. If the identifier is not found anywhere in the)*SIV* list, then the reference is satisfied by a global object. Local and global identifier referencing is treated more fully in Section 4.

WORKSPACE MANAGEMENT

One of the most important features of APL is its ability to save the complete contents of a workspace and then take it up again later. Work can then be continued as if there had been no interruption. On Multics, workspaces are saved as segments -- or, if necessary, as multisegment files -- anywhere in the storage system hierarchy. Normal Multics quota and access conventions govern the storage of saved workspaces.

When a workspace is saved, everything necessary to resume the session in progress is remembered. The values of all variables, both local and global, the definitions of all functions and groups, everything in the state indicator, and the settings of all workspace parameters are saved.

When a workspace is to be taken up again, the user has a choice of how much of the saved workspace to recall. He can copy individual global variables or functions or groups via the *)COPY* system command with specific objects named; or he can copy all global objects but not the state of execution or workspace parameter via the *)COPY* command with no objects named; or he can recall the entire workspace via the *)LOAD* command.

Objects can be moved from workspace to workspace or duplicated in several workspaces.

Workspace Identification

Each workspace has associated with it a workspace identification, which is the absolute or relative pathname of the workspace.

The active workspace also has a workspace identification: that of the workspace most recently loaded. If no workspace has yet been loaded, the active workspace has the identification *CLEAR WS*. Note that a workspace id of *CLEAR WS* does not mean that the active workspace is clear; it only means that it started out as a clear workspace.

When a *)LOAD* command is issued, APL locates the saved workspace in the Multics hierarchy, loads a copy of its contents into the active workspace -- discarding the old contents of the active workspace -- then replaces the current wsid -- workspace identification -- with the pathname given to *)LOAD*, and finally prints a message indicating the date and time at which the workspace was last *)SAVED*.

When a *)SAVE* command is issued with no pathname as argument, the active workspace is stored according to its current wsid. If a pathname argument is given, then the wsid is replaced by the new pathname, and the active workspace is saved at the new place in the Multics hierarchy.

It is an error to issue a *)SAVE* command with no pathname given when the workspace identification is *CLEAR WS*.

The wsid can also be inspected or changed at any time with the *)WSID* system command. No commands other than *)CLEAR*, *)SAVE*, *)LOAD*, and *)WSID* affect or concern themselves with the active wsid.

The user is cautioned that changing the working directory when the workspace identification is only a relative pathname can change the meaning of that pathname -- and a succeeding *)SAVE* command with no argument may not necessarily refer to the previously loaded workspace.

Passwords

It is possible to associate a password with a saved workspace. When a workspace has been saved with a password, APL prevents a load or copy from it unless the password can be supplied. The password is also required to delete a saved workspace.

Cautious users should note, however, that nothing prevents another user from supplying a program of his own construction to search for interesting things in your saved workspaces.

The mechanics of supplying passwords are as follows. Passwords are accepted by the *)SAVE*, *)LOAD*, *)COPY*, *)PCOPY* and *)DROP* system commands. To indicate that a password is to supplied, a colon follows the last character of the pathname constituting the workspace identification in the command line. Then, the user has the choice of supplying the password in one of two ways: he can enter it into the command line, immediately following the colon; or, he can enter -- in its place -- a quote quad ⍞, indicate that APL should use the get_password_ subroutine to ensure that the password is not visible to other users. Entering a ⍞ in place of the password results in the prompt *PASSWORD:* and an opportunity to enter the password with either the printer turned off, or on top of a pre-printed mask of random characters, depending upon the terminal.

The password may be from zero to eight characters in length, with zero characters indicating that no password is to be used.

The remainder of the command line is typed normally. After receiving the correct password, APL proceeds normally.

If a workspace saved with a password is addressed by a *)LOAD*, *)COPY*, *)PCOPY*, or *)DROP* command without a password or with an incorrect password, then the command is ignored.

If a *)SAVE* command is used without a colon, then the workspace is saved with its current password -- if any. To remove or change the password of a workspace, the *)SAVE* command must be issued with a colon. A password of zero characters is considered the same as no password.

A clear workspace has no password.

The *)CLEAR* System Command

The *)CLEAR* system command is used to clear the active workspace. When the *)CLEAR* command is issued, the APL processor types *CLEAR¨WS*, erases all objects, discards the state indicator and all local variables; resets the index origin, digits, fuzz, password, latent expression, and workspace identification to their default values, and reads the calendar clock to obtain a new seed for the random number generator.

The *)LIB*, *)LIBD* System Command

The *)LIB* and *)LIBD* commands each print a list of the pathname of all saved APL workspaces in the user's hierarchy. *)LIB* prints a horizontal list while *)LIBD* prints a vertical list.

)LIBD additionally lists -- for each such workspace -- the date-time saved and date-time used data.

The *)SAVE* System Command

)SAVE is used to save a copy of the active workspace. Everything in the workspace is saved. Workspaces become Multics segments or multisegment files.

)SAVE can be issued in three forms: the command alone,

```
)SAVE
```

or with a workspace identification (i.e., pathname),

```
    )SAVE wsid
```

or with a workspace identification and a colon,

```
    )SAVE wsid :
```

In the first form -- the command alone, the workspace is saved under its current identification and with its current password. See the preceding two sections for a description of workspace identifications and passwords. This form of)*SAVE* is invalid when the workspace identification is set to *CLEAR WS*.

The response to this command is the current date and time, followed by the workspace identification as a reminder.

In the second form, the active workspace is saved under the new identification (i.e., at a new place in the storage system hierarchy), but with the current password if any. The current workspace identification is replaced by the wsid argument to)*SAVE*. The response to this form of the command is the current date and time.

In the final form of)*SAVE*, APL requires a password. Then the active workspace is saved under the new identification with the new password. An empty password is the same as no password. The current workspace identification and password are both changed to those given in the command line. Again, the response to this command is the date and time.

The)*SAVE* command does not alter the current workspace in any way, except that its wsid and password may change. Following the)*SAVE*, computations can resume in the current workspace.

If any Multics errors occur in performing the save, such as record quota or access violations, they are reported and the current workspace remains unchanged.

The)*LOAD* System Command

The)*LOAD* system command takes as its argument a workspace identification, (i.e., pathname), optionally followed by a colon. If the colon is supplied, APL requires a password. If the password supplied does not match that of the saved workspace whose identification is given in the)*LOAD* command, then the error is reported and the current workspace remains unchanged. Otherwise, the current active workspace contents are discarded and replaced by a copy of the saved workspace.

The response to the)*LOAD* system command is saved followed by the date and time that the workspace was saved.

The saved workspace itself is not altered by the)*LOAD* command.

The)*COPY* System Command

The)*COPY* system command copies functions, groups, and global variables from a saved workspace <u>into</u> the active workspace. The active workspace remains unchanged $\overline{\text{except}}$ for the addition of the new object or objects.

The *)COPY* system command can be issued to two forms. The first is:

```
)COPY wsid
```

where wsid is the pathname of a saved workspace. If the workspace was saved with a password, then it must be given. This form of the command places copies of all global functions, groups, and variables contained in the saved workspace into the active workspace. If any object of the same name as an object to be copied already exists in the active workspace, it is erased and replaced by the new object. All other objects in the active workspace remain unchanged, as do the workspace parameters and the state indicator -- unless *SI DAMAGE* occurs.

In the second form of the *)COPY* command, a list of specific objects is mentioned, as:

```
)COPY wsid object object...
```

where, as before, wsid is the pathname of a saved workspace, and each 'object' is the name of a global function, group, or variable in the saved workspace.

The function of this form of the *)COPY* command is to copy only the objects mentioned from the saved workspace into the active workspace. If any object is a group, however, all of its members are copied as well.

As in the other form of *)COPY*, naming conflicts between copied objects and existing objects are resolved by erasing the existing objects.

If a specified object does not exist in the saved workspace, an error report gives the message that it was not copied.

In any case, there is no change to the saved workspace. The response to the *)COPY* command is the date and time the donor workspace was saved.

The)*PCOPY* System Command

The)*PCOPY* (protected copy) system command behaves exactly like the)*COPY* system command with the exception of its treatment of naming conflicts. With the protected copy command, when an object to be copied has the same name as an existing object, the existing object remains unchanged, and the saved object is simply not copied. The names of any objects not copied are reported.

Like the)*COPY* command, the)*PCOPY* command can copy either specified objects or all global objects out of a saved workspace. The normal response to the command is the date and time the workspace was saved. There is no change made to the saved workspace.

The)*CONTINUE* System Command

The)*CONTINUE* system command provides no new capability to the APL system, but is simply a convenient way to terminate an APL session that must be resumed again later. The)*CONTINUE* command behaves identically to the sequence

```
)SAVE CONTINUE
)OFF
```

of commands. That is, the current workspace is saved under the name *CONTINUE*, and the APL session is terminated. Later, the command:

```
)LOAD CONTINUE
```

can be used to pick up the work again as if there had been no interruption.

The)*WSID* System Command

The)*WSID* system command is used to inspect or change the current workspace identification. If)*WSID* is typed with no arguments, as:

```
)WSID
```

then the current workspace identification is printed out.

)*WSID* can also set the current workspace identification. In this case, the user types:

```
)WSID wsid
```

where "wsid" is any absolute or relative pathname. The former identification is typed in reply.

The purpose of setting the workspace identification is to allow a later)*SAVE* command given without an identification to save the workspace in the desired place. Beyond)*WSID*, the only commands that affect or concern themselves with the workspace identification are)*CLEAR*,)*SAVE*, and)*LOAD*.

The)*DROP* System Command

The)*DROP* system command is used to delete the saved copy of a workspace. The form of the command is:

```
)DROP wsid
```

where "wsid" is the pathname of the saved workspace to be deleted. A password is not required to delete a saved workspace.

)*DROP* has no effect on the active workspace.

Version 1 APL Workspaces

In addition to the system commands for manipulation of workspaces, analogous commands exist for use with workspaces created by Multics Version 1 APL. These commands are)*V1LIB*,)*V1COPY*,)*V1PCOPY*, and)*V1DROP*. Their syntax and actions are identical to their Version 2 APL counterparts. Thus in order to convert a workspace from Version 1 APL to Version 2, just copy it into the current workspace using)*V1COPY*, then save it using the regular)*SAVE* system command. The Version 1 workspace is not deleted or replaced. To delete it, use the)*V1DROP* system command.

COMMUNICATING WITH MULTICS

In addition to the commands relating to the saving and reloading of workspaces, a number of other commands involve communication between APL and Multics. The)*Q*,)*QUIT*, and)*OFF* system commands are used to exit from APL. The)*PORTS* system command prints the names of other users currently logged in to Multics. The)*R* system command returns the users Multics command line while the *?* system command lists all the system commands. Finally, the)*E* system command provides a means of executing Multics commands without exiting from APL.

The)*Q*,)*QUIT*,)*OFF* System Commands

The)*Q*,)*QUIT*, and)*OFF* system commands are all identical. They cause the APL processor to return to its caller. If APL was invoked as a Multics command, this amounts to a return to Multics command level.

Following a return to Multics, the current workspace is no longer accessible. If the user wishes to save the results of an APL session, a)*SAVE* command must be issued before returning, or else the)*CONTINUE* command should be used to exit instead of)*Q*,)*QUIT*, or)*OFF*.

The)*PORTS* System Command

The)*PORTS* system command prints a list of the Multics users currently logged in. It is implemented as a call on the Multics "who" command. Any arguments typed after the)*PORTS* command are simply passed on to the "who" command, so a certain amount of selectivity is possible. Refer to the description of the "who" command in the Multics Commands and Active Functions manual (Order No.: AG92), for further information.

The)*EXEC*,)*E* System Commands

The)*EXEC* and)*E* system commands are identical. They are used to execute an arbitrary Multics command line from within APL. The entire remainder of the command line following the)*E* is passed unchanged to the Multics command processor for execution.

The user is cautioned that the command line itself has been read by APL; hence, it has undergone the APL rather than the Multics input processing. While the APL and the Multics character sets largely overlap, there are some differences. It is up to the user to anticipate the translations mentioned in Section 2 and compensate for them where necessary. For example, if one types

```
    )E SM MGS M WHY ERROR WHEN ¨123¨ IS TYPED?
```

the actual message transmitted, in ASCII, is

```
    Why error when S" is typed?
```

The)*HELP* System Command

The)*HELP* system command provides an interface to the Multics help facility.)*HELP* is used to print out the on-line documentation available on Multics for both Multics APL and most of the other facilities available on Multics.

If the *)HELP* command is given no arguments, it prints a summary of the info files available on Multics APL. If it is given an argument, the argument is interpreted as the name of a Multics info file. A full Multics pathname may be specified.

The *)MSG* System Command

The *)MSG* system command provides an interface to the Multics send_message facility, allowing APL users to send interactive messages to other Multics users. It takes one or more arguments, the first of which is the Multics user to send the message to (for example *⌈MITH.|ULTICS*). The remainder of the line is interpreted as the text of the message, and is sent to the specified user. If there is no text given on the request line, the message *ιNPUT:* is printed, and text lines are read from the terminal and sent to the specified user as they are typed. To exit the message sending session, type a line containing only a period (.).

EXTERNAL FUNCTIONS

The Multics APL interpreter permits APL programs to make external calls out to object segments that have been created by other Multics translators, such as PL/I, provided that those object segments obey a specified interface. To the APL program, such a call looks like an ordinary reference to a defined function; the function may accept zero, one, or two arguments, and it may optionally return a result.

The *)DFN*, *)MFN*, *)ZFN* System Commands

The system commands *)DFN*, *)MFN*, and *)ZFN* are used to define external function names. The command *)DFN* is used to declare a dyadic function; i.e., one accepting two arguments. The command *)MFN* is used to declare a monadic function; i.e., one accepting one argument. And the command *)ZFN* is used to declare a zero-adic (niladic) function; i.e., one accepting no arguments. Whether an external function produces a result need not be specified at the time its name is defined; in fact, the same function can at times return and at other times not return a value, as it chooses.

Definition Syntax

The syntax of an external name definition is:

```
)DFN aplname pathname
```

or else:

```
)DFN pathname
```

where *)DFN* can be replaced by *)MFN* or *)ZFN* as appropriate to the function being defined. The first form defines the name "aplname" to be an externally-coded dyadic function. When an APL program makes a function reference to "aplname", the APL interpreter performs a call on the object segment "pathname" with the calling sequence described below. When "pathname" returns, any returned value is considered as the result of "aplname", and execution of the APL program resumes.

The "pathname" may be an absolute or relative pathname, or it may be a reference name, in which case the Multics search rules are used to obtain its referent. The "pathname" may contain both a segment name and an entry name separated by a dollar sign, as "*A$B*", or it may simply contain a segment name, as "*A*", which is considered "*A$A*", i.e., a call to entry point "*A*" in segment "*A*". Note that the dollar sign must be typed as S backspace / (*$*) on Selectric-type terminals.

In the second form of definition, where "aplname" is not specified, it is considered to be the same as the entry point name of "pathname", (for example, "*B*" if pathname were "*A$B*", or "*XYZ*" if pathname were simply "*XYZ*").

Definition Errors

A definition error report can be due to: (1) an invalid character in the function name; (2) an invalid pathname; (3) inability to find the specified pathname/reference name; (4) conflict with existing reference name; (5) a global variable or group already in existence with the proposed function name.

External Functions Cannot Be Edited

External functions cannot be edited in any way by the APL editor. An attempt to open one for editing results in a definition error report. However, external functions can be erased or redefined, and their definitions can be copied from one workspace to another.

External Function Calling Sequence

A procedure, say "f", that is to be called by APL as an external function, must conform to the following calling sequence:

```
f:  procedure (operators_argument);

declare 1 operators_argument aligned,
           2 operands (2) aligned,
             3 value pointer unaligned,
             3 on_stack bit (1) aligned,
           2 operator aligned,
             3 dimension fixed bin,
             3 padding bit (18) unaligned,
             3 op2 fixed bin (8) unaligned,
             3 op1 fixed bin (8) unaligned,
           2 result pointer unaligned,
           2 error_code fixed bin (35),
           2 where_error fixed bin;
```

Each of the fields in operators_argument is described below:

operands(1).value	pointer to the left operand's value bead. If this is null, the operator is monadic.
operands(1).on_stack	equal to "1"b if the operand is on the value stack. See below for a detailed explanation of conventions for using the value stack.
operands(2).value	pointer to the right operand's value bead.
operands(2).on_stack	Similar to operands(1).on_stack flag.
dimension	the dimension along which the operator should operate. This is always the last dimension for external functions.
padding	this field is unused.
op2	this field is unused for external functions.
op1	this field is unused for external functions.
result	(output) pointer to the result value bead. This bead must be on the value stack. See below for a detailed explanation of conventions for using the value stack.

error_code	(output) set to a status code in apl_error_table_ before the operator signals apl_operator_error_. See below for a list of status codes.
where_error	(output) set by error processing in operator to indicate which operand the error applies to. By default the error marker is placed under the external function name. Add one to where_error to cause the error marker to be placed under the left operand. Subtract one from where_error to cause the error marker to be placed under the right operand.

An APL value bead is declared:

```
declare 1 value_bead aligned based (value_ptr),
          2 bead_type unaligned,
            3 operator bit,
            3 symbol bit,
            3 value bit,
            3 function bit,
            3 group bit,
            3 label bit,
            3 shared_variable bit,
            3 lexed_function bit,
          2 data_type unaligned,
            3 list_value bit,
            3 character_value bit,
            3 numeric_value bit,
            3 integral_value bit,
            3 boolean_value bit,
          2 unused bit (5) unaligned,
          2 bead_size bit (18) unaligned,
          2 reference_count fixed bin (29),
          2 total_data_elements
                          fixed bin (21),
          2 rhorho fixed bin,
          2 data_pointer pointer unaligned,
          2 rho fixed bin (21) dim (n refer
                          (value_bead.rhorho));

declare 1 character_data_structure aligned based (data_ptr),
          2 character_datum char (1) unaligned
                          dim (0:data_elements - 1);

declare 1 numeric_datum float bin (63) aligned
                          dimension (0:data_elements - 1)
                          based (data_ptr);
```

where:

string (bead_type) is "001000000"b for a value bead.

string (data_type)	is "01000"b for a character value, "00100"b for a numeric value, "00110"b for a integral numeric value, and "00111"b for a boolean numeric value.
unused	is unused.
bead_size	is not used or set by an operator or external function.
reference_count	is not used or set by an operator or external function.
total_data_elements	is the number of elements in the APL value. This is 1 for a scalar, ρ*V* for a vector *V*, and ρ,*A* for an array *A*.
rhorho	is the "rhorho" (number of dimensions) of the APL value. This is 0 for a scalar, 1 for a vector, and ρρ*A* for an array *A*.
data_pointer	is a pointer to the array of elements, either character or numeric. The data array always immediately follows the value bead in storage.
rho	is the array of dimensions of the value. This is undefined for a scalar, a single-element array for a vector, and a vector of integers for an APL array. (ρ*A* for an array *A*).

Status Codes for Use by External Functions

When an external function reports an error (either in one of its operands, or an error arising from a computation) it assigns one of the following status codes to operators_argument.error_code and returns. This activates the APL error mechanism.

The status codes are:

apl_error_table_$rank prints *RANK ERROR*

apl_error_table_$index prints *INDEX ERROR*

apl_error_table_$length prints *LENGTH ERROR*

apl_error_table_$domain prints *DOMAIN ERROR*

apl_error_table_$system_error
prints *SYSTEM ERROR*

Conventions for Using the Value Stack

The Multics APL interpreter normally passes operands by reference; results of expressions are passed by value, however. The difference is important; operands passed by reference may not be modified in any way, operands passed by value may be overwritten in the course of operation, as an optimization to avoid allocating a temporary work area. The on_stack flag is set by the APL interpreter to indicate whether an operand has been passed by value or by reference. If the on_stack flag equals "1"b, that operand has been passed by value, on the "value stack".

The value stack is a segment (or several segments) used solely to hold results of operations, and intermediate work areas. It is managed with a stack discipline; new storage is allocated at the end, and the pointer to the end of the stack is advanced. The subroutine apl_push_stack_ in the include file apl_push_stack_fcn.incl.pl1 should be called to allocate space on the value stack. It takes one argument, which is the number of words needed.

A typical external function operates as follows:

1. Checks its operands for consistency (make sure both are numeric, or both are character, make sure ranks are compatible, etc).

2. Allocates a value_bead for the result on the value stack.

3. Performs the operation.

4. Pops operands that are on the value stack off of it. Operand 2 (the right operand) is guaranteed to be lower on the stack than operand 1 (the left operand), if both operands are on the stack. Therefore, if operand 2 is on the stack, the result value_bead should overlay operand 2. If operand 1 is on the stack, the result value_bead should overlay it. If neither operand is on the stack, the result value_bead must stay where it is. Now copy (if necessary) the result to its final place on the value stack, resetting ws_info.value_stack_ptr to point to the end of the result.

5. Sets operators_argument.result to point to the result value_bead.

6. Returns.

External Function Include Segment

The include segment apl_external_function.incl.pl1 may be used by writers of external functions. It contains declarations for operators_argument, value_bead, character_data_structure, numeric_datum, the APL status codes, and the storage management external entries. The following external entry is declared in apl_external_function.incl.pl1, and should be used to allocate space on the value stack.

apl_push_stack_

This function allocates storage on the current value stack.

Usage

```
declare apl_push_stack_ entry (fixed bin (19)) returns (ptr);

data_ptr = apl_push_stack_ (n_words);
```

where:

1. data_ptr (output)
   is the pointer to the storage that has been allocated. This storage is always aligned on an even word boundary.

2. n_words (input)
   is the number of words to be allocated on the value stack.

SECTION 6

FILE SYSTEM

THE MULTICS APL FILE SYSTEM

The Multics APL file system provides the APL user with the ability to save the values of APL computations in a file for later retrieval. Unlike APL workspaces, this saving and loading may be done under the control of APL programs, allowing complex manipulation of data and maintenance of special purpose databases. In addition, the APL file system allows several APL users to share data and perform coordinated simultaneous operations on databases.

Organization of APL Files

An APL file is divided into components, one component for each APL value in the file. APL values of any shape or rank, containing numeric or character data may be stored in a component of a file.

Each component has associated with it a positive integer corresponding to its position in the file called its component number. This component number is used to refer to the component and to the APL value associated with it. Components in a file are numbered consecutively, starting with 1 for the first component added to a new file. Components can be deleted from either end of a file, so the number of the first component in the file is not always 1. Components can not be deleted from the middle of the file. Thus there are never any gaps in the components in a file (e.g. 5 6 7 8 9 is a possible numbering of all of the components in a file while 5 6 8 9 is not).

APL files differ from workspaces in several respects:

1. More than one file may be active at a time in one APL user's environment. Each file is referred to by a unique number, so all active files may be referenced concurrently.

2. Files may be shared simultaneously by multiple APL users. Locking mechanisms are provided to ensure file integrity.

3. The files are manipulated by APL system functions, allowing file operations to be performed from within APL programs as well as from the keyboard.

Use of APL Files

The first step in using an APL file is to associate the file with a positive integer called a "file number". This process is called tying a file. The file number (or tie number) is used to identify the file for all subsequent file operations. The value of the file number is chosen by the user and must be different from any other file number in use at the time it is selected.

After the file has been tied, the file system functions may be used to perform operations on it. Each function takes as an argument the file number of the file to be operated upon. In addition, some files require a component number specifying the particular value in question. Some of the file system functions return a result, others do not.

File Manipulation Functions

⎕*FCREATE*

Syntax: 'file_name' ⎕*FCREATE* file_number

The ⎕*FCREATE* function creates an APL file with the specified name and ties it to a file number for subsequent use. If no directory is specified in the file name, the file is created in the working directory. APL files have the suffix ".cf.apl". The suffix need not be supplied by the user. If a file with the specified name already exists, an error occurs. When a file is first created it is empty and the first and last component numbers are 0.

⎕*FTIE*

Syntax: 'file_name' ⎕*FTIE* file_number

The file function *⎕FTIE* ties an APL file for exclusive use. When a file is exclusively tied, no other APL user may tie the file for the duration of its use. To tie a file for shared use by multiple APL users see the *⎕FSTIE* function. Unlike *⎕FCREATE*, the specified file must already exist and no other user may have the file tied. The suffix ".cf.apl" need not be supplied.

⎕FSTIE

Syntax: 'file_name' *⎕FSTIE* file_number

The *⎕FSTIE* function ties an APL file for shared use. Any number of APL users may have a file tied for shared use simultaneously. Like *⎕FTIE* the file must already exist. No other user may have the file exclusively tied. The suffix ".cf.apl" need not be supplied.

⎕FUNTIE

Syntax: *⎕FUNTIE* file_number_vector

The *⎕FUNTIE* function closes and unties APL files. The file numbers of files that have been untied are no longer associated with those files and may be reused by subsequent *⎕FTIE* or *⎕FCREATE* operations. If the vector of file numbers is empty, no files are untied. If any of the specified file numbers are not tied to a file an error occurs, but the remainder of the specified files are untied.

⎕FRENAME

Syntax: 'file_name' *⎕FRENAME* file_number

The *⎕FRENAME* function changes the name of the file tied to file_number to file_name. The file remains otherwise unchanged, with the same components, ACL, etc. The file must be exclusively tied.

⎕*FERASE*

Syntax: 'file_name' ⎕*FERASE* file_number

The ⎕*FERASE* function deletes the APL file specified by both the file name and file number. The file must be exclusively tied. If the specified file name does not match the name of the file that is tied to the file number, an error occurs. The suffix ".cf.apl" need not be supplied.

⎕*FAPPEND*

Syntax: value ⎕*FAPPEND* file_number

⎕*FAPPEND* adds an APL value to the end of a file, giving it a component number one greater than that of the last component already in the file. The value may be the result of a computation, the value of a variable, or simply a constant.

⎕*FREPLACE*

Syntax: value ⎕*FREPLACE* file_number component_number

The function ⎕*FREPLACE* replaces the value in a specified component in a file with a new value. If the specified component number does not already exist in the file, an error occurs. The new value may be any APL value, regardless of the shape or type of the value being replaced.

⎕*FREAD*

Syntax: result ← ⎕*FREAD* file_number component_number

The ⎕*FREAD* function returns the value stored in the specified component in the specified file. The value returned may be assigned to a variable, used in further calculations, or printed out. If there is no component corresponding to the component number specified, an error occurs.

⎕FDROP

Syntax: *⎕FDROP* file_number drop_number

The function *⎕FDROP* removes components from either end of an APL file. If the number of components to be dropped is positive, that many components are removed (dropped) from the low-numbered end of the file. If the number is negative, the components are dropped from the high-numbered end of the file, and the number of components removed is equal to the absolute value of the number given. If there are fewer components in the file than the number specified, an error occurs. If all of the components are dropped from a file, the first and last component numbers are set to 0, i.e., the file looks just like a newly created file.

⎕FNUMS

Syntax: result ← *⎕FNUMS*

⎕FNUMS returns a vector with the file numbers of all the files currently tied in the user's APL environment. If there are no files tied, the empty vector is returned. The simplest way to untie all of the files currently tied is to execute the statement *⎕FUNTIE ⎕FNUMS*, passing the vector returned by *⎕FNUMS* on to *⎕FUNTIE*.

⎕FNAMES

Syntax: result ← *⎕FNAMES*

⎕FNAMES returns a character matrix, each row of which contains the pathname of the APL file tied to the corresponding file number in the vector returned by *⎕FNUMS*.

⎕FLIB

Syntax: result ← *⎕FLIB* 'library_name'

The ⎕*FLIB* functions returns a character matrix, each row of which contains the name of an APL file contained in the directory specified by library_name. The library name can be either a library number or a Multics pathname.

⎕*FLIM*

Syntax: result ← ⎕*FLIM* file_number

The ⎕*FLIM* function returns a two-element numeric vector. The first element of the vector is the component number of the first component in the file and the second element is one greater than the component number of the last component in the file, i.e., the component number that would be assigned to the next value added to the file by ⎕*FAPPEND*.

⎕*FSIZE*

Syntax: result ← ⎕*FSIZE* file_number

The ⎕*FSIZE* function returns a four-element numeric vector. The first two elements of this vector are the same as those returned by ⎕*FLIM*, the component number of the first component in the file and one greater than the component number of the last component in the file. The third element is the amount of storage currently being used by the file, in bytes. The fourth element is the maximum size to which the file may grow. This number is not meaningful in Multics APL, since the maximum size of a file is limited only by the quota allocated to the containing directory of the file. Therefore, the fourth element is only included for the sake of compatability with other APL file systems, and is defined to always be the value of the largest existing number in Multics APL, 1.701411835E38.

⎕*FRDCI*

Syntax: result ← ⎕*FRDCI* file_number component_number

⎕*FRDCI* returns a three-element numeric vector. The first element of the vector is the size of the specified component in bytes. The second element is the user number of the APL user who last wrote a value into the component. This number is only meaningful when the APL user who modified the component is part of a subsystem where user numbers are assigned, or has invoked APL with the -user_number control argument (see the apl command description). The default user number is 100. The third element of the vector is the time at which the component was written. This time is measured in a standard Multics clock reading, the number of microseconds since midnight January 1, 1901 GMT.

⎕*FHOLD*

Syntax: ⎕*FHOLD* file_number_vector

The ⎕*FHOLD* function is used to lock a file opened for shared use so that no other APL user may modify the file. ⎕*FHOLD* first unlocks all of the files currently locked by the user, then proceeds to lock all of the files specified in the vector of file numbers. Thus, calling ⎕*FHOLD* with an empty vector simply unlocks all of the currently locked files. If any of the specified files are already locked by another user, ⎕*FHOLD* waits until the file becomes free, then locks it and continues. The files are locked in an order that guarantees that no deadlock can occur between users performing ⎕*FHOLD* operations.

⎕*FLISTACL*

Syntax: result ← ⎕*FLISTACL* file_number

⎕*FLISTACL* returns a character matrix containing the Multics file system access control list (ACL) for the file. The character matrix has 36 columns -- 3 for the access modes, 1 blank column, and 32 for the access id -- and as many rows as is necessary. For more information on Multics access control, see the Multics Programmer's Reference Manual, (Order No. AG91).

⎕*FSETACL*

Syntax: access_matrix ⎕*FSETACL* file_number

The *⎕FSETACL* function takes a character matrix like the one returned by *⎕FLISTACL* and sets the ACL for the specified file to the value of the matrix. The matrix must be 36 columns wide, and may have any number of rows. However, if a vector is supplied for the left argument, it need be only as long as is necessary to contain the access modes, the blank column, and the access id (a minimum of 5 characters). The first three columns of each row must consist of some combination of the characters r e w or blank; the access modes. The fourth column must be blank. Columns 5-36 must contain a valid access id of the form used by the Multics access control commands. For more information about file system access control, see the section on access control in this chapter.

⎕FADDACL

Syntax: access_matrix *⎕FADDACL* file_number

The *⎕FADDACL* function is similar to *⎕FSETACL*, except that instead of replacing the ACL of the file with the one specified by the matrix, the entries in the matrix are added to the existing ACL. If any of the entries in the matrix duplicate an existing entry on the file's ACL, the access modes in the ACL are replaced by those in the matrix entry.

⎕FDELETEACL

Syntax: access_id_matrix *⎕FDELETEACL* file_number

The *⎕FDELETEACL* function takes a 32-column-wide character matrix, each row of which is a Multics storage system access id. Any entries on the ACL of the file matching any of the access ids in the matrix are deleted from the ACL. Entries in the matrix that are not on the ACL for the file are ignored.

Access Control

Many applications involving the use of the APL file system require the use of one file by several users. To control which users can access which files, the APL file system provides an interface to Multics storage system access control. The following discussion provides an overview of Multics access control and how the APL file system interfaces with it. For a more detailed discussion of Multics access control, see the Multics Programmer's Reference Manual (Order No. AG91).

Each APL file has associated with it a list of users that can access the file, and what form of access they have. This list is called the Access Control List, or ACL, of the file.

Each entry in the ACL is composed of two parts, the access id and the access modes. The access id is used to identify a Multics user or group of users. An access id is composed of three parts, separated by periods: the person id, the project id, and the instance tag (e.g. Smith.Student.a). Any of the fields may be replaced by an asterisk (*), signifying that any string will match that field. Thus the access id Smith.*.* will match the user Smith on any project, and *.Student.* will match any user on the Student project. The access id *.*.* matches any user on any project.

Associated with each access id are the access modes for the user. The access modes specify what access the corresponding user has to the file. There are two access modes for APL files: read (r) and write (w) access (a third access mode, execute (e), exists, but is not meaningful for APL files and may be ignored by users of the APL file system). A user may have both read and write access to a file, only read access, or no access at all (null access). The combination of write access and no read access is equivalent to having null access to the file.

Several file system functions exist for manipulating the ACLs of APL files. ⎕*FADDACL* and ⎕*FDELETEACL* add and delete ACL entries respectively, and ⎕*FSETACL* replaces the entire ACL for the file with one supplied by the user. For more detailed information, see the individual descriptions of the functions.

File Sharing

The APL file system allows the simultaneous use of an APL file by several APL users. In any application where a database is shared among multiple users it is desirable to have a mechanism for ensuring exclusive use of the file while executing a critical section of an APL program. The ⎕*FHOLD* function provides this ability.

The ⎕*FHOLD* function takes as its arguments a vector of file numbers of files to be locked. After unlocking any files that the user already has locked, it attempts to lock all of the specified files.

While a user has a file locked, no other user may perform any operations on the file that would change any of the data in the file (e.g. ⎕*FAPPEND*, ⎕*FREPLACE*). Other users may, however, read information from the file (i.e. using ⎕*FREAD*, ⎕*RDCI* etc.).

When several APL users are sharing code that does file manipulations, (e.g. interactive updating of a common database), the locking capability provided by the ⎕*FHOLD* function can be used to keep the data in a consistent state. If the critical code that performs the actual file reading and writing operations is surrounded by calls to the ⎕*FHOLD* function, then the user executing the code is guaranteed to have exclusive use of the file for the duration of the important operations. Thus any number of users can read and write the same database without interfering with one another.

SECTION 7

SYSTEM FUNCTIONS

THE MULTICS APL SYSTEM FUNCTIONS

System functions are those defined by the APL system itself; some of which are programming aids while others return information pertaining to the APL system.

The system functions, like system variables, are characterized by the ⎕ (quad) character followed by the name of the particular function, which is then followed by arguments, separated by spaces.

System functions may be called from within an APL program or from the APL environment.

The current APL system functions are:

⎕*AF*............	Active Function
⎕*AI*............	Accounting Information
⎕*CALL*..........	Call
⎕*CR*............	Canonical Representation
⎕*CS*............	Character Set
⎕*DL*............	Delay

⎕*EC*............	Execute Command
⎕*EX*............	Expunge
⎕*FX*............	Fix
⎕*LC*............	Line Counter
⎕*NC*............	Name Count
⎕*NL*............	Name List
⎕*TS*............	Time Stamp
⎕*TT*............	Terminal Type
⎕*UL*............	User Load
⎕*WA*............	Workspace Available
⎕*WU*............	Workspace Used

Active Function

Syntax

```
     ⎕AF Multics_active_function_expression
```

Multics_active_function_expression is an APL character vector that specifies a function expression that returns a value. For example:

```
     A ← ⎕AF 'TIME'
```

Semantics

The Active Function function executes a Multics active function expression from within the APL system. This APL function always returns a vector.

Notes

When this function has completed execution, the user will be returned to the APL system, unless the function is one that implicitly specifies,or results in, an exit from the APL system. The 'new_proc' and 'logout' commands should not be called by this function; use the ⎕*EC* system function for these commands.

If the Multics function resulted in an error, or could not be executed for some reason, the value returned is the error message (in APL character vector form). The error message can be assigned to an APL variable if the user desires.

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

If the function specified by the argument does not return a result, the error message 'PROCEDURE CALLED IMPROPERLY' is returned.

If the function specified by the argument cannot be located, the error message 'SEGMENT NOT FOUND' is returned.

If the argument is an empty character vector or one consisting of whitespace only, the message 'CODE 100.' is returned.

SYNTAX ERROR is a result of zero arguments (one is required).

DOMAIN ERROR is a result of a numeric argument (must be character), or a matrix or array argument (must be a single vector).

Accounting Information

Syntax

⎕*AI*

Semantics

The Accounting Information function produces a 4 element vector consisting of:

1. User Number
2. Computer time used in this session
3. Connect time since sign on
4. Typing time.

Items 2, 3, and 4 are measured in milliseconds.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

This function always returns a vector.

CONTEXT ERROR is a result of an attempt to pass an argument to the function.

Call

Syntax

```
  Function call:  V ← ⎕CALL (entry_dcl; arg1; arg2; ...; argN)
  Subroutine call:  ⎕CALL (entry_dcl; arg1; arg2; ...; argN)
```

entry_dcl is an APL character value containing a PL/I style entry declaration specifying the routine to be called, the number of arguments it takes, whether it is a subroutine or function, and the types of the arguments and function value. (Input) (See "Entry Declaration" below for details.) arg1, arg2, ... argN are the APL variables and values to be used as the arguments to the routine which is being called. (Update) If an argument is a simple variable (as opposed to a constant, an expression or an indexed variable), the value of that variable is updated to reflect any changes made by the called routine.

Semantics

An APL system function to provide APL users the ability to call a FORTRAN or PL/I routine. If the routine is a subroutine, no result is returned to APL. But if the routine is a function, the function's value is returned as the result.

Entry Declaration

The entry declaration is identical to that of PL/I (except that the 'entry' keyword is optional), with the following restrictions:

(1) The attributes in a parameter declaration must be in the following order:

dimensions, type, size and alignment.

(2) A lower bound may not be specified for a dimension.

(3) The mode (i.e 'real' or 'complex') may not be specified.

(4) The only types supported are: bit, char, entry, fixed bin, and float bin.

(5) Neither dimensions nor parameter descriptions (other than 'options (variable)') may be specified for 'entry' values.

(6) A scale factor may not be specified for 'fixed' values.

(7) 'fixed' and 'float' values may not be unaligned.

Example

A typical declaration would be:

'get_line_length_$stream(char(*), fixed bin(35)) returns(fixed bin)'

Notes

If a simple variable is passed as an argument, that variable need not have been previously assigned a value. In such a case, the value passed to the called routine for that argument has the shape and type indicated by the entry declaration and is initialized to binary zeroes.

The value of an argument must agree with the type specified in the entry declaration. For example, if an argument is to be passed as a 'bit' value, it must be numeric and contain only zeroes and ones.

The shape of an argument must agree with that specified in the entry declaration. This usually means that an argument has the shape indicated by the declaration. However, an argument that is to be passed as a 'bit' or 'char' value is also considered to have the correct shape if its rank is one greater than in the declaration, its shape when the last dimension is excluded is the same as in the declaration, and the length of the last dimension is the same as the size attribute in the declaration. For example a 3x4 character matrix may be passed as: (3, 4) char (1)' or '(3) char (4)

Either a positive integer or an asterisk may be used in the entry declaration to specify the length of a dimension or the size of a 'bit' or 'char' value. An asterisk in a dimension specification means use the current length of the corresponding dimension of the argument. An asterisk in a size attribute means use the current length of the last dimension of the argument. Asterisks may not be used when the corresponding argument is a simple variable that has not yet been assigned a value. Asterisks may only be used in the 'returns' attribute if the routine being called was written in PL/I and contains asterisks in the 'returns' attribute of its header.

If 'options (variable)' is given in place of parameter declarations, any number of arguments may be supplied. A rank N numeric argument is passed as an N-dimension array of 'float bin(63)' numbers. A rank N character argument is passed as an (N-1)-dimension array of 'char(M)', where M is the size of the arguments last dimension.

Canonical Representation

Syntax

```
     ⎕CR function_name
```

function_name must be an APL character vector that specifies a function name local to and residing in the current active APL workspace.

Semantics

The Canonical Representation function returns the character representation of a function.

Notes

This function will always return a matrix.

A 0 by 0 matrix will be returned if the function specified by the argument could not be found, or if more than one syntactically correct argument was passed.

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

SYNTAX ERROR is a result of zero arguments (a character vector is required).

DOMAIN ERROR is a result of a numeric value as the argument.

Character Set

Syntax

⎕*CS*

Semantics

The Character Set function (Atomic Vector on other systems) returns a vector of 196 characters, each of which is an element in the APL character set.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

CONTEXT ERROR is a result of an attempt to pass an argument to the function.

Delay

Syntax

```
    ⎕DL N_seconds
```

N_seconds is a positive,scalar integer value. It is the number of seconds of delay.

Semantics

The Delay function delays for N seconds and returns the actual number of seconds delayed.

Notes

If N_seconds is <= 0, then 0 is returned.

A soft interrupt will terminate the delay. The number of seconds of delay up to the interrupt are returned.

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

SYNTAX ERROR is a result of zero arguments (one integer value is required).

DOMAIN ERROR is a result of an attempt tp pass a character scalar as an argument, or any multi-element value of any kind as an argument.

Execute Command

Syntax

```
     ⎕EC Multics_command
```

Multics_command is an APL character vector that specifies the Multics command or executable segment of code to be executed. For example:

```
     ⎕EC 'cp fileone filetwo'
```

Semantics

The Execute Command function executes a Multics command or executable segment of code from within the APL system. A value is not returned by this function.

Notes

When the command has completed execution, the user will be returned to the APL system, unless the command is one that explicitly specifies,or implicitly results in, an exit from the APL system. The 'new_proc' and 'logout' commands can be executed with this function.

The character vector must be an executable Multics command or a Multics system error message will be printed to the user interface.

If the command is an active function call (i.e., returns a value), the result cannot be assigned to an APL (user defined) variable.

The command can be any Multics command, not just an active function. See apl.quadAF.info for more information on active functions.

SYNTAX ERROR is a result of zero arguments (one is required).

DOMAIN ERROR is a result of a numeric argument (must be character), or a matrix or array argument (must be a single vector).

Expunge

Syntax

```
     ⎕EX object_name
```

object_name is a character vector that specifies the name of an APL object to be erased, or the name of a character matrix whose rows are used to specify the name of an object to be erased.

Semantics

The Expunge function erases variables and functions under program control. The most local instance of the object is used.

1 is returned if:
- the object did not exist within the current workspace.
- a successful operation has taken place.

0 is returned if:
- the object could not be erased (those labels, functions or groups in the state indicator).
- the object named provided was not a valid APL name. e.g. 123*TEST*

Notes

If the function is called correctly, ⎕*EX* always returns a vector of 1's, 0's or a combination of both.

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

All labels, values or functions can be deleted using the following method:

```
                    ⎕EX (⎕NL *)
```

where ⎕*NL* is the Name List function and * is 1 , 2 or 3.

SYNTAX ERROR is a result of zero arguments.

DOMAIN ERROR is a result of a numeric argument.

Fix

Syntax

```
     ⎕FX matrix
```

matrix is an APL character matrix whose rows are syntactically correct APL code and whose format follows the guidelines for APL function definition. The argument is NOT a character vector specifying the matrix name, it is the matrix or matrix name itself. For example:

```
R ← ⎕FX   GRAPHICS       is correct

R ← ⎕FX  'GRAPHICS'      is incorrect
```

where GRAPHICS is a character matrix.

Semantics

The Fix function converts a canonical representation of a function, in character matrix form, into an APL function. If the operation is successful, a character vector specifying the new function name is returned.

Notes

If the matrix cannot be converted, for syntactic reasons, an error message will be printed and the source of the error will be displayed.

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

See apl.quadCR.info for information on the inverse operation of this function (i.e., function to canonical representation).

SYNTAX ERROR is a result of zero arguments (one character matrix must be passed), or a combination of character (scalars and/or vectors) and numeric value(s) being supplied as arguments.

CONTEXT ERROR is a result of more than one argument being supplied. At least one of the arguments must be valid for this error to occur.

DOMAIN ERROR is a result of either one or more numeric values being supplied as arguments, or one or more character scalars and/or vectors being supplied as arguments.

Line Counter Syntax

⎕*LC*

Semantics

The Line Counter function returns a numeric vector whose elements are pending and suspended lines of functions appearing on the state indicator.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

A vector is always returned.

CONTEXT ERROR is the result of an attempt to pass an argument to the function.

Name Count

```
     ⎕NC  apl_value
```

apl_value can be a character vector or a character matrix. The vector (or row of a character matrix) must specify a valid APL name and/or identifier. For example:

```
     e1)     A ← 'FUNONE'
             B ← ⎕NC A
             B
         3
     e2)     A ← 2 6 ρ 'FUNONE' 'FUNTWO'
             B ← ⎕NC A
             B
         3 3
```

Where *FUNONE* and *FUNTWO* are functions in the current active workspace.

Semantics

The Name Count function returns an integer value that specifies the type of the argument:

0 = Name is not in use at the present level. It may be used to define a variable, label, function or group.

1 = The name has been defined as a label.

2 = The name has been defined as a variable.

3 = The name has been defined as a function.

4 = The name is unavailable for use.

(The shape of the result depends on the number of the arguments. The result could be a scalar or a vector. See Notes 1 and 2 below)

Notes

An integer vector is returned if the argument is a character matrix. Each row is handled as an independent character vector and passed to the function as such.

An integer scalar is returned if the argument is a character vector.

If there are two or more names within the character vector (i.e., alphabetic and/or alphanumeric characters separated by whitespace characters) then the returned value will be a 4. This is an indication that the argument was malformed. For example:

```
      A ← ⎕NC 'FUNONE     FUNTWO'
      A
   4
```

If there are two or more character vectors being passed as arguments, then the returned value will be 0. This does not necessarily mean the names are available for use. For example:

```
      A ← ⎕NC 'FUNONE   'FUNTWO'
      A
   0
```

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

SYNTAX ERROR is a result of zero arguments.

DOMAIN ERROR is a result of an integer value as an argument.

Name List Syntax

```
     L ⎕NL N
```

N is an integer scalar or integer vector whose value(s) must be >= 1 and <= 3. The following table shows the integer value that corresponds to each type(s).

VALUE	TYPE(S)
1	Label
2	Value (ie scalar,vector,matrix,array)
3	Function

L is an optional argument that can be a character vector or scalar where each element is the first letter of an object name. Only objects whose first letter is an element in the argument (or is the argument in the case of a scalar) are considered. For example:

```
     'ARQ' ⎕NL 3
```

lists the functions whose names begin with an A , R and Q.

Semantics

The Name List function lists the names of the APL type specified by N. Only the names used within the current, active APL workspace are listed.

Notes

If N is an integer vector, the list produced by this function may not be in the correct order. That is, types could get mixed in the output list.

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

This function will always return a matrix, barring the occurrence of system errors.

A 0 by 0 matrix will be returned if the type specified by the argument could not be found, or if more than one syntactically correct argument was passed.

SYNTAX ERROR is a result of zero arguments.

DOMAIN ERROR is a result of a character value as an argument, or an integer value that is not within the specified allowable range of values (1 <= N <= 3).

Time Stamp

Syntax

⎕*TS*

Semantics

The Time Stamp function returns a 7-element vector consisting of the Year, Month, Day, Hour, Minute, Second, and Millisecond.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

CONTEXT ERROR is the result of an attempt to pass an argument to the function.

Terminal Type

Syntax

⎕*TT*

Semantics

The Terminal Type function returns a numeric scalar value identifying the type of terminal in use. The value returned is >= -11 and <= 5. In following list, the possible values are given with their respective terminal names.

-11...LA36	-7...TN300	-3...ASCII	1...CORR2741
-10...BITPAIRED	-6...ABSENTEE	-2...TELETYPE	2...2741
-9...TYPEPAIRED	-5...1030	-1...TEK4013	3...1050
-8...ARDS	-4...TELERAY11	0... -	4...3270-DAF
			5...3270

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

CONTEXT ERROR is a result of an attempt to pass an argument to the function.

User Load

Syntax

⎕*UL*

Semantics

The User Load function returns the number of users presently on the system.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

CONTEXT ERROR is the result of an attempt to pass an argument to the function.

Workspace Available

Syntax

⎕*WA*

Semantics

The Workspace Available function returns the amount of available storage in the active workspace. The unit of measurement is bytes.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

CONTEXT ERROR is the result of an attempt to pass an argument to the function.

Workspace Used

Syntax

⎕*WU*

Semantics

The Workspace Used function returns a numeric scalar whose value specifies the amount of the current workspace already in use.

Notes

If the value returned is not going to be used as the right argument to the assignment operator, or not used within the calling expression, it is printed to the user interface.

CONTEXT ERROR is a result of an attempt to pass an argument to the function.

SECTION 8

SYSTEM VARIABLES

<u>THE MULTICS APL SYSTEM VARIABLES</u>

System variables are variables reserved by the APL system. Each workspace has a set of parameters local to itself that are set by assigning a value to the system variables.

A workspace parameter has a limited range of values. An error message will be printed if there is an attempt to assign a system variable a value not within its range of values.

Typing the name of the system variable alone yields its current setting as an explicit result.

The current APL system variables are:

⎕*CT*............	Comparison Tolerance
⎕*IO*............	Index Origin
⎕*IT*............	Integer Tolerance
⎕*LX*............	Latent Expression
⎕*PP*............	Printing Precision
⎕*PW*............	Page Width
⎕*RL*............	Random Link

Comparison Tolerance

Usage

⎕*CT* ← *N*

where:

1. *N*

   is a numeric scalar >= 0 and <= 1 .

Semantics

⎕*CT* is a system variable known on other systems as 'fuzz'. When comparing two numeric values, ⎕*CT* is used in conjunction with the values themselves in determining the allowable margin of difference that distinguishes one value from the next. Two numbers are considered equal if:

(|*A* - *B*) ≤ |⎕*CT* × (*A* - *B*)

Index Origin

Usage

    ⎕IO ← N

where:

1. *N*

    is either 0 or 1 .

Semantics

The ⎕*IO* system variable changes the index origin to 0 or 1. If set to 0, the first element of a vector is accessed with the value 0 as its minimum subscript value. e.g. *V*[0] . If set to 1, the first element is accessed with the value 1 as its minimum subscript value. e.g. *V*[1]

Integer Tolerance

Usage

$\square IT \leftarrow N$

where:

1. N

   is >= 0 and <= 1 .

Semantics

The $\square IT$ system variable is used in determining which values are integers and which are not. Mulitcs APL considers a value to be the integer $\lfloor C + 0.5$ if:

$$(|C - \lfloor C + 0.5) < \square IT$$

Notes

If 0.0001 <= N < 0.5 then $\square IT$ is set to 0.

If 0.5 <= N < 1.0 then $\square IT$ is set to 1.

If N = 1 then $C = (C - (1 \mid C)) + ((1 \mid C) \geq 0.5)$

If N <= 0.0001 and $|(1 \mid C) \geq \square IT$ then $C = \lfloor C$

Latent Expression

Usage

```
    ⎕LX ← apl_expression
```

where:

1. apl_expression

   is an APL character vector that specifies an executable APL expression.

Semantics

The latent expression for any given workspace is one that is executed immediately after the workspace is loaded. If ⎕LX does not hold an expression, the keyboard is unlocked and the time and date are printed. If there is a latent expression value, however, the keyboard is locked and the expression is executed. The time and date will not appear.

Notes

If the expression could not be executed, the following error messages are printed:

```
    ⍎ ERROR

    ⍎ ⎕LX
```

Printing Precision

Usage

$\square PP \leftarrow N$

where:

1. N

   is >= 0 and <= 19

Semantics

The value of $\square PP$ indicates the degree of printing precision used to display numeric values. A value is printed to the maximum of 10 * N precision. e.g. if $\square PP \leftarrow 3$ then 5.34567 would be printed as 5.346.

Page Width

Usage

```
⎕PW ← N
```

where:

1. *N*

   is >= 30 and <= 390

Semantics

The value of ⎕*PW* indicates the maximum value of the terminal page width. That is, ⎕*PW* is the maximum number of characters that may appear on one line of terminal output.

Notes

The ⎕*PW* value is not operational when in audit, although it may appear to be. Hardcopies will not reflect the terminal appearance of the text during the audit.

Any output overflowing the ⎕*PW* setting is continued on succeeding lines and indented the appropriate number of spaces, as determined by the current)*TABS* setting.

Random Link

Usage

⎕*RL* ← *N*

where:

1. *N*

   is >= 0 and <= 34359738367

Semantics

This variable is the seed for the random number generator.

SECTION 9

STREAM I/O

THE MULTICS APL STREAM I/O FILE SYSTEM

The Multics APL stream i/o file system is provided for the purpose of manipulating Multics segments as stream files from within the APL system. This facility gives the user the ability to save and retrieve APL character data from an ascii text segment.

Operations equivalent to the APL standard file handling routines are provided, plus several which are meaningful only in the sequential access stream file environment.

To access these functions, you must enter the appropriate external function definitions, as presented in the following list:

CREATE.........	)DFN CREATE APL_CODED_$CREATE
EOF............	)MFN EOF APL_CODED_$EOF
NUMS...........	)ZFN NUMS APL_CODED_$NUMS
POSITION.......	)DFN POSITION APL_CODED_$POSITION
READ...........	)DFN READ APL_CODED_$READ
REWIND.........	)MFN REWIND APL_CODED_$REWIND
TIE............	)DFN TIE APL_CODED_$TIE
UNTIE..........	)MFN UNTIE APL_CODED_$UNTIE
WRITE..........	)DFN WRITE APL_CODED_$WRITE

CREATE

Syntax: 'path_name' ⎕*CREATE* file_number

The *CREATE* subroutine creates a segment named 'path_name' and opens it for stream i/o. File number is analogous to 'tie_number' in APL files, but the two systems are independent of each other. This means you may have a stream file 7 and an APL file 7 tied concurrently. An attempt to create an existing file will result in an error message and program termination.

EOF

Syntax: r ← *EOF* file_number_vector

The *EOF* function returns a boolean vector of 'end_of_file' status for each file specified by a number in the file_number_vector. *EOF* becomes true when the last character in a file has been read.

NUMS

Syntax: R ← *NUMS*

The *NUMS* function returns the vector of tied file numbers. An empty vector is returned if no files are tied.

POSITION

Syntax: type [,skip] POSITION file_number

The *POSITION* function repositions the file. If 'type' is:

```
¯1 : the file is rewound.
 0 : 'skip' lines are skipped.
 1 : the file is positioned after the last line.
 2 : the file is postitioned at the 'skip'th character.
 3 : 'skip' characters are skipped.
```

If 'type' is ¯1 or 1, 'skip' is ignored. The default value of 'skip' is 1. 'skip' may be a negative value, resulting in a backward skip.

READ

Syntax: R ← no_of_lines *READ* file_number

The *READ* function returns a character matrix of dimension [I J] where I is the number of lines actually read (I <= no_of_lines ... End of file may be reached before the specified number of lines are read). and J is the length of the longest line read (J <= 512). Lines containing less than J characters are right padded with blanks.

REWIND

Syntax: *REWIND* file_number_vector

The *REWIND* subroutine positions each file specified in the file_number_vector such that the next READ operation would read the first line in the file or the next WRITE would truncate and replace the file contents with new data.

TIE

Syntax: 'path_name' *TIE* file_number

The *TIE* subroutine opens a segment named 'path_name' for stream i/o. Pathname may be relative or absolute.

UNTIE

Syntax: *UNTIE* file_number

The *UNTIE* subroutine closes and detaches a segment opened for stream i/o.

WRITE

Syntax: 'char_vector' *WRITE* file_number

The *WRITE* subroutine writes the characters of the left argument to the designated stream file, specified by the right argument. Newline characters (octal escape ¨012) must be included explicitly if desired.

SECTION 10

EXTERNAL FUNCTIONS

THE MULTICS APL EXTERNAL FUNCTIONS

The following procedures and functions are available for use by an APL program when the appropriate definitions are made within the workspace where the program resides. These external procedures and functions are written in PL/1 but contain the outward syntactic properties that define an APL function.

The appropriate function definitions that should be made in the programs workspace are listed in the Usage section of each individual function/procedure description.

The following is a list of the current APL external functions:

ERF..................	apl_erf_
PICKUP...............	apl_pickup_float_bin_2_
IOA..................	apl_ioa_
GET-LIST-NUMS........	apl_get_list_nums_
READ-SEGMENT.........	apl_read_segment_

apl_erf_

A scalar , monadic function that accepts any numeric argument and applies the pl1 builtin "error function" to each value.

Usage

```
)MFN ERF APL_ERF_
R <- ERF  value
```

where:

1. value
   is a numeric value of any shape.

2. R
   is the result value. It has the same shape as the argument.

apl_pickup_float_bin_2_

Read double-precision floating point numbers from a Multics segment into an APL vector.

Usage

```
)MFN PICKUP APL_PICKUP_FLOAT_BIN_2_
V ← PICKUP 'path_name'
```

where:

1. 'path_name'
   is the pathname of a Multics segment containing the floating point numbers. The pathname can be relative or absolute.

2. V
   is the vector of numbers that is returned.

Notes

The bitcount must be set to 72 times the number of elements.

apl_ioa_

Permits an APL program to do formated or unformated output using the ioa_ subroutine.

Usage

```
)DFN IOA APL_IOA_
)DFN IOA_NNL APL_IOA_$NNL
)DFN IOA_RS APL_IOA_$RS
)DFN IOA_RSNNL AL_IOA_$RSNNL
'control_string' IOA apl_value_list
'control_string' IOA_NNL apl_value_list
R <- 'control_string' IOA_RS apl_value_list
R <- 'control_string' IOA_RSNNL apl_value_list
```

where:

1. 'control_string'
   is an APL character scalar or vector that specifies the ioa_ control string. Either the circumflex (^) or APL overbar("253) character may be used as the ioa_ control character.

2. apl_value_list
   is either a single APL value (or expression), or a list of APL values. A list of values has the form (VAL1;VAL2;VAL3 ...) where each value is separated from the next by a semicolon, and the entire list is surrounded by parentheses.

Notes

All character arguments are raveled into PL/I character strings, no matter what the original APL shape.

All numeric arguments are raveled into a PL/I array of one dimension, no matter what the original APL shape.

apl_get_list_nums_

Reads and converts a segment containing PL/1 or FORTRAN numbers (in character form) into an APL numeric vector.

Usage

```
)MFN GET_LIST_NUMS APL_GET_LIST_NUMS_
R <- GET_LIST_NUMS 'path_name'
```

where:

1. 'path_name'
   is the relative or absolute pathname of the Multics text segment to be read.

2. R
   is the numeric vector that is returned by the function. The length of the vector is the number of values that were read.

Notes

The data in the segment must be in a form suitable for reading by PL/I list directed input. The segment (or multisegment file) is opened for stream input and read using list directed input until all of the data has been read. A message is printed giving the (1 origin) index of any values that cannot be converted, along with the bad value itself. A zero is substituted in the result vector for these bad values.

PL/I list directed input permits any white space character (blank, tab, new line, new page) to be used as a delimiter between values. Also, a comma may be used to separate values. Two adjacent commas cause a zero to be returned. Any of the forms of PL/I (or FORTRAN) constants may be used; the value may be signed, may have a decimal point, may have a (signed) exponent, and may be binary or decimal, fixed or float. Only the real part of any complex values is used. Since blanks are a delimiter, no blanks may appear within a single value.

This function can print non-APL error messages; therefore it is advisable to run in)ERRS LONG mode.

apl_read_segment_

Reads a Multics segment into an APL character vector.

```
 )MFN READ_SEGMENT APL_READ_SEGMENT_
 R ← READ_SEGMENT 'path_name'
```

where:

1. 'path_name'
   is the pathname of the Multics segment to be read. It can be relative or absolute.

2. R
   is the character vector that is returned by the function.

Notes

The segment is read in as a character vector, containing as many elements as there are bytes in the segment. Thus, lines are separated by new line characters (octal 012), not the usual apl matrix convention. The output of the ⎕*CS* system function may be subscripted to obtain a new line character in a program.

While the most useful form is probably to read in text segments, no restriction is placed on the type of data that may be read; any segment is ok.

This is the most efficient method for reading data from segments into apl.

SECTION 11

APL SAMPLE PROGRAMS

APL SAMPLE PROGRAMS

The programs presented in this section were written with the intent of providing programmers, less familiar with APL, an opportunity to become more familiar with the language.

Many system variables, system functions, external functions and standard APL operators are included to demostrate their usefulness within an APL program.

APL Programming Style

To avoid confusion, the programs are written in an uncondensed format. Many APL programmers prefer to reduce the length of a program by condensing the code in each function.

Executing a Sample Program

A list of all the functions used by each program are listed following the 'Description' section. Many functions are not user defined. If you wish to try running these programs, you must make the appropriate function definitions. Every external function definition can be found in Section 10 of the manual. All stream i/o function definitions are found in Section 9 of the manual. You must enter these definitions before a program will execute correctly. They need only be entered once if they are saved in the workspace with the calling program. To make a function definition, enter the definition while APL is at command level. Make sure the workspace is saved before leaving the APL system, or the definitions will be lost and need to be reentered before executing the program the next time.

Card Dealer

Function:

This program deals four hands of thirteen randomly selected cards.

Description:

There are two user defined functions in the program. The main function, *DEAL*, calls the second function, *COMMON*, three times during its execution. This program utilizes the APL stream i/o facility, writing the four hands to the Multics segment 'deck'.

```
      )FNS
COMMON  CREATE  DEAL    EOF     TIE     UNTIE   WRITE


      ∇DEAL[⎕]∇
    ∇ DEAL ; H1 ; H2 ; H3 ; H4;
⍝ TIE THE FILE AND OPEN FOR STREAM I/O
[1]   'DECK' TIE 1
⍝ CREATE A 'DECK' OF 52 CARDS. COLUMN ONE HOLDS THE RANK OF THE
⍝ CARDS WHILE COLUMN TWO HOLDS THE SUIT.
[2]   CARDS ← 52 2 ρ 'AC2C3C4C5C6C7C8C9CTCJCQCKCAD2D3D4D5D6D7D8
      D9DTDJDQDKDAH2H3
¨CH4H5H6H7H8H9HTHJHQHKHAS2S3S4S5S6S7S8S9STSJSQSKS'
⍝ CREATE A MATRIX OF 1'S. THIS VARIABLE WILL BE USED TO MASK OUT
⍝ PREVIOUSLY CHOOSEN CARDS FROM THE 'CARDS' MATRIX
[3]   CHART ← 1 52 ρ 1
⍝ RESET SEED FOR RANDOM NUMBER GENERATOR USING TIME AND DATE
[4]   ⎕RL ← (⌶20) + (⌶21) + (⌶25)
⍝ GENERATE 13 RANDOM NUMBERS, NONE OF WHICH IS LESS THAN 1 OR
⍝ GREATER THAN 52.  ASSIGN THE RANDOM NUMBERS TO 'ROLL'. EACH
⍝ ELEMENT IN 'ROLL' IS THE ROW INDICE OF 'CARDS'.
[5]   H1 ← CARDS[(ROLL ← 13 ? 52) ;]
⍝ WRITE THE FIRST HAND TO THE FILE 'DECK'.
[6]   (,(H1[;])) WRITE 1
⍝ WRITE A LINEFEED CHARACTER TO THE FILE 'DECK'.
[7]   '
' WRITE 1
⍝ HAND TWO.
[8]   H2 ← COMMON 1 39
[9]   (,(H2[;])) WRITE 1
[10]  '
' WRITE 1
⍝ HAND THREE
[11]  H3 ← COMMON 1 26
[12]  (,(H3[;])) WRITE 1
[13]  '
' WRITE 1
⍝ HAND 4
```

```
[15]  (,(H4[;])) WRITE 1
[16]  '
' WRITE 1
[17]  UNTIE 1
    ∇


      ∇COMMON[⎕]∇
    ∇ FINAL ← COMMON INC
⍝ EACH ELEMENT IN 'CHART' AT 'ROLL[N]' (WHERE 1 ≤ N ≤ 13) BECOMES
⍝ EQUAL TO 0.
[1]   CHART[1;ROLL] ← 0
⍝ MASK OUT EACH ROW OF 'CARDS' THAT CORRESPONDS TO A 0 ELEMENT IN
⍝ 'CHART'.THE CARDS FROM THE LAST HAND HAVE REMOVED FROM THE DECK.
[2]   CARDS ← CHART[1;] ⌿ CARDS
⍝ CREATE A NEW 'CHART' VECTOR WITH (⍴CARDS)[1] ELEMENTS.
[3]   CHART ← INC ⍴ 1
⍝ PRODUCE A NEW HAND
[4]   FINAL ← CARDS[ROLL←(CHART[1;ROLL]
    ∇


      )Q
```

Graph Plotting

Function:

This program produces a graph by reading a data file containing x y coordinates and plotting them to the scale specified by the contents of the APL numeric vector *DATA*.

Description:

There are fifteen functions in this program, eleven of which are user written. The main function, *PLOT*, calls all the user written functions directly, with the exception of *STORE* and *CONVERT*. The values returned by each function called by *PLOT* are used as input to the next function.

The variable *DATA* is a six element vector containing information pertaining to the scaling and axis labeling of the graph.

DATA[1]:	Minimum x coordinate in the input file. This value is a feature to the user, for the purpose of verifying the data.
DATA[2]:	Minimum y coordinate in the input file. This value is also for data verification.
DATA[3]:	At every *DATA*[3] space on the x-axis, number the axis at that point, if space permits.
DATA[4]:	At every *DATA*[4] space on the y-axis, number the axis at that point.
DATA[5]:	Every space on the x-axis is an increment of *DATA*[5]. eg. *DATA*[5] ← 0.25 means each space is an increment of 0.25. Therefore, the value 1 would require 4 spaces. If *DATA*[5] ← 0.75 then the value 3 would require 4 spaces.
DATA[6]:	Every space on the y-axis is an increment of *DATA*[4].

To get the most accurate graph possible, you may need to readjust the values in *DATA*. Range restrictions are done automatically, so graph space is maximized, regardless of *DATA* values. That is, if the maximum value in the data file is 10 and the minimum is -5, only the values between -5 and 10 are printed.

```
      )FNS
CONVERT GET_LIST_NUMS   INPUT    LIMITS  MATRIX  MAXMIN  PLOT    POINTS
SECTIONS         STORE  TIE      UNTIE   WRITE   XSIDE   YSIDE


⍝ THIS FUNCTION, THE DRIVER, REQUIRES ONE CHARACTER VECTOR AS AN
⍝ ARGUMENT.  THE ARGUMENT SPECIFIES THE NAME OF THE INPUT FILE
⍝ CONTAINING THE DATA.

      ∇PLOT[⎕]∇
    ∇ Z ← PLOT FILE ; V ;
⍝ CALL THE REST OF THE FUNCTIONS.
[1]    Z ← V POINTS (XSIDE YSIDE MATRIX SECTIONS LIMITS MAXMIN
     (V ← INPUT FILE))
    ∇



⍝ THIS FUNCTION TAKES THE FILE NAME PASSED TO THE DRIVER FUNCTION
⍝ AS ITS ARGUMENT. THIS FUNCTION CALLS THE EXTERNAL FUNCTION
⍝ GET-LIST-NUMS TO READ NUMERIC VALUES FROM A MULTICS SEGMENT
⍝ INTO AN α*⎕ VECTOR.

      ∇INPUT[⎕]∇
    ∇ Z ← INPUT FILE ; TEMP ;
⍝ STORE THE DATA IN THE VECTOR 'TEMP'
[1]    TEMP ← GET_LIST_NUMS FILE
⍝ RESHAPE THE VECTOR, SUCH THAT COLUMN ONE CONTAINS X VALUES WHILE
⍝ COLUMN TWO CONTAINS THE RESPECTIVE Y VALUES.
[2]    Z ← (((⍴TEMP) ÷ 2),2) ⍴ TEMP
    ∇



⍝ THIS FUNCTION TAKES THE DATA MATRIX AS ITS ARGUMENT. THE
⍝ MAXIMUM AND MINIMUM VALUES ARE COMPUTED IN THIS FUNCTION.
⍝ ELEMENTS 3, 4, 5, AND 6 OF 'DATA' ARE SET TO THEIR
⍝ ABSOLUTE VALUES.

      ∇MAXMIN[⎕]∇
    ∇ Z ← MAXMIN INFO ; XMAX ; YMAX ;
[1]    XMAX ← ⌈ / INFO[;1]
[2]    YMAX ← ⌈ / INFO[;2]
[3]    DATA[1] ← ⌊(⌊ / INFO[;1])
[4]    DATA[2] ← ⌊(⌊ / INFO[;2])
[5]    DATA[5] ← |DATA[5]
[6]    DATA[6] ← |DATA[6]
[7]    DATA[3] ← |DATA[3]
[8]    DATA[4] ← |DATA[4]
⍝ ENSURE THAT THE TWO MINIMUM VALUES CAN BE DIVIDED EVENLY BY
⍝ THE INCREMENT VALUES IN 'DATA[5]' AND 'DATA[6]'.
[9]    DATA[2] ← DATA[2] - (DATA[6] | DATA[2])
```

```
[11] DONE: Z ← XMAX,YMAX
     ∇



⍝ THIS FUNCTION COMPUTES THE GRAPHS UPPER AND LOWER LIMITS,
⍝ COMPLYING WITH THE INCREMENT LEVEL SPECIFIED IN 'DATA'. THE
⍝ MAXIMUM X Y VALUES ARE ROUNDED SUCH THAT THE MAXIMUM VALUE
⍝ ON EACH AXIS IS EVENLY DIVISIBLE BY 'DATA[5]' AND 'DATA[6]'.

       ∇LIMITS[⎕]∇
     ∇ Z ← LIMITS MINMAX ; TOPX ; TOPY ;
⍝ IF THE MAXIMUM Y VALUE IS EVENLY DIVISIBLE BY 'DATA[6]', GO TO
⍝ LABEL 1.
[1]    → ((DATA[6] | MINMAX[2]) .FNT
⍝ THE MAXIMUM Y VALUE IS NOT EVENLY DIVISIBLE BY 'DATA[6]'.
⍝ ROUND THE MAXIMUM Y VALUE UP TO THE FIRST NUMBER EVENLY
⍝ DIVISIBLE BY 'DATA[6]'.
[2]    TOPY ← MINMAX[2] + (DATA[6] - (DATA[6] | MINMAX[2]))
[3]    → L2
⍝ ROUND THE MAXIMUM Y VALUE DOWN TO THE FIRST NUMBER EVENLY
⍝ DIVISIBLE BY 'DATA[6]'.
[4]  L1: TOPY ← MINMAX[2] + (DATA[6] - (DATA[6] | MINMAX[2])) - DATA[6]
⍝ THE SAME PROCEDURE AS ABOVE APPLIES THE THE X MAXIMUM VALUE.
[5]  L2: → ((DATA[5] | MINMAX[1]) .FNT
[6]    TOPX ← MINMAX[1] + (DATA[5] - (DATA[5] | MINMAX[1]))
[7]    → L4
[8]  L3: TOPX ← MINMAX[1] + (DATA[5] - (DATA[5] | MINMAX[1])) - DATA[5]
[9]  L4: Z ← TOPY,TOPX
     ∇



⍝ THE ARGUMENT OF THIS FUNCTION IS THE MODIFIED MAXIMUM AND
⍝ MINIMUM VALUES COMPUTED BY THE 'LIMITS' FUNCTION. THE
⍝ TOTAL NUMBER OF SPACES NEEDED TO CREATE THE GRAPH IS
⍝ CALCULATED HERE.

       ∇SECTIONS[⎕]∇
     ∇ Z ← SECTIONS XY ; X ; Y ;
⍝ Y    (YMAX + ABS(YMIN)) ÷ YINCREMENT
[1]    Y ← (XY[1] + (|DATA[2])) ÷ DATA[6]
⍝ X    (XMAX + ABS(XMIN)) ÷ XINCREMENT
[2]    X ← (XY[2] + (|DATA[1])) ÷ DATA[5]
[3]    Z ← Y,X
     ∇


⍝ THE ARGUMENT TO THIS FUNCTION IS A VECTOR CONTAINING THE NUMBER
⍝ OF SPACES ON THE Y AXIS AND X AXIS.  THIS FUNCTION CONSTRUCTS
⍝ THE ACTUAL MATRIX. THE AXIS ARE DRAWN AND AT EVERY 'DATA[4]' AND
⍝ 'DATA[3]' SPACE, A '+' IS SUBSTITUTED FOR THE '|' AND '-'
⍝ ON EACH RESPECTIVE AXIS.
```

```
      ∇MATRIX[⎕]∇
    ∇ Z ← MATRIX XY ; M ;
⍝ CONSTRUCT A (YMAX + 1) × (XMAX + 1) MATRIX.  THE ADDITION VALUE
⍝ OF ONE ON EACH AXIS ACCOUNTS FOR THE NUMBERING WHICH WILL OCCUR
⍝ LATER.
[1]   M ← ((XY[1] + 1),(XY[2] + 1)) ρ ' '
⍝ DRAW THE Y AXIS IN COLUMN 1.
[2]   M[;1] ← '|'
⍝ DRAW THE X AXIS IN THE LAST ROW.
[3]   M[XY[1] + 1;] ← '-'
⍝ LET I .FNT
[4]   I ← (ρM)[1]
⍝ STARTING AT THE 'BOTTOM' OF THE MATRIX, PLACE A '+' WHERE THE
⍝ TWO AXIS MEET AND THEN ENTER A LOOP THAT PLACES A '+' AT
⍝ EVERY 'DATA[4]' SPACES ON THE Y AXIS. I IS DECREMENTED EVERY
⍝ PASS THROUGH THE LOOP, AS IT SPECIFIES THE ROW THE '+' SHOULD
⍝ BE PRINTED IN. STOP THE LOOP WHEN THE 'TOP' OF THE MATRIX HAS
⍝ BEEN REACHED.
[5]  L1: M[I;1] ← '+'
[6]   I ← I - DATA[4]
[7]   →   (I ≥ 1) / L1
⍝ STARTING AT COLUMN 1, AND THE 'BOTTOM' OF THE MATRIX, START
⍝ PRINTING THE '+' ALONG THE X AXIS.  THIS PROCESS IS MUCH THE
⍝ SAME AS THE PROCESS DESCRIBED ABOVE, BUT INSTEAD OF DECREMENTING
⍝ I, IT IS INCREMENTED BY 'DATA[3]' AT EACH PASS THROUGH THE
⍝ LOOP.
[8]   I ← 1
[9]  L2: M[(ρM)[1];I] ← '+'
[10]  I ← I + DATA[3]
[11]  → (I ≤ (ρM)[2]) / L2
[12]  Z ← M
    ∇


⍝ THIS FUNCTION IS A TOOL USED BY 'YSIDE' AND 'XSIDE'. IT TAKES
⍝ A NUMERIC ARGUMENT AND CONVERTS IT TO A CHARACTER VECTOR.
⊢004⍝ THIS FUNCTION WILL CONVERT REAL NUMBER VALUES IN DECIMAL
⍝ NOTATION UP TO FOUR DECIMAL PLACES OF ACCURACY.

      ∇CONVERT[⎕]∇
    ∇ Z ← CONVERT V ; S ; T ; N ; A ; B
⍝ SET THE INDEX ORIGIN TO 0.  THIS IS NECESSARY TO INDEX INTO
⍝ THE VARIABLE 'N'.
[1]   ⎕IO ← 0
⍝ IF THE VALUE IS NEGATIVE, MAKE THE FIRST ELEMENT OF THE
⍝ OUTPUT CHARACTER VECTOR, 'A', A '-'. IF THE VALUE IS POSITIVE,
⍝ MAKE THE FIRST ELEMENT OF 'A' A BLANK.
[2]   → (V < 0) / NG
⍝ HANDLE A POSITIVE VALUE.
[3]   A ← ' '
[4]   → START
⍝ HANDLE A NEGATIVE VALUE.
[5]  NG: A ← '-'
```

```
[6]   V ← V × ¯1
⍝ INTIALIZE 'N'.
[7]   START: N ← '0123456789'
[8]   B ← ''
⍝ DOES THE VALUE HAVE A DECIMAL PORTION? IF NO, PROCESS INTEGER.
⍝ ELSE, ASSIGN THE DECIMAL PORTION TO 'S' AND SET THE FIRST
⍝ ELEMENT OF THE CHARACTER VECTOR, 'B', TO '.'.
[9]   → ((1 | V)
[10]  S ← 1 | V
[11]  B ← '.'
⍝ IF A DECIMAL PORTION EXISTS, PROCESS IT IN THIS LOOP.
⍝ MULTIPLY THE DECIMAL VALUE BY 10.  THE FIRST DIGIT IN THE
⍝ DECIMAL VALUE WILL BE TO THE LEFT OF TH DECIMAL POINT AFTER
⍝ DOING THIS. E.G  .344 × 10
⍝ NOW 'S' IS EQUAL TO 3.44.  THE LAST OPERATION IN LINE [12]
⍝ IS 3.44 - .44  OR  (S - (1 | 3.44)). THIS GIVES US THE VALUE
⍝ 3, WHICH IS ASSIGNED TO THE VARIABLE 'T'..
[12] L0: T ← S - (1 | (S ← ((1 | S) × 10)))
⍝ JOIN THE CHARACTER 'N[(⌊T)]' WITH THE VECTOR 'B'.
[13]  B ← B,N[(⌊T)]
⍝ CONTINUE THE PROCESS DESCRIBED ABOVE UNTIL THE DECIMAL PORTION
⍝ IS LESS THAN 0.00009.
[14]  → (S > 0.00009) / L0
⍝ PROCESS THE INTEGER PORTION OF THE INITIAL VALUE. IT IS MUCH
⍝ THE SAME OPERATION AS THAT DESCRIBED ABOVE, EXCEPT THAT THE
⍝ VALUE IS BEEN MULTIPLIED BY 10 INSTEAD OF BEING DIVIDED BY 10.
[15] L1: V ← (V - (T ← (10 | V))) ÷ 10
[16]  A ← A,N[(⌊T)]
[17]  → (V > 0) / L1
⍝ JOIN THE DECIMAL VECTOR TO THE INTEGER VECTOR. THE LAST
⍝ ELEMENT OF 'B' IS DROPPED AND THE REMAINDER IS JOINED TO THE
⍝ TRANSPOSE OF 'A[1]' TO 'A[(⍴A)]'. THE RESULT OF THIS OPERATION
⍝ IS JOINED WITH 'A[0]'.
[18]  Z ← A[0],(⌽(1 ↓ A)), (¯1 ↓ B)
⍝ RESET THE INDEX ORIGIN TO 1.
[19]  ⎕IO ← 1
    ∇


⍝ THIS FUNCTION TAKES THE MATRIX CREATED BY 'MATRIX' AS ITS
⍝ ARGUMENT.  THE Y AXIS IS NUMBERED AT EVERY 'DATA[4]' SPACE.

      ∇YSIDE[⎕]∇
    ∇ Z ← YSIDE M ; YS ; YM ; TEMP ; BLANKS ; I ;
⍝ CREATE A VECTOR OF BLANKS.
[1]   BLANKS ← 10 ⍴ ' '
⍝ 'YM' BECOMES EQUAL TO THE MAXIMUM NUMBER A Y AXIS NUMBER MAY
⍝ REACH. 'Y2' IS TEH MINIMUM DATA VALUE.
[2]   YM ← DATA[2] + DATA[6] × ((⍴M)[1] - 1)
[3]   YS ← DATA[2]
⍝ CREATE A MATRIX OF SPACES.  TO ENSURE THAT EVERY NUMBER BETWEEN
⍝ YMIN AND YMAX CAN BE PRINTED, THE CHARACTER LENGTH OF EACH VALUE
⍝ IS COMPARED.  THE LENGTH OF THE LARGER OF THE TWO IS USED IN
⍝ THE CREATION OF THE TEMPORARY MATRIX.  THE TEMPORARY MATRIX WILL
⍝ BE USED AS A TEMPORARY BUFFER FOR THE AXIS NUMBERS.  LATER, WHEN
⍝ ALL THE NUMBERS HAVE BEEN CONVERTED AND PLACED IN THE TEMPORARY
⍝ MATRIX, THE MAIN MATRIX AND THE TEMPORARY MATRIX WILL BE JOINED
⍝ ALONG THE Y AXIS.
[4]   → ( (ρ(CONVERT YS)) > (ρ(CONVERT YM)) ) / Y1
[5]   TEMP ← ( (ρM)[1],(4 + (ρ(CONVERT YM))) ) ρ ' '
[6]   → Y2
[7]  Y1: TEMP ← ( (ρM)[1],(4 + (ρ(CONVERT YS))) ) ρ ' '
⍝ SET 'I' TO THE NUMBER OF ROWS IN  MATRIX 'M'.
[8]  Y2: I ← (ρM)[1]
⍝ 'TEMP[I;]' IS REPLACED BY THE CHARACTER REPRESENTATION OF THE
⍝ Y AXIS NUMBER VALUE.  IF THE CONVERTED VALUES LENGTH IS LESS
⍝ THAN THE NUMBER OF COLUMNS IN 'TEMP', IT MUST BE PADDED WITH
⍝ BLANKS.
[9]  L1: TEMP[I;] ← ( ( (ρTEMP)[2] - ρ(CONVERT YS) ) ↑ BLANKS),CONVERT Y
⍝ DECREMENT 'I' TO THE NEXT ROW THAT REQUIRES A Y AXIS NUMBER.
[10]  I ← I - DATA[4]
⍝ COMPUTE THE NEXT Y AXIS NUMBER.
[11]  YS ← YS + (DATA[4] × DATA[6])
⍝ QUIT WHEN THE 'TOP' OF THE MATRIX HAS BEEN REACHED.
[12]  → (I ≥ 1) / L1
⍝ JOIN A COLUMN OF BLANKS TO THE TEMPORARY BUFFER TO SEPARATE
⍝ THE NUMBERS IN 'TEMP' FROM THE AXIS IN 'M'.
[13]  TEMP ← TEMP,((ρTEMP)[1],1) ρ ' '
⍝ JOIN THE Y AXIS NUMBER LABELS TO THE MAIN MATRIX.
[14]  Z ← TEMP , M
    ∇


⍝ THIS FUNCTION TAKES THE MATRIX WITH THE LABELED Y AXIS AS AN
⍝ ARGUMENT.  THE X AXIS IS LABELED IN THIS FUNCTION. A TEMPORARY
⍝ MATRIX 'TEMP' IS CREATED TO STORE THE NUMBER LABELS AS THEY
⍝ ARE COMPUTED AND CONVERTED TO THEIR CHARACTER RESPRESENTATION.

       ∇XSIDE[⎕]∇
    ∇ Z ← XSIDE M ; XS ; XM ; TEMP ; I ;
⍝ 'XS'
[1]   XS ← DATA[1]
⍝ 'I' .FNT
[2]   I ← M[(ρM)[1] ;] ⍳ '+'
⍝ CREATE A TEMPORARY VECTOR OF BLANKS.
[3]   TEMP ← (1,((ρM)[2] + 1)) ρ ' '
⍝ 'XM'
[4]   XM ← DATA[1] + DATA[5] × ((ρM)[2] - I)
⍝ IF THERE IS INSUFFICIENT SPACE TO PRINT EACH NUMBER ON THE
⍝ X AXIS, DON'T PRINT ANY NUMBER LABELS.
[5]   → ((ρ(CONVERT XM)) > (DATA[3] - 1)) /STOP
⍝ 'X'
⍝ X AXIS.
[6]  MAIN: X ← CONVERT XS
⍝ 'INC2' .FNT
[7]   INC2 ← ρX
```

```
⍝ IF LENGTH 'X' IS ODD, 'INC' .FNT
⍝ ELSE 'INC'
[8]    → ((1 | (⍴X) ÷ 2) > 0) / OD
[9]    INC ← ¯1 × (((⍴X) ÷ 2) - 1)
[10]   → GO
[11] OD:INC ← ¯1 × (((⍴X) ÷ 2) - 0.5)
⍝ ODD LENGTHED NUMBER LABELS ARE CENTERED UNDER THEIR RESPECTIVE
⍝ '+' APPEARING IN THE 'M' MATRIX.  EVEN LENGTHED NUMBER LABELS
⍝ ARE CENTERED WITH THE 'BEST FIT' METHOD. NEGATIVE NUMBERS WILL
⍝ HAVE THE EXCESS CHARACTER TO THE LEFT OF THE '+' WHILE POSITIVE
⍝ NUMBERS WILL HAVE THE EXCESS CHARACTER TO THE RIGHT OF THE '+'.
⍝ THE NUMBER LABELS ARE TRANSFERRED TO THE TEMPORARY VECTOR ONE AT
⍝ A TIME.
[12] GO: TEMP[1;(I - INC)] ← X[INC2]
[13]   INC2 ← INC2 - 1
[14]   INC ← INC + 1
⍝ IF THE NUMBER LABEL HAS BEEN TRANSFERRED, COMPUTE THE NEXT
⍝ NUMBER LABEL. ELSE, TRANSFER THE NEXT CHARACTER IN THE NUMBER
⍝ LABEL TO THE TEMPORARY MATRIX.
[15]  → (INC2 ≥ 1) / GO
⍝ COMPUTE THE NEXT NUMBER LABEL.
[16]  XS ← XS + DATA[3] × DATA[5]
⍝ ADVANCE TO THE NEXT LOCATION ON THE X AXIS.
[17]  I ← I + DATA[3]
⍝ IF THE END OF THE X AXIS HAS BEEN REACHED, QUIT.⊢005
[18]  → (XS ≤ XM) / MAIN
⍝ JOIN A BLANK LINE TO THE 'BOTTOM' OF THE MATRIX 'M'.
[19] STOP: M ← M,[2](((⍴M)[1],1) ⍴ ' ')
⍝ JOIN THE X AXIS NUMBER LABELS TO THE MAIN MATRIX.
[20]  Z ← M,[1]TEMP
    ∇


⍝ TO THIS POINT, THE GRAPH HAS BEEN CREATED AND IS READY TO HAVE
⍝ THE DATA PLOTTED.  ROUNDING OF VALUES IS SLIGHTLY DIFFERENT
⍝ FOR POSITIVE AND NEGATIVE VALUES. NEGATIVE VALUES ARE ROUNDED
⍝ DOWN WHILE POSITIVE VALUES ARE ROUNDED UP.  THIS REDUCES THE
⍝ APPEARANCE OF GRAPHIC INCONSISTENCIES BETWEEN NEGATIVE AND
⍝ POSITIVE VALUES.

      ∇POINTS[⎕]∇
    ∇ Z ← V POINTS M ; X ; Y ; I ;
[1]    I ← 1
⍝ GET DATA FROM DATA MATRIX 'V'.
[2]  L1: X ← V[I;1]
[3]    Y ← V[I;2]
⍝ ROUND THE VALUES TO THE RULES SPECIFIED ABOVE.
[4]    → ((DATA[6] | (|Y)) ≥ (DATA[6] ÷ 2)) / L2
[5]    Y ← (×Y) × (Y × (×Y)) - (DATA[6] | (|Y))
[6]    → L3
[7]  L2: Y ← (×Y) × (Y × (×Y)) + (DATA[6] - (DATA[6] | (|Y)))
[8]  L3: → ((DATA[5] | (|X)) ≥ (DATA[5] ÷ 2)) / L4
[9]    X ← (×X) × (X × (×X)) - (DATA[5] | (|X))
```

```
[10]  → L5
[11] L4: X ← (×X) × (X × (×X)) + (DATA[5] - (DATA[5] | (|X)))
⍝ SCALE THE X AND Y COORDINATES.
[12] L5: Y ← ((⍴M)[1] - 1) - ((DATA[2] ÷ ¯1) + Y) ÷ DATA[6]
[13]  X ← (M[(⍴M)[1] - 1;] ⍳ '+') + ((DATA[1] ÷ ¯1) + X) ÷ DATA[5]
⍝ PRINT AN '*' AT THE CORRECT SPECIFIED BY X AND Y.
[14]  M[Y;X] ← '*'
⍝ WHILE DATA IS NOT EXHAUSTED, CONTINUE PLOTTING THE POINTS.
[15]  → ( (I ← I + 1) ≤ (⍴V)[1] ) / L1
⍝ THE FINAL PRODUCT IS RETURNED !
[16]  Z ← M
    ∇


⍝ THIS FUNCTION IS OPTIONAL. IT STORES THE GRAPH IN A MULTICS
⍝ SEGMENT.  THE STREAM I/O FACILITY IS UTILIZED, SO GRAPHS MAY
⍝ SENT TO A PRINTER IF DESIRED.

       ∇STORE[⎕]∇
    ∇ STORE M ; I ;
[1]    I ← 1
[2]    'PLOTS' TIE 1
⍝ APPEND A LINEFEED CHARATER TO EACH ROW OF THE GRAPH BEFORE IT
⍝ IS WRITTEN TO FILE.
[3]   L1: (M[I;],⎕CS[11]) WRITE 1
[4]    I ← I + 1
[5]    → (I ≤ (⍴M)[1]) / L1
[6]    UNTIE 1
    ∇


       )Q
```

APPENDIX A

GLOSSARY

array
: a value with any number of dimensions. Generally the terms scalar, vector, and matrix will be used when possible; the term _array_ covers _all_ values.

composite operation
: a class of operations whose result is defined in terms of repeated applications of one or two scalar operators to one or two arguments. The shape of the result is defined by the particular composite operation, not the scalar operator(s). The four composite operations are inner product, outer product, reduction, and scan.

diamond line
: any number of statements, including zero, each separated by a diamond. Contains no label. See "line."

dyadic
: taking two arguments - a right argument and a left argument.

explicit result
: a value created as the primary (perhaps sole) result of evaluating an operator or function or pseudo-operator.

explicit subexpression
: an expression that is contained in a larger expression and is delimited by matching parenthesis. Short for "explicitly delimited subexpression."

expression

a valid combination of APL symbols that produces one explicit result and any number (including zero) of implicit results.

external function

a program that is separately compiled and exists outside the active APL workspace. It has a name and a syntactic usage description in the workspace, but is actually a program written in PL/I.

function

a sequence of one or more function lines, together with a header, that defines a stored APL program. The header defines its name and syntactic usage.

function line

a diamond line that can be preceded by a label.

implicit result

a value or effect created as a secondary (perhaps sole) result of evaluating a pseudo-operator or function. For example, the explicit result of [(specification) is its right argument. Its implicit result is the assignment of the right argument to the name given as the left argument.

implicit subexpression

an expression that is contained in a larger expression and is delimited by the syntactic rules of APL, and the right-to-left order of evaluation of APL. Short for "implicitly delimited subexpression."

line

a single statement. Contains no diamonds nor a label.

matrix

a value with two dimensions.

mixed operator

a class of operators that defines the shape of the result in terms of the individual operator and the particular shape of the argument or arguments. Membership, shape, and reshape are examples of mixed operators.

monadic
: taking one argument - always a right argument.

niladic
: taking no arguments.

operation
: any APL construct that can take one or more arguments. The following are operations: composite operations, external functions, functions, indexing, mixed output, operators, pseudo-operators, system functions. Even though they take no arguments, niladic functions are considered operations as well.

operator
: a builtin (not user-definable) APL construct that takes one or two arguments and returns one explicit result and no implicit results. An operator is always represented by a single graphic symbol. See mixed operator and scalar operator.

pseudo-operator
: a builtin (not user-definable) APL construct that takes one or two arguments, may return an explicit result, and may have an implicit result. Specification and execute are examples of pseudo-operators.

rank
: the number of dimensions in a value.

scalar
: a value with no (zero) dimensions.

scalar operator
: a class of operators that defines the shape of the result solely in terms of the shape of the argument or arguments, not the particular operator. In particular, the result of applying such an operator to non-scalar arguments is an extension of the result when applied to scalar arguments. Signum, add, and equal are examples of scalar operators.

shape
: (as a noun) the vector of dimension extents of a value - the number of dimensions (rank) together with the length of each dimension. Also the name of a monadic mixed operator represented by the rho R symbol.

statement
: an expression that is not contained in any other expression; i.e. an expression that is on a line by itself, or an expression that is between diamonds.

stop pseudo-variable
: see trace pseudo-variable.

subexpression
: an expression that is contained in a larger expression.

system function
: a builtin (not user-definable) function, with a reserved name, that is always present in the active workspace. Lfx is an example of a system function.

trace pseudo-variable
: a variable associated with each user-defined function that specifies a (possibly empty) vector of line numbers. Pseudo-variables can appear in the same contexts as normal variables, except for indexed assignment and localization. Assigning a value to a stop or trace pseudo-variable causes an implicit effect when the function is executed, namely, the stopping or tracing of lines of that function.

type
: the representation of a value; either character or numeric. A given value is either one or the other; the two cannot be mixed.

value
: a collection of any number (including zero) of character or numeric elements. A value is completely characterized by its type, shape, and list of elements.

vector
: a value with one dimension.

APPENDIX B

COMMANDS

This appendix gives descriptions of the Multics commands that relate to APL.

Name: apl, v2apl

Invokes the APL interpreter, optionally loading a saved workspace.

Usage

```
apl {workspace_id} {-control_args}
```

where:

1. workspace_id
   is the pathname of a saved workspace to be loaded. The default is to load the user's continue workspace, if any, otherwise to provide a clear workspace.

2. control_args
   may be chosen from the following:

   -terminal_type STR, -ttp STR
   specifies the kind of terminal being used. Possible values of STR are:

1050	CORR2741
2741	LA36
1030	TEK4013
ARDS	TEK4015
ASCII	TELERAY11
BITPAIRED	TN300
	TYPEPAIRED

   This control argument specifies which one of several character translation tables is to be used by APL when reading or writing to the terminal. Since there are several different kinds of APL terminals, each incompatible with the rest, it is important that the correct table be used. Specifying a terminal type to APL changes the terminal type only as long as APL is active. The default depends on the user's existing terminal type (refer to the set_tty command, in Multics Commands and Active Functions manual (Order No.: AG92)). These terminal types default to the same APL terminal type: 1050, 2741, CORR2741, ARDS, TN300, TEK4013, TEK4015, ASCII, LA36, TELERAY11. All other terminal types default to ASCII. The APL terminal types BITPAIRED and TYPEPAIRED are generic terminal types that can be used with any APL/ASCII terminal of the appropriate type.

   -brief_errors, -bfe
   causes APL to print short error messages. This is the default.

-long_errors, -lge
: causes APL to print long error messages. The short form of the message is printed, followed by a more detailed explanation of the error.

-user_number N
: sets the APL user number (returned by some APL functions) to N. The default is 100.

-check, -ck
: causes a compatibility error to occur if a monadic transpose of rank greater than 2, or a residue or encode with a negative left argument is encountered. (The definition of these cases is different in Version 2 APL from Version 1 APL).

-debug, -db
: causes APL to call the listener (cu_$cl) upon system errors. This puts the user at a new command level. The default is to remain in APL. This control argument is intended for debugging apl itself.

-no_quit_handler, -nqh
: causes APL to ignore the quit condition. The default is to trap all quits within APL.

-temp_dir path, -td path
: changes the directory that is used to hold the temporary segments that contain the active workspace to path. The default is to use the process directory.

Note

This command invokes the Version 2 APL interpreter, which replaces the obsolete Version 1 APL interpreter.

Name: apl_end

Resets the user's terminal environment to the normal Multics environment, removing all of the special attachments and translations for APL that are put in effect by the apl_start command.

Usage

```
     apl_end
```

Notes

See the apl_start command for a description of the APL terminal environment.

Name: apl_start

Sets up the user's terminal environment as it is when running APL.

Usage

```
apl_start {-control_arg}
```

where:

1. control_arg
   may be -terminal_type STR, -ttp STR to specify the kind of terminal being used. Possible values of STR are:

1050	CORR2741
2741	LA36
1030	TEK4013
ARDS	TEK4015
ASCII	TELERAY11
BITPAIRED	TN300
	TYPEPAIRED

   This control argument specifies which one of several character translation tables is to be used by APL when reading or writing to the terminal. Since there are several different kinds of APL terminals, each incompatible with the rest, it is important that the correct table be used. Specifying a terminal type to APL changes the terminal type only as long as APL is active. The default depends on the user's existing terminal type (refer to the set_tty command, in the Multics Commands and Active Functions manual (Order No.: AG92). These terminal types default to the same APL terminal type: 1050, 2741, CORR2741, ARDS, TN300, TEK4013, TEK4015, ASCII, LA36, TELERAY11. All other terminal types default to ASCII. The APL terminal types BITPAIRED and TYPEPAIRED are generic terminal types that can be used with any APL/ASCII terminal of the appropriate type.

Notes

The apl_start command is used to set up the user's terminal environment as it is during an APL session without actually invoking the APL interpreter. After invoking apl_start, the appropriate APL character set translations for the user's terminal type will be in effect, including the use of an APL graphic character set if the terminal has one.

When interacting with Multics, the following translation rule is used: the APL letters are translated into Multics lower case letters, and the underscored APL letters are translated into Multics uppercase letters, both on input and output.

The apl_start command is particularly useful for preparing exec_com or absentee input segments that wish to execute APL expressions. See also the description of apl_end.

Name: convert_tsoapl_workspace, ctw

Converts a TSO APL saved workspace, as read into a Multics segment by read_tsoapl_tape, into a Multics APL workspace. Only global names and values are converted; the state indicator (SI) is not converted.

Usage

```
     convert_tsoapl_workspace path {newpath}
```

where:

1. path
   is the pathname of the segment to be converted. The suffix ".sv.tsoapl" is assumed.

2. newpath
   is the pathname of the segment in which to place the converted workspace. The default is to create the segment in the working directory with the same entryname as the TSO APL workspace, but with the suffix ".sv.apl".

Name: display_tsoapl_workspace, dtw

Lists the contents of a saved TSO APL workspace read into a Multics segment by read_tsoapl_tape. The names and values of all global objects are displayed.

Usage

```
     display_tsoapl_workspace path {-control_arg}
```

where:

1. path
   is the pathname of the segment containing the workspace to be displayed. The suffix .sv.tsoapl is assumed.

2. control_arg
   may be

   -long, -lg
   list the workspace system variables (DIGITS, WIDTH, etc.) as well as the user defined objects. The default is to list only the user's objects.

Name: read_tsoapl_tape, rtt

Reads an APL SELDUMP tape, such as those created by APLUTIL on TSO, placing the saved APL workspaces on the tape into segments in the working directory. The segments are given the names of the saved workspaces, with the suffix ".libN.sv.tsoapl", where N is the library number of the workspace.

Usage

```
     read_tsoapl_tape tapeid {filename1 ... filenamen}
        {-control_args}
```

where:

1. tapeid
   is the tape slot number of the tape.

2. file_namei
   is the name of a workspace to be read from the tape. The default is to read all workspaces on the tape.

3. control_args
   may be chosen from the following:

   -attach_description STR, -atd STR
   Use STR as the attach description. The default attach description is "tape_nstd_ tapename -bk 10000".

   -density N, -den N
   Use N as the density setting. N must be 200, 556, 800, or 1600. The default is 1600.

   -list, -ls
   List the names of the workspaces on the tape, without reading the workspaces into segments. The default is to list and read the workspaces.

Notes

The user must have rw permission on the segment >sc1>rcp>workspace.acs, in order to get larger than normal tape buffers (tapes created by APLUTIL have 10,000 byte records, which is larger than the default buffer size).

Since Multics permits only ASCII characters in segment names, any delta characters in the workspace name are translated to "d", and any underlined delta characters are translated to "D".

INDEX

A

accounting information 7-5

add, subtract, multiply, divide 3-15

APL
  APL external functions 10-1
  character set 1-2, 2-2
  communicating with Multics 5-18
  file sharing 6-9
  file system 6-9
  history of 1-1
  Multics file system 6-1
  organization of files 6-1
  sample programs 11-1
  stream I/O 9-1
  system functions 7-1
  system variables 8-1
  terminal I/O conventions 2-8
  use of files 6-2
  values 3-1

apl command 2-1, B-2

apl_push_stack_ function 5-26

ASCII terminals 2-10

assignment pseudo-operator 3-69

ATTN processing 2-12

B

binomial coefficients 3-17

C

calling APL 2-1

canonical representation 7-8

catenate 3-23

ceiling and floor 3-18

character set 1-2, 7-9

circular 3-21

closed expressions 3-64

commands B-1
  execute 7-11

comments 3-63

comparison operators 3-19

comparison tolerance 8-2

composite operations 3-49

compress 3-42

deal 3-44

decode 3-44

delay 7-10

diamond lines 3-65

E

EBCDIC terminals 2-9

editing a line 4-28

encode 3-44

environment parameters 5-1

erase processing 2-11

error handling 4-18

errors 4-19

escape processing 2-13

execute pseudo-operator 3-75

execution flow 4-7

execution termination 4-17

expand 3-43

expressions 3-54
  implicit expressions 3-56
  right-to-left rule 3-55
  subexpressions 3-55

expunge 7-12

F

factorial 3-17

fix 7-13

format 3-47

function calls
  pendent 4-21
  suspended 4-21

functions 3-68, 4-1
  active 7-3
  arguments 4-1
  call 7-6
  editing a 4-26
  editing halted functions 4-23
  external 5-19, 5-20, 5-21, 5-2
    5-25
    apl_erf_ 10-2
    apl_get_list_nums_ 10-5
    apl_ioa_ 10-4
    apl_pickup_float_bin_2_ 10-3
    apl_read_segment_ 10-6
  file manipulation 6-2
  locked 4-20
  restarting a suspended function
    4-23
  results 4-1
  stream I/O 9-1

functions calls
  halted 4-21

G

general rules for catenate 3-28

general rules for laminate 3-35

generating algorithm 3-19

glossary A-1

grade up, grade down 3-39

groups 3-68

I

I-Beam 3-46

identifiers 3-66

index generator 3-37

indexing 3-59, 3-70

inner product 3-54

input
  character 3-74
  evaluated 3-72

input line processing 2-10
  ATTN 2-11
  canonicalization 2-11
  erase 2-11
  kill 2-11

input prompt 2-2

internal codes 2-2

interrupts 4-17
  strong 4-18
  weak 4-17

K

kill processing 2-11

L

labels 3-63

laminate 3-31

line counter 7-15

lists 3-58

locked functions 4-16

logical operators 3-20

M

magnitude 3-17

matrices and arrays 3-6
  character 3-8
  matrix divide 3-45
  matrix inverse 3-45
  numeric 3-8

matrix
  divide 3-45
  inverse 3-45

maximum and minimum 3-18

membership 3-43

mixed operators 3-21

mixed output 3-61

name count 7-16

name list 7-18

names 3-65, 3-66
  function 3-67
  group 3-67
  syntax of 3-66
  variable 3-66

O

objects 3-65

operator
  dyadic scalar 3-12
  mixed 3-11
  monadic scalar 3-11

operators 3-11
  scalar 3-11

outer product 3-53

overstrikes 2-8

P

parameters
  environment 5-1
  session 5-2
  workspace 5-2

PI times 3-21

plus, negative 3-15

pornography 3-76

power, logarithm 3-16

R

ravel 3-23

reciprocal 3-16

recursion 4-12

reduction 3-50

reshape 3-22

residue 3-16

results
  explicit 3-71
  implicit 3-70, 4-19
  output of explicit 3-72

reverse 3-40

roll 3-19

rotate 3-40

S

scan 3-52

session parameters 5-2

shape 3-22

SI
  clearing 4-23
  damage 4-24
  exploring 4-22

signum 3-16

state indicator 4-16

statements 3-63

stop pseudo-variables 4-19

suspension 4-18

symbol table 5-5

system commands 5-1

T

tabulating 2-9

take, drop 3-38

terminal I/O conventions 2-8
  ASCII terminals 2-10
  EBCDIC terminals 2-9
  input line processing 2-10
  overstrikes 2-8
  tabulating 2-9

terminal type 7-21

time stamp 7-20

transpose 3-41

U

user load 7-22

using APL 2-13

V

values 3-1
  elements 3-3
  output of 3-3
    scalars 3-4
      scientific notation 3-4
  rank 3-2
  shape 3-2
    operator 3-3
  type 3-1

variables 3-67

vectors 3-5
  character 3-5
  numeric 3-6

W

workspace 3-79

workspace (cont)
  active 3-80
  available 7-23
  identification 5-10
  management 5-10
  parameters 5-2
  passwords 5-11
  used 7-24

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	MULTICS APL USER'S GUIDE	ORDER NO.	AK95-02
		DATED	DECEMBER 1985

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

PLEASE FOLD AND TAPE–
NOTE: U.S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

44840, 5C286, Printed in U.S.A.

AK95-02