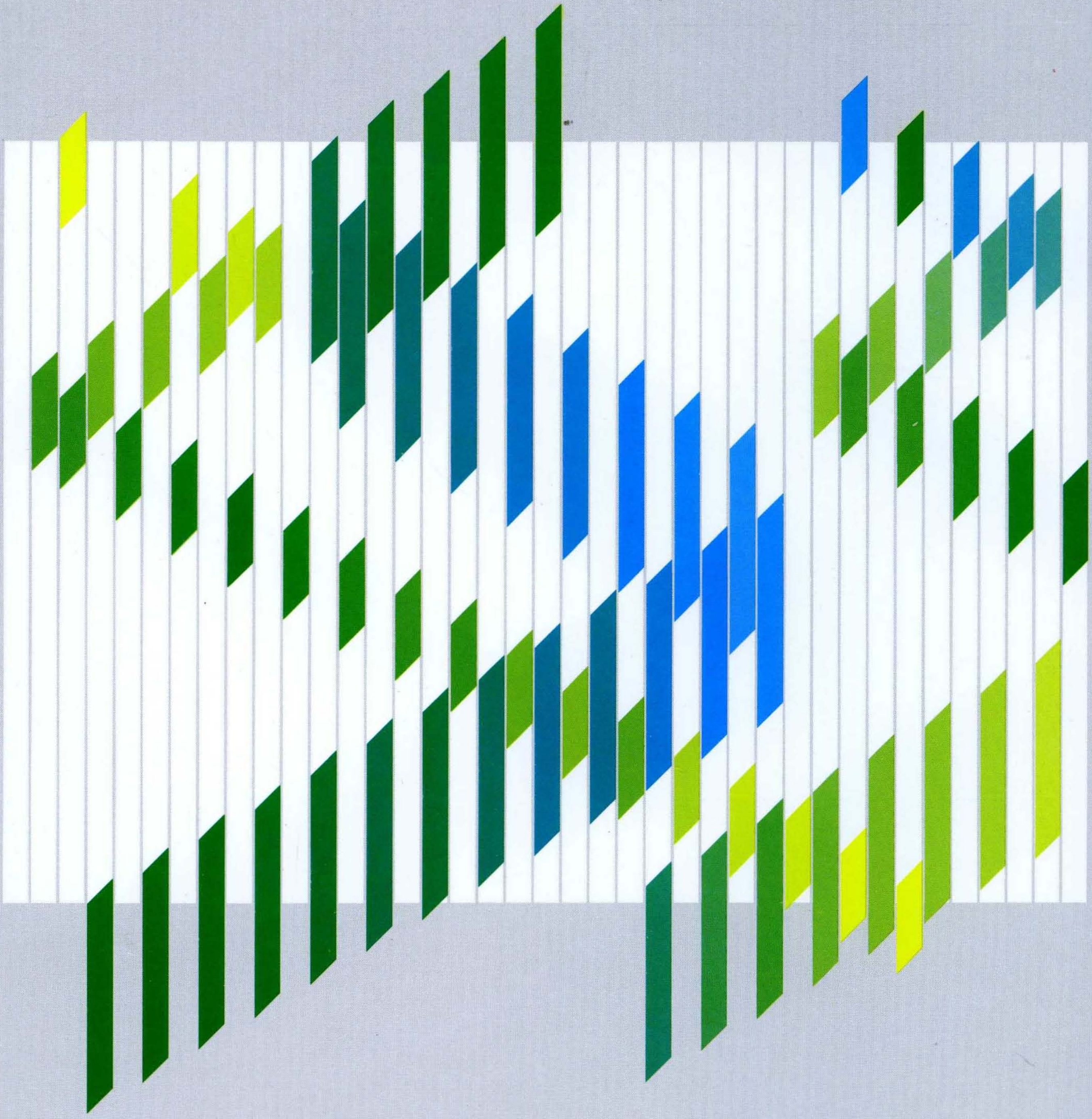
IBM

An Introduction to APL2

Program Product



Order Number:
SH20-9229-1

Program Number:
5668-899

Release Number:
2

An Introduction to APL2

Program Number 5668-899
Release 2

SH20-9229-1

Second Edition (December 1985)

This is a major revision of, and makes obsolete, SH20-9229-0.

This edition applies to Release 2 of APL2, Program Product 5668-899, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Amendments" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any subsequent republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370 and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

The APL2 system puts an advanced computing system within the reach of a wide range of users. APL2 is distinguished from earlier systems by its power and by the radical simplicity of the instructions that control it. This combination makes APL2 well suited not only to the advanced scientific or technical user and professional programmer, but also to the occasional user and to the user with little or no previous experience with computers.

This manual is intended to provide an introduction to the APL2 system. It will show you the mechanics of using the system, and how to write effective programs to cover a wide range of applications. It explains in detail many points that the experienced user will find obvious — and you may therefore prefer to skip some portions. But be aware that APL2 has a wide range of new features that were not available in previous versions of APL. Even the advanced APL users in the audience might find it helpful to review the fundamental sections.

This book makes no attempt to exhaustively define all of the capabilities of APL; it mentions only a few of the specialized applications that are possible using APL.

It is strongly recommended that you supplement your reading of this manual with a copy of *APL2 Programming: Language Reference*, which offers complete formal definitions of all of the operations in the APL language and all of the features of the APL2 system. In this manual, we're concerned with providing you with a basic orientation to the way the system is used, and arming you with the fundamental skills needed to make APL2 work effectively for you.

Related Publications

GH20-9214	*APL2 General Information*[1]
SH20-9215	*APL2 Migration Guide*
SH20-9216	*APL2 Programming: Guide*
SH20-9217	*APL2 Programming: Using Structured Query Language (SQL)*
SH20-9218	*APL2 Programming: System Services Reference*
SH20-9220	*APL2 Messages and Codes*
SH20-9221	*APL2 Installation and Customization under CMS*
SH20-9222	*APL2 Installation and Customization under TSO*
SX26-3737	*APL2 Reference Summary*
SX26-3738	*APL2 Reference Card*
SY26-3931	*APL2 Diagnosis Guide*
SY26-3932	*APL2 Diagnosis Reference*
SH20-9230	*APL2 GRAPHPAK: User's Guide and Reference*
SH20-9227	*APL2 Programming: Language Reference*
SH20-9233	*APL2 Programming: Using the Supplied Workspaces*

Whereas there is no prerequisite publication to this Introduction manual, it is recommended that you have a copy of *APL2 Programming: Language Reference* available while reading this manual.

More advanced topics in APL application programming are covered in *APL2 Programming: Guide.*

[1] *APL2 General Information* contains a description of each of the other books in the APL2 Language Library.

APL2 Program Product Library

Task	APL2 Publications
Evaluation	General Information Licensed Program Specifications
Installation	Installation and Customization under CMS Installation and Customization under TSO
Migration	Migration Guide
Reference	Language Reference System Services Reference Reference Summary Reference Card
Programming	An Introduction to APL2 Programming Guide Using Structured Query Language (SQL) APL2 GRAPHPAK: User's Guide and Reference Using the Supplied Workspaces
Diagnosis	Messages and Codes Diagnosis Guide Diagnosis Reference

Summary of Amendments

Release 2, December 1985

Enhancements

- Calls to Routines Written in Other Languages

  Two associated processors are provided which allow routines written in FORTRAN, Assembler, or REXX to be called from APL2 applications.

- Access to System Editors

  The *)EDITOR* command has been extended to allow either a TSO CLIST, or a CMS command or EXEC to be named. This CMS command or EXEC, or TSO CLIST will be executed to invoke a system editor such as XEDIT or ISPF when APL2 editing is requested.

- APL2 Language Enhancements

  - Vector specification has been included to improve clarity and programmer productivity.

  - A new system function, ⎕*EC*, has been added to allow controlled execution of APL expressions. This facility further enhances the error handling facilities introduced in APL2 Release 1.

  - APL2 character set restrictions have been relaxed to allow a larger number of acceptable characters in literals and comments and to allow the use of lowercase alphabetics in APL2 names.

- APL2 Workspace Enhancements

  - A new workspace, *GDMX*, is provided to assist users in developing applications which interface with GDDM.

  - Improvements have been made to the SQL workspace to support new function in AP 127, greater productivity in implementing SQL-based applications, and interface to the GDDM Interactive Chart Utility for graphic display of relational data.

Miscellaneous Improvements

A substantial number of improvements have been made to APL2 in order to supplement existing function and to improve usability, performance, reliability, and error reporting. Among these are:

1. AP 127 has been improved by adding support for the SQL/DS CONNECT command and for retrieval of DB2 message text.

2. Controlled invocation of APL2 has been added to allow APL2 applications to be invoked from other applications without intervening APL2 output or screen display.

3. In the MVS/XA environment, global shared storage is now placed in extended CSA thus alleviating storage constraints and allowing larger global shared storage sizes.

4. TCAM is no longer supported in the MVS/TSO environment.

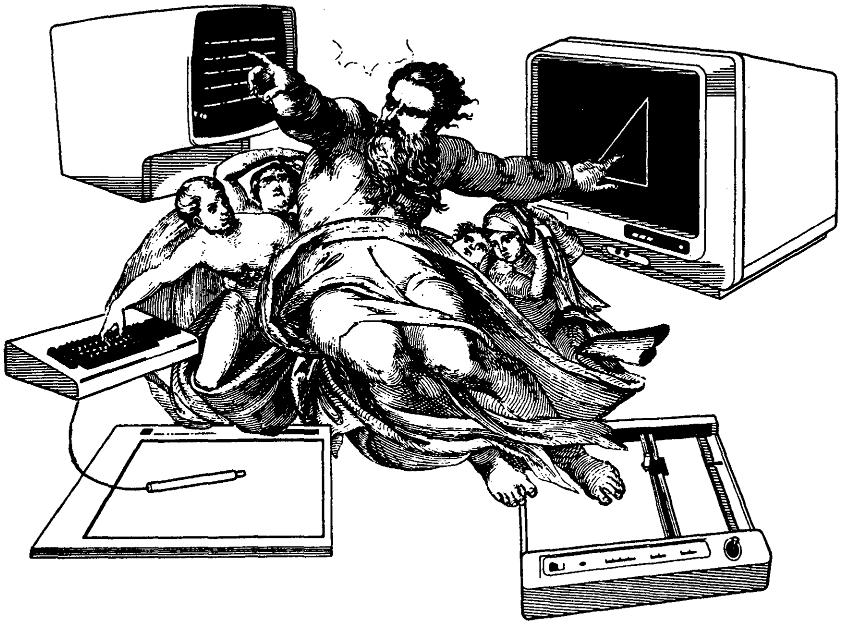
5. VM/SP Release 3 or later is now a prerequisite for APL2 Release 2 in the VM/CMS environment.

Contents

Chapter 1: Getting the Flavor of Things

If you're new to APL, this chapter may provide some understanding of what APL is all about; we'll discuss the syntax and characteristics of the language.

If you're *not* new to APL, we *still* recommend following through this chapter. APL2 offers many capabilities that were not present in other versions of APL. Reading this chapter may acquaint you with some of the new power of the language. We'll introduce some new terms here, and give you some examples of what can be done.



Used with the permission of The Dick Sutphen Studio.

APL — What Is It?

Introduction

APL is a general-purpose language that enjoys extensive use in such diverse applications as commercial data processing, system design, mathematical and scientific computation, and the teaching of mathematics and other subjects. It has proved to be particularly useful in data base applications, where its computational power and communication facilities combine to enhance the productivity of both application programmers and end users.

When implemented as a computing system, APL is used from a typewriter-like keyboard. Statements that specify the work to be done are entered by typing them, and in response, the computer displays the result of the computation. The result appears at a device that accompanies the keyboard, such as video display or printer. In addition to work that is performed purely at the keyboard and its associated display, entries may also invoke the use of printers, disk-files, tapes or other remote devices.

The letters APL originated with the initials of a book written by K. E. Iverson, *A Programming Language* (New York: Wiley, 1962). Dr. Iverson first worked on the language at Harvard University, and then continued its development at IBM with the collaboration of A. D. Falkoff and others. The term *APL* now refers to the language that is an outgrowth of this work.

APL2 is a particular implementation of that language with extensions that have been developed within IBM over the last several years. The treatment of nested arrays in APL2 is based on J. A. Brown's dissertation and T. More's theory of arrays.[2]

[2] J. A. Brown, *A Generalization of APL*, Ph.D. Dissertation, 1971, Dept. of Computer and Information Science, Syracuse University, Syracuse, New York, Clearing House 74h004942 AD-770488./5.

T. More, "A Theory of Arrays with Applications to Databases," IBM Cambridge Scientific Center report G320-2106, Sept. 1975.

Power, Relevance, and Simplicity

A programming language should be relevant. That is, you should have to write only what is logically necessary to specify the job you want done. This may seem an obvious point, but many of the earlier programming languages would have forced you to be concerned as much with the internal requirements of the machine as with your own statement of your problem. APL2 takes care of those internal considerations automatically.

A programming language needs both power and simplicity. By power, we mean the ability to handle large or complicated tasks. By simplicity, we mean the ability to state what must be done briefly and neatly, in a way that is easy to read and easy to write. You might think that power and simplicity are competing requirements, so that if you have one you can't have the other, but that is not necessarily so. Simplicity does not mean that the computer is confined to doing simple tasks, but that the user has a simple way to write his instructions to the computer. The power of APL as a programming language comes in part from its simplicity; it is this simplicity that makes it simultaneously well suited to the beginner and to the advanced user.

A Short Example of the Use of APL

If the work to be done can be adequately specified simply by keying a statement made up of numbers and symbols, names will not be required; simply typing in the expression to be evaluated causes the result to be displayed. Let's try out an example:

> Many bacteria can duplicate themselves once every half hour. If a single infectious organism began reproducing at 9 o'clock in the morning, how fast would the resultant colony grow?

```
      2*2×3 12 16
64 16777216 4294967296
```

Our keyboard input

The system's response

At noon (three hours later), we would have a colony of 64 members...

At nine o'clock that same night (12 hours later), we would have over 16 *million* newcomers around...

By 1 o'clock the next morning (16 hours later), we would be greeted by four *billion* new offspring!

Several distinctive features of APL are illustrated in this example: familiar symbols, such as "×", are used where possible, other symbols are introduced where necessary (such as the "*" for the power function), and (***very*** important!) *a group of numbers can be worked on together.*

The Characteristics of APL

The primitive objects of the language are arrays (lists, tables, lists of tables, and so forth). For example, $A+B$ is meaningful for any arrays A and B.

The syntax is simple. There is no hierarchy of function precedence, and built-in functions and user-defined functions (programs) are treated alike.

The rules of "programming grammar" are few. The definitions of the built-in functions are independent of the type of data to which they apply, and they have no hidden side effects.

The sequence control is simple. One statement type embraces all types of branches (conditional, unconditional, or computed), and the termination of the execution of any function always returns control to the point of use.

External communication is established by means of data that is directly shared between APL and other systems or subsystems. These *shared variables* are treated both syntactically and semantically like other data. A subclass, called *system variables*, provides convenient communication between APL programs and their environment.

The utility of the built-in functions, called *primitive functions*, is vastly enhanced by *operators* which modify their behavior in a systematic manner. For example, *reduction* (denoted by "/") modifies a function to apply over all elements of a list, as in $+/L$ for summation of the elements of L. Axis specification allows functions like reduction to be applied to a table in a specified direction. In addition, APL allows you to "roll-your-own"; that is, both functions and operators may be user-defined for your own needs.

The number of primitive functions is small enough that each is represented by a single easily-read and easily-written symbol, yet the set of primitives embraces operations from simple addition to a complex form of grading (sorting) and formatting.

Used with the permission of Hart Publishing Company, Inc.

Getting Started in APL

Fundamentals

APL Is Interactive

The APL system takes one APL expression at a time, converts it to "machine instructions" (the computer's internal language), executes it, and then proceeds to the next line. This is in contrast to traditional program compilers which convert complete programs to machine language before executing any expressions. This allows you a high degree of interaction with the computer. If something that you enter is invalid, you will get quick feedback on the problem before you proceed further.

Who Typed What?

During an APL terminal session, you and APL will take turns using the terminal. While you type information in, APL waits for some signal from you that it is *its* turn to use the terminal for displaying the results from your input. This signal from you is the depression of the "CARRIAGE-RETURN" or "ENTER" or "EXECUTE" key—the name of the key differs between different types of terminals, but the action is the same: it's simply a means of telling APL that you have finished typing a line, and that you're ready for APL to evaluate that line.

When APL displays information for you, it starts each new line at the left margin. After it finishes displaying any such output, it signals to you that it is ready for you to type in another keyboard input by indenting six spaces from the left margin and halting. This position is the indication that it's ready for you to take "your turn." For example,

```
      2+2 ◄──────────  You typed this in...
4 ◄──────────────────  and APL responded with this.
      AREA             The six-position indent indicated
12                     that you could enter something else.
```

Expressions

A typical *expression* in APL is of the form:

```
AREA←3×4
```

The effect of the statement is to assign to the *name* "*AREA*" the value that is the result of 3×4 to the right of the *assignment arrow*, "←"; it may be read informally as "area *is* three times four."

If the leftmost part of an expression is not a name followed by an assignment arrow, the result of the expression is displayed.[3] For example:

```
      3×4
12
      PERIMETER←2×(3+4)
      PERIMETER
14
```

Displaying any *intermediate result* in an APL expression can be obtained by including the characters "⎕←" after any portion of the expression which would produce an intermediate result. Moreover, any number of assignment arrows may occur in an expression. For example:

```
      A←2+⎕←3×B←4
12
      A
14
      B
4
```

You may also assign a set of names from the items of a *vector*:

```
      (A B)←14 4
      A
14
```

This is called *vector assignment*.

[3] The "leftmost" part of the expression is significant here, because APL's order of evaluation is right-to-left. The leftmost part of the line, therefore, is the last part to be evaluated. But we'll get to the order of evaluation rules a little later on.

The terminal entry and display devices used with APL systems include a variety of typewriter-like and display-tube devices. Their characteristics vary, but the essential common characteristics are:

1. The ability to enter and display APL characters.

2. A means of signalling completion (and release to the system) of an entry.

3. Facilities for convenient revision of an entry before release.

4. Facilities to interrupt execution at the end of an expression (*attention*) and within an expression (*interrupt*).

5. A *cursor* (some form of pointer) to show where on the line the next character entered will appear.

All examples in this manual are presented as they would appear on a typewriter-like device. Even though video terminals are in very common use now, this typewriter-like presentation is done because the characters commonly used on some of the typewriter-like terminals are distinctive, and you can easily differentiate them from the explanatory text. The text of this manual, therefore, is entered in upright characters, and *ITALIC CAPITALS* will be used to indicate the portions of the examples that you might actually see at your terminal (even though *your* terminal might display upright block characters instead of the italics that we use here).

On those typewriter-like terminals, the release signal is produced by the carrier return key, and revision is handled by backspacing to the point of revision, striking the ATTENTION button, and entering the revision. An inverted caret supplied by the system marks the point of correction. For example:

```
      3+4×5+
         ∨
         +5+6
18
```

On terminals of this type the ATTENTION key is also used for interrupting execution. A single strike of this key while execution is in progress provides an attention signal, and a double strike provides an interrupt: an attention says "stop when it's convenient," and an interrupt says "stop immediately."

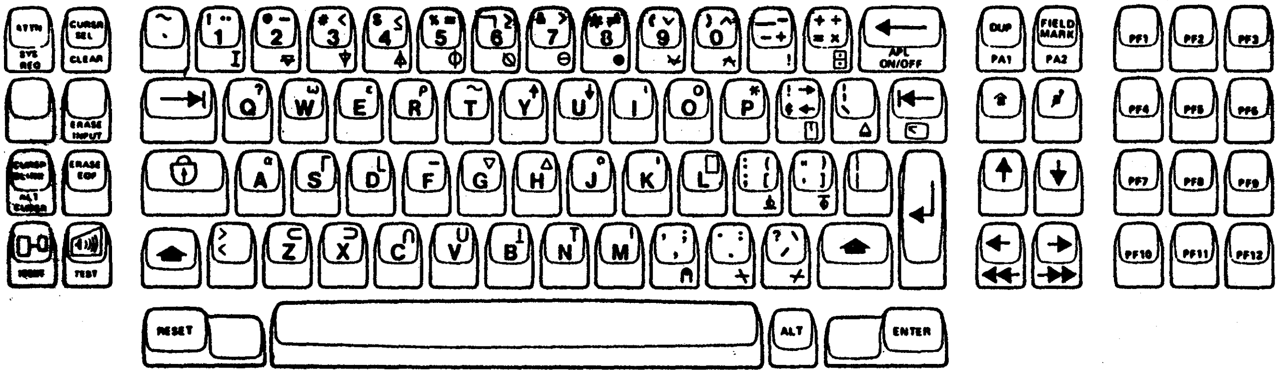
On some video display terminals, such as those in the IBM 3270 and 3290-series, typing corrections may be entered by simply backspacing the cursor onto the portion of the line to be corrected, and typing over the line. The "ERASE EOF" key (meaning "erase to end of field") on the left side of the keyboard will delete everything from the point where the cursor is positioned through the end of the current line.

Attention and interrupt is signalled on the 3270 terminals by depressing the PA2 key, once for attention and twice for interrupt.

The APL Character Set

The characters that may occur in a statement fall into four main classes: alphabetic, numeric, special, and blank. The alphabetics are composed of the roman alphabet in uppercase italic font, the same alphabet underscored, plus "∆", and "⍙". The entire set of alphabetics is shown under the discussion of "Names" on page 12. The entire set of displayable characters that are supported in APL2 are shown in the chart on page 11 along with suggested names and the scheme for forming (as composites of other symbols) those characters that may not be directly available on the keys of some terminals.

A Typical APL Keyboard



IBM 3278/3279 Keyboard

The APL2 Character Set

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A̲ B̲ C̲ D̲ E̲ F̲ G̲ H̲ I̲ J̲ K̲ L̲ M̲ N̲ O̲ P̲ Q̲ R̲ S̲ T̲ U̲ V̲ W̲ X̲ Y̲ Z̲
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9

¨	dieresis	⍺	alpha[1]	⍱	down caret tilde	∨ ~
¯	overbar	⌈	up stile	⍲	up caret tilde	∧ ~
<	less	⌊	down stile	⍒	del stile	∇ |
≤	not greater	_	underbar	⍋	delta stile	∆ |
=	equal	∇	del	⍙	delta underbar	∆ _
≥	not less	∆	delta	⌽	circle stile	○ |
>	greater	∘	jot	⍉	circle slope	○ \
≠	not equal	'	quote	⊖	circle bar	○ -
∨	down caret	⎕	quad	⍟	circle star	○ *
∧	up caret	(	left paren	⌶	down tack up tack[1]	⊥ ⊤
-	bar	)	right paren	⍫	del tilde	∇ ~
÷	divide	[	left bracket	⍎	down tack jot	⊥ ∘
+	plus	]	right bracket	⍕	up tack jot	⊤ ∘
×	times	⊂	left shoe	⍀	slope bar	\ -
?	query	⊃	right shoe	⌿	slash bar	/ -
⍵	omega[1]	∩	up shoe	⍝	up shoe jot	∩ ∘
∊	epsilon	∪	down shoe	⍞	quad quote	⎕ '
⍴	rho	⊥	down tack	!	quote dot	. '
~	tilde	⊤	up tack	⌹	quad divide	⎕ ÷
↑	up arrow	|	stile	⍂	quad slope[1]	⎕ \
↓	down arrow	;	semicolon	⌻	quad jot[1]	⎕ ∘
⍳	iota	:	colon	⌷	left bracket right bracket[1]	[ ]
○	circle	,	comma	≡	equal underbar	= _
*	star	.	dot	⍷	epsilon underbar	∊ _
→	right arrow	\	slope	⍸	iota underbar[1]	⍳ _
←	left arrow	/	slash	⍤	dieresis dot[1]	@ .
			blank (space)	%	percent[1] [2]	/ ÷
				&	ampersand[1] [2]	| ∊
				¢	cent[1] [2]	| ⊂
				$	dollar[1] [2]	*S* /
				#	pound[1] [2]	*N* =
				@	at[1] [2]	*Q* ∘
				!	exclamation[1] [2]	' ∘
				|	vertical bar[1] [2]	| ⊥
				~	tilde[1] [2]	| ~
				¬	not[1] [2]	/ ~
				¦	split bar[1] [2]	, '
				"	double quote[1] [2]	¨ '
				{	left brace[1] [2]	- (
				}	right brace[1] [2]	-)
				\	backslash[1] [2]	\ |
				`	accent[1] [2]	\ ¯

Note: The lowercase alphabetics (“*a*” through “*z*”) may be typed as “*A*” overstruck with “¯” through “*Z*” overstruck with “¯”, respectively.

All overstrike combinations may be entered in either order.

[1] These characters have no assigned purpose, other than use as decorators.

[2] National-use characters may have alternate graphics in different countries, although they do ***not*** have alternate overstrikes.

The names suggested are for the symbols themselves and not necessarily for the functions they represent. For example, the *down stile*, "⌊", represents both the *minimum*, a function of two arguments, and the *floor* (or *integer part*), a function of one argument. In general, most of the special characters (such as +, -, ×, and ÷) are used to denote *primitive functions* which are assigned fixed meanings, and the alphabetic characters are used to form *names* which may be assigned and reassigned significance as user-defined variables, defined functions and operators, and labels. The blank serves as a separator to mark divisions between names (which are of arbitrary length).

Any available display font (or character set) may be used for your APL terminal session, as long as your terminal permits the display font to be changed without changing the behavior of the entry keyboard or communication with the system — as, for instance, in changing the typing element on certain typewriters. For example, in textual work a font with normal upper- and lowercase roman is commonly employed.

Names

Valid characters for forming names are:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ∆

A̲ B̲ C̲ D̲ E̲ F̲ G̲ H̲ I̲ J̲ K̲ L̲ M̲ N̲ O̲ P̲ Q̲ R̲ S̲ T̲ U̲ V̲ W̲ X̲ Y̲ Z̲ ⍙

0 1 2 3 4 5 6 7 8 9 ¯ _ These cannot *start* a name

With certain settings of the CASE parameters, lowercase letters are used in place of underscored letters. In this book, only the underscored letters are used.

Names of workspaces, functions, variables, operators, and labels may be formed of any sequence of the above characters, as long as they contain no blanks, and don't start with a numeric digit, or with the characters "¯" or "_". For example,

```
A             )
ABC̲           |
SALES_REPORT  }  — are all valid names
TAX1984       |
∆             )

A B              Invalid name — contains a space
1984TAX          Invalid name — starts with a numeric
_REPORT          Invalid name — starts with "_"
DATA.3           Invalid name — "." isn't allowed
```

The environment in which APL operations take place is bounded by the active workspace (described in Chapter 3). Hence, the same name may be used to designate different objects (that is, variables, functions, operators, and labels) in different workspaces, without interference. Also, because workspaces themselves are never the subject of APL operations, but only of system commands, it is possible for a workspace to have the same name as an object it holds.

A workspace name is limited to a length that is governed by the particular type of system upon which you're running. A typical workspace name-length limit is eight characters. The names of variables, functions, operators, and labels, however, may be of *any* desired length. Any length of name that you choose is retained by the system and is significant.

Numbers

All numbers entered or displayed are in decimal, either in conventional form (including a decimal point if appropriate) or in "scaled form." The scaled form consists of an integer or decimal fraction called the *multiplier* followed immediately by an "*E*" and then by an integer (which must not include a decimal point) called the *scale*. The scale specifies the power of ten by which the multiplier is to be multiplied. Thus `1.44E2` is equivalent to `144`.

In a similar fashion, APL accepts complex numbers with a "*J*" separating the real and imaginary parts. Optionally, a polar form is available, with the angle expressed in either radians or degrees. For example, the square root of negative one may be entered as `0J1` in its standard form, as `0R1.570796327` in polar radian form, and as `1D90` in polar degree form. Complex numbers are always displayed using the *J* form.

Negative numbers are represented by an overbar immediately preceding the number. For example, `¯1.44` and `¯144E¯2` are equivalent negative numbers. The overbar can be used as part of a numeric constant and is distinguished from the bar that denotes negation, as in `-X`. The overbar may *not* be used to denote negation of a value stored under a name; that is, "`¯X`" is invalid.

Functions

The word "function" derives from a word that means to execute or to perform. A *function* executes some action on its *argument* (or arguments) to produce a result that may serve as an argument to another function. For example:

```
      3×4
12
      2+(3×4)
14
      (-6)÷3
¯2
```

Functions represented by symbols, such as "+", "-", "×" and "÷", are called *primitive functions*, because they are "primitive" to the system; that is, they are automatically available for use in any workspace without having to copy them from somewhere. Functions may also be user-defined and given names.

A function that takes one argument (such as the negation used above) is said to be *monadic*, and a function that takes two arguments (such as the *times* function) is said to be *dyadic*. All APL functions are either monadic or dyadic or, in the case of defined functions only, may also be *niladic* (taking no argument). With both primitive and user-defined functions, the same symbol or name can represent both monadic and dyadic functions. For example, `X-Y` denotes *subtraction* of `Y` from `X` (a dyadic function), and `-Y` denotes *negation* of `Y` (a monadic function).

Operators

The normal operation of a function may be altered by applying an *operator* to it. For example, "+" and "×" are primitive functions; applying the "/" operator to produce "+/" and "×/" modifies their normal operation in a precise, defined manner, and produces a new, *derived function*. Operators apply equally to user-defined functions, and, in fact, the operators themselves may be user-defined.

Terminology: Functions versus Operators

Over the years, there has sometimes been confusion between the terms "function" and "operator." The terms have sometimes been used interchangeably. In APL, it's useful to differentiate the terms.

A *function* is that which takes in one or more *data* objects (or "arguments") and returns new *data* (result). An example of a *monadic* (single-argument) function is: "⍳"... "⍳3" takes in one piece of data (the argument "3"), and returns new data in the form of the result, "1 2 3". An example of a *dyadic* (two-argument) function is "+"... "2+3" takes in two arguments and returns new data in the form of the result, "5".

An *operator* is that which takes in one or more data objects *or functions* and returns a new *function*. An example of a symbol that can be used as an operator is "/"... "+/" takes in a function (plus) and returns a new *derived* function; in this case, sum reduction. The new derived function, then, acts like any other function. It takes in data (for example, "+/1 2 3") and returns new data ("6").

Both functions and operators may be user-defined. Later discussions in this manual will discuss both of those constructions.

For some in-depth discussions on functions and operators, see *APL2 Programming: Language Reference*. And finally, if you're looking for some all-inclusive generic term for those funny APL squiggles, call them "symbols" (or even squiggles, if you must)—but *not always* operators, please.

Data

Data used in APL is one of two types: either *numeric* or *character*. Data is produced by:

- Explicit entry at the keyboard,
- Execution of APL functions and operators, or
- Use of shared variables and system variables (described later in this manual).

Arrays

APL functions apply to collections of individual data items called *arrays*. An array is an ordered collection of items[4] arranged along rectangular dimensions (called *axes*), where these items are numbers, characters, or other arrays.

Rank and Shape

The *rank* of an APL array is the number of dimensions or axes that it has. For those of you who may already be familiar with some other computer languages, you may be thinking of the term *dimension* as the amount of data that be stored; that's *not* what we mean here. We are referring to the axes, not the *length* of the data. For our purposes, a *dimension* and an *axis* are synonymous. When we want to refer to the amount of data *along* each of those dimensions, that's what we will call *shape*.

For example, a simple list of numbers has only one dimension — only length — and therefore is of rank one:

```
      V←2 3 5 7 11 13 17 19
      V
2 3 5 7 11 13 17 19
```

Any array may contain both numbers and characters:

```
      V←2 3 'A' 4 'B' 5
      V
2 3 A 4 B 5
```

In APL, data in a list form like this is referred to as a *vector*.

An example of a rank-two object would be a table of numbers:

```
      M← 2 5 ρ ι10
      M
1 2 3 4  5
6 7 8 9 10
```

(We'll talk about ρ and ι in a moment)

In APL, two-dimensional data like this is referred to as a *matrix*. Either of these examples could just as easily have used character data, or a mixture of numeric and character data.

[4] These "items" are often conversationally referred to as the "elements" that make up the array; these two terms mean the same thing.

A *scalar* has no dimensions and is of *rank zero.* Arrays range from these dimensionless scalars to multidimensional arrays of arbitrary rank and size. Here are the three most commonly-used ranks:

Rank	APL Name	Equivalent to a
0	Scalar	Point
1	Vector	Line or list
2	Matrix	Table

The *vector* is a simple form of array which may be formed by listing its elements. For example:

```
      V←2 3 5 7 11 13 17 19

      A←'A' 'B' 'C' 'D' 'E' 'F'
```

or

```
      A←'ABCDEF'
```

The *shape* of an array may be measured by using the *shape* function, denoted by the "rho" (ρ) symbol:

```
      V
2 3 5 7 11 13 17 19

      ρV
8

      A
ABCDEF

      ρA
6
```

The shape function returns a count of the number of items along each of the dimensions. In the case of those vectors, there was only one dimension; a matrix, because it is "two-dimensional," will return two numbers:

```
      N← 3 4 ρ ⍳12
      N
1  2  3  4
5  6  7  8
9 10 11 12

      ρN
3 4

      ρM
2 5
```

We showed an example above of how a vector is entered at the terminal, but a matrix cannot be directly entered. You'll have to use a function to tell APL the shape that you want. A matrix is commonly formed by listing the items of data that the matrix is to contain, and then using the *reshape* function to create the desired shape. The reshape function uses the same rho-symbol that the shape function uses, but has a left argument stating the desired resultant shape. The matrix shown above, for instance, could be formed like this:

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

The number of numbers used to the left of the ρ-symbol determines the rank of the object being formed. Here, the two numbers "3 4" create a rank-two object — a matrix. In a similar fashion, the rank of an object may be measured by counting the number of numbers that are returned with the monadic use of the ρ-symbol... in other words, measuring the shape of the shape:

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

      ρN            Shape of N
3 4

      ρρN           Rank of N
2
```

The right argument for the reshape function may be in any form: it could be a directly-entered list of items as we discussed above, or it could be data already stored under a name:

```
      M←2 4ρV
      M
2  3  5  7
11 13 17 19
```

```
      B←2 4ρA
      B
ABCD
EFAB
```

Arrays of arbitrary shape and rank may be produced by the same scheme. For example:

```
      T←2 3 4ρ'ABCDEFGHIJKLMNOPQRSTUVWX'
      T
ABCD
EFGH
IJKL

MNOP
QRST
UVWX

      ρT
2 3 4
```

This three-dimensional array has two planes, each with three rows and four columns. Three-dimensional arrays display with a blank line separating the planes, and higher-dimensional arrays simply extend this scheme.

Variables

An array that is stored under a name is called a *variable*, because its value may be varied at any time simply by reassigning a new value to the name. All of the names that we have shown in this "Arrays" discussion ("*V*", "*A*", "*M*", and so forth) are variables.

Constants

A *constant* is a number or string of numbers or a character or string of characters that appears explicitly in an APL expression.

A single number entered by itself is accepted by the system as a *scalar*. A constant *vector* may be entered by listing the numeric components in order, separated by one or more spaces.

A scalar character constant may be entered by placing the character between quotation marks (as in `'A'`), and a character vector may be entered by listing the characters between quotation marks (as in `'THIS IS TEXT'`). The blanks are part of the data, and are treated like the other characters — that last example is twelve characters long. Such a vector is displayed by the system as the sequence of characters, with no enclosing quotes and with no separation of the successive elements (characters).

Quotes

The quote character itself must be entered as a pair of quotes. Thus, the contraction of *CANNOT* is entered as `'CAN''T'` ...APL displays it as *CAN'T*, and it consists of five characters.

Bracket Indexing

The elements of an array may be selected by *bracket indexing*. For example:

```
      V←2 3 5 7 11 13 17 19
      V[3 1 5]
5 2 11

      (2 3 5 7 11 13 17 19)[3 1 5]
5 2 11

      A←'ABCDEFGH'
      A[8 5 1 4]
HEAD

      'ABCDEFGH'[8 5 1 4]
HEAD
```

The numbers within the square brackets indicate the positions of the data that is being selected. If *any* of the indices are out of range, you'll get an error message:

```
      'ABCDEFGH'[8 5 1 35]
INDEX ERROR
      'ABCDEFGH'[8 5 1 35]
      ^         ^
```

Elements may be selected from any array (other than a scalar) by indexing in the manner shown for vectors, except that indices must be provided for each dimension:

```
      M[2;3]                 │       T[2;1;4]
17                           │P
      M[2 1;2 3 4]           │       T[2;1 2 3;1 2 3 4]
13 17 19                     │MNOP
 3  5  7                     │QRST
                             │UVWX
      ⍴M[2 1;2 3 4]          │       ⍴T[2;1 2 3;1 2 3 4]
2 3                          │3 4
```

Elements can't be selected from a scalar through bracket indexing, because a scalar has no dimensions (or axes) from which to select its data.

Index Origin

The indexing used in the foregoing examples is called *origin* 1 because the first element along each axis (or dimension) is selected by the index 1. You may also use *origin* 0 indexing by setting the *index origin* to 0. The index origin is controlled by a *system variable* denoted by ⎕IO. Thus:

```
      ⎕IO←1            │      ⎕IO←0

      V[1 2 3]         │      V[0 1 2]
2 3 5                  │2 3 5

      B[2;3]           │      B[1;2]
A                      │A

      ⍳3               │      ⍳3
1 2 3                  │0 1 2
```

This function is the
"Index Generator"

In APL, you always have the choice of using either origin 1 or origin 0. You may find that the use of origin 0 may make some applications easier to write. This is especially true where certain mathematical operations are being performed. Calculations involving number-base conversions, for example, are often cleaner if you're working in origin 0. Some indexing operations themselves are also a little cleaner. For example:

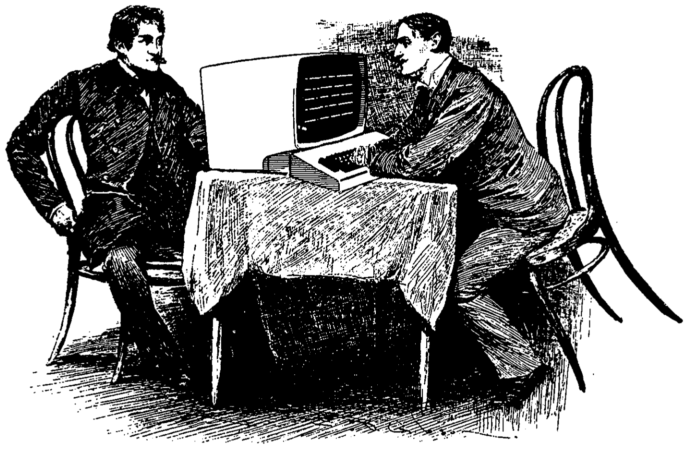
```
      ⎕IO←1        |      ⎕IO←0
                   |
      N            |      N
1  2  3  4         |1  2  3  4
5  6  7  8         |5  6  7  8
9 10 11 12         |9 10 11 12
                   |
      '∘⎕'[1+N>6]  |      '∘⎕'[N>6]
∘∘∘∘               |∘∘∘∘
∘∘⎕⎕               |∘∘⎕⎕
⎕⎕⎕⎕               |⎕⎕⎕⎕
```

However, origin 0 can also be confusing at times, simply because most of us grew up being accustomed to thinking of a series of numbers as starting with one instead of zero. [Neither of these is correct, of course; in our hearts we all know that the number series really begins at negative infinity.] But years of seeing lists numbered "1, 2, 3" instead of "0, 1, 2" tends to leave its mark. Throughout our lives we have been taught that:

0. House numbers start with 1 (rarely with 0)
1. Magazine pages so often tend to start with 1
2. Days of the month start with 1 (it's *really* hard to find an exception here)

So, rather than complicating your life by bucking this ingrained bias, APL uses origin 1 as its default; you can always change it, but that's what you'll see when you first sign on.

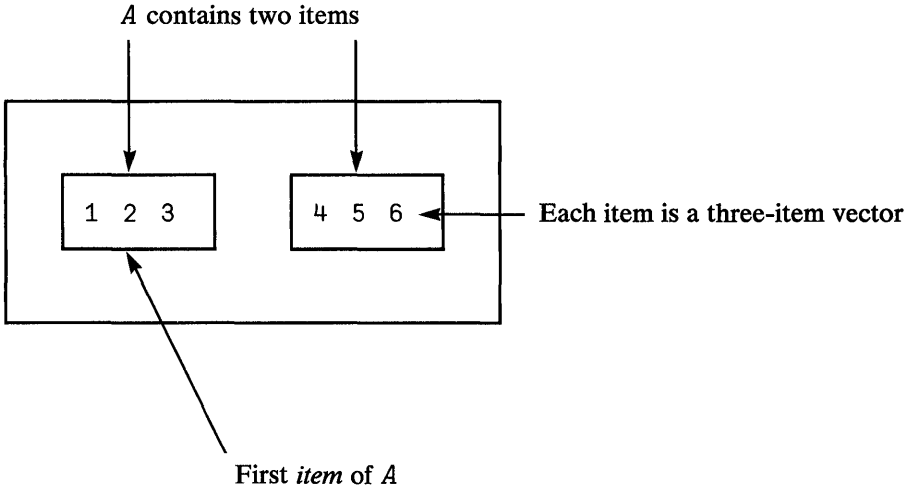
Because of this default, ***all further examples in this manual will be shown in origin 1 unless otherwise stated.***



Used with the permission of Hart Publishing Company, Inc.

Some Terminology for Adding More Structure to Arrays

Let's assume that we have an array named "*A*", which contains two pieces of data: a string of numeric data having the value "1 2 3", and a similar string of numeric data having the value "4 5 6". "*A*" then can be represented as a *two-item* vector. (We're keeping this discussion separate from the discussion of the particular APL notation that we would use to *form* such a vector — that will come a bit later.) For example,



In APL2, an item of an array can be any other array.

For more information, see the section on "Understanding Arrays," following on pages 33-45.

Spaces

The *blank character* is used as a separator. (A "blank" is the character, and "space" is its result.) The spaces that one or more blank characters produce are needed to separate names of adjacent defined functions, constants, and variables. For example, if *F* is a defined function, then the expression 3 *F* 4 must be entered with the indicated spaces. The exact number of spaces used in succession is of no importance, and extra spaces may be used freely. Spaces are not required between primitive functions and constants or variables, or between a succession of primitive functions, but they may be used if desired. For example, the expression 3+4 may be entered with no spaces.

Remember that example in the previous section, though: "¯144*E*¯2" is a single numeric constant, because APL recognizes the "*E*" as indicating "exponential form," whereas "¯144 *E* ¯2" would attempt to combine a user-defined function or variable named "*E*" with two numbers. In this particular context, the spaces are significant.

Whenspacesarerequired (and when they are n't even allow ed)

Ofallofthekeysonthekeyboard, perhaps the space is the most important.[5] Even the layout of the keyboard reflects its importance.

The uses of the space can be divided into two categories:

1. The space is used as a literal blank character, to *visually* separate words or numbers.

2. The space is used to separate objects that would take on a different meaning if they weren't separated (for example, the vector 2 3 is quite different from the value 23).

When used inside of quotes, the space is treated just like any other character. Spaces within comments are left just where you enter them; APL doesn't touch them at all. And, now that comments may peacefully co-exist on the same line as APL code, the *leading* blanks between the code and the comment are significant and are retained. See page 102 for some additional comments on comments.

Other than in character constants and comments, the space is used to ensure proper syntax of the expression. Its only purpose is to allow the juxtaposition[6] of objects on the same line. When they're used that way, *multiple spaces act exactly the same as a single space.*

Consider the expression "3-2". No spaces are needed here, because the objects "3", "-", and "2" cannot be confused with another single object when put directly next to each other — they simply form an APL expression. If spaces are inserted the meaning does not change: "3 - 2", or even "   3   -   2 ".

However, given a dyadic function *MINUS*, then "3*MINUS*2" is invalid. It would have to be written as "3 *MINUS* 2", with one or more spaces separating "3" from "*MINUS*" and "*MINUS*" from "2". Here, the spaces separate the *arguments of a defined function* from the *name of that function.*

Let's consider another example: The expression "-4 5" contains three objects, "-", "4", and "5". The space is used here so that the two numbers "4" and "5" aren't confused with the number "45". The addition of *extra* spaces won't change the expression at all. Therefore, "- 4 5", " - 4   5   ", and "   -   4   5   ", are all the same as "-4 5". Here, the space is used to separate the items of a vector.

The space is used to separate objects that are juxtaposed. If the objects do not create visual ambiguities when they're put directly together, then the space is unnecessary. (Parentheses can also be used to separate juxtaposed items.) The objects that can merge are names, numeric constants, and character constants. Perhaps it would be useful to look at all combinations of the juxtaposition of these objects to see the possibilities.

[5] See?

[6] "Juxtaposition" refers to the positioning of terms side-by-side in an expression. The expression "*A* *B*" shows two names in close proximity or "juxtaposition."

Here are examples of these various possibilities, along with what would be produced if the space were removed:

Example, with a space	Meaning *with* the space	Same example but without the space	Meaning *without* the space
NAME NAME	Two separate names	*NAMENAME*	A longer (and different) name
NAME 3	One name and one number	*NAME*3	A longer (and different) name
NAME '*TEXT*'	One name and a character vector	*NAME*'*TEXT*'	(no change)
2 *NAME*	A number and a name	2*NAME*	*ERROR*
2 3	A two-item numeric vector	23	A different numeric scalar
2 '*TEXT*'	A number and a character vector	2'*TEXT*'	(no change)
'*TEXT*' *NAME*	A character vector and a name	'*TEXT*'*NAME*	(no change)
'*TEXT*' 3	A character vector and a number	'*TEXT*'3	(no change)
'*TEXT*' '*TEXT*'	Two character vectors	'*TEXT*''*TEXT*'	A single character vector containing a quote character

Parentheses may be used to group and separate objects. Given a defined function called "*REPORT*", and a variable called "*SALES*", either "*REPORT SALES*" or "*REPORT*(*SALES*)" are valid. Spaces are not needed in the second example, because the parentheses themselves separate the names.

As long as objects are separated from each other, the semantic rules of APL can take over to give meaning to expressions.

Therefore:

- A space that's not part of a character constant or a comment is used to separate juxtaposed objects.
- When used to separate objects, extra spaces have no ill effects.
- Parentheses may be used instead of spaces either to separate objects or to group objects.

Parentheses

Parentheses are used in the familiar way to control the order of evaluation in a statement. Any expression within matching parentheses is evaluated before applying any function to the result outside the matching pair. Parentheses are always permissible if they are properly paired and what is inside evaluates to an array, a function, or an operator.

In conventional notation, the order of evaluation of an unparenthesized sequence of monadic functions may be stated as follows: the (right-hand) argument of any function is the value of the entire expression to the right. For example, in conventional notation Log Sin Arctan x means the Log of Sin Arctan x, which means Log of Sin of Arctan x. In APL, the same rule applies to dyadic functions as well. Moreover, all functions, both primitive and defined, are treated alike; there is no hierarchy among functions (such as multiplication being done before addition or subtraction).

An equivalent statement of this rule is that an unparenthesized expression is evaluated in order from right to left. For example, the expression `3×8⌈3*|5-7` is equivalent to `3×(8⌈(3*(|(5-7))))`. The result of each expression is `27`. Parentheses are often used to surround the left argument of a function, so that it is evaluated in one complete piece. For example, `(12÷3)×2` is `8` and `12÷3×2` is `2`. However, redundant pairs of parentheses can be used at will. Thus, `12÷(3×2)` is also `2`.

Here are other ways that parentheses are used to modify the order of evaluation:

1. They *group* pieces of an expression or data.

2. They *separate* pieces of an expression or data from other such pieces.

Any parentheses that do not *both* group and separate are redundant and may be eliminated without altering the meaning of the expression. For example:

`(2 3)` These parentheses *group* the values 2 and 3 together, but they don't separate them from anything, so they're redundant. The expression could be restated as "2 3" without altering the meaning of the expression. This is a numeric vector of length two.

`(2) (3)` These parentheses *separate* the values 2 and 3 from each other, but they don't group anything, so they are also redundant. The expression could also be restated as "2 3" without altering the meaning of the expression. This is also a numeric vector of length two and is equivalent to the previous one.

`(2 3) (4 5)` These parentheses both *group* the first and second pairs of values together, and *separate* the pairs from each other. The parentheses are *not* redundant here; they are needed. Moving or eliding any of the parentheses would alter the meaning of the expression. This is a "nested" array of length two, containing two two-element numeric vectors.

Order of Evaluation

In APL, the order of evaluation is always from right to left for functions, except as modified by the use of parentheses. In particular, there is *no* hierarchy among the functions (such as multiplication being executed before addition, and so forth). All functions are treated alike. The reason for this is simple: if we're dealing with only a half-dozen or so functions, the rules of hierarchy are straightforward. But in APL, the large number of functions would make such hierarchical rules very cumbersome and difficult to remember. So the rules are concise. The order of evaluation is *right-to-left* for any function.

Pairs of parentheses are used in APL in exactly the same way that they are used in conventional mathematics, or for that matter, the same way that they are used in most other computer languages. When parentheses are encountered during the right-to-left evaluation of the input line, the normal order of execution is interrupted, and expressions within the parentheses are evaluated first (also in a right-to-left fashion). Then the original right-to-left scan is continued.

APL will always try to use any function in a *dyadic* sense if it can. It will scan from right to left until it encounters a function, continue (looking for a left argument, so that it can use the function dyadically if possible) until it comes to *another function* (or to the left end of the expression — whichever comes first). It will then *back up* one position, and evaluate everything to the right. It will then begin scanning to the left again, using *that result* as the right argument for the next function, or as the final result if there are no other functions.

It should be seen from those rules that any function takes *everything* to its right as its right argument.

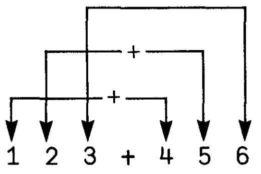
For example,

```
      3×4+5
27
      (3×4)+5
17
      ¯1+(1 2 3×4)+5 6 7
8 13 18
```

Let's examine APL's evaluation rules a little further. When you enter an expression into APL, here is how the system would evaluate it, using the rules stated on the preceding page. The representation here is *not* what you would see at the terminal, but rather, a sort of "inside view" of the workings of APL. As bits of the input line are evaluated, we'll reprint the line, and underscore the portion of the line that APL is looking at, print the *intermediate result* at that point, and then continue on through the line, using that intermediate result as the right argument for the next function.

	$2+2\ \ 2+2$	Here's the expression that we'll try out first. Be careful; follow all of the rules here!
2	$2+2\ \ 2+\underline{2}$	First pass — examine the right argument for the first function, the rightmost "+" function.
2 2	$2+\underline{2\ \ 2}+2$	Look to see if that function, "+", has a left argument, trying to use the function in a dyadic manner if possible; that's *always* the preferred usage.
4 4	$2+\underline{2\ \ 2+2}$	Keep going until you see the *second* "+" function, then you *back up one*, and evaluate everything to the right. Now use the first result, "4 4", as the right argument for the next function (the leftmost "+" function), such that the operation becomes "2+ 4 4".
2	$\underline{2}+2\ \ 2+2$	Continue scanning to the left, until we come to another function (or to the end of the statement); that will be the left argument for this function.
6 6	$\underline{2+2\ \ 2+2}$	Now add that left argument to the *intermediate result* that we got from the previous addition. And there's our final result.

Some functions, like the "+" in the previous example, treat data in an item-by-item manner. These are examples of the dyadic scalar functions:



(We'll discuss scalar functions in more detail later on. There will be a table of them, coming up on page 37.)

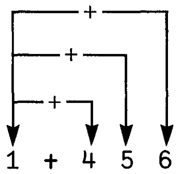
That's equivalent to:

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

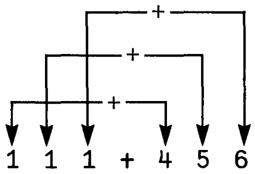
or

```
            5 7 9
```

If one side of the expression contains a scalar,[7] that side is (logically) replicated to match the rank and shape of the other side; that's "*scalar extension:*"



...is
equivalent
to...



or

```
      (1+4) (1+5) (1+6)
```

or

```
            5 6 7
```

Errors

Entry of a statement that cannot be executed will invoke an *error report*. Newcomers to APL often tend to worry needlessly about typing inputs that result in errors. These error reports are some of the most ***helpful*** aids that you could ask for toward learning the language. APL error reports are designed to be clear, concise, and precise.

As opposed to doing everything that you could do to prevent generating errors, it may be helpful to deliberately try out many of the error conditions. This is a good way of learning how various functions are defined. Learning by doing is always preferable. And don't worry that you may enter something that you shouldn't have... nothing that you can enter can hurt the machine. This gives you full freedom to experiment.

An APL error report indicates the nature of the error and displays carets, indicating both where the error occurred and where the execution halted. For example:

```
      B←1 2 3 + A←4 5
LENGTH ERROR
      B←1 2 3+A←4 5
       ^    ^
```

[7] For convenience, a relaxation of the rules sometimes permits a one-element vector to be used as though it were a scalar.

There's a wealth of information available from these error reports. Let's see just what this message is telling us:

Here is the line that you typed in.

```
      B←1 2 3 + A←4 5
LENGTH ERROR
      B←1 2 3+A←4 5
       ^     ^
```

"*LENGTH ERROR*" says that the lengths of the two arguments to this function don't match, so it isn't clear which number is to be added to which other number.

APL reprints your input line, so that you can verify that it read it properly (occasionally a bad telephone connection will garble things during transmission). Extraneous blanks will be removed.

There will typically be two carets under the line of code. The ***left caret*** shows you how far APL got in its right-to-left scan of the line (here, the assignment of a value to *A* has been done, but the assignment to *B* has *not* yet been done). The ***right caret*** shows you the point of the actual error. Normally, that will indicate which function APL was evaluating when the error occurred. In this example, the arguments to the "+" function aren't compatible with each other, so the requested addition can't be performed.

We'll examine error reports in more detail later on, in the discussion of "Display of Errors" on pages 161-168.

Chapter 2: How To Use Some of the Pieces

Here are detailed descriptions of some of the new facilities of APL2. New functions are described, and some hints are included to help you get going in the new directions. If you are relatively experienced with APL, then this chapter may be helpful. This chapter does assume that you have at least a moderate familiarity with the language.

Understanding Arrays

The data structures in APL2 may be reviewed as follows:

- A single number or character is an "array."
- An array *A* is a collection of zero or more other arrays, called the "items" of *A*. These items are ordered along "n" directions, called "axes" or "dimensions."
- The number of axes or dimensions that an array has is called its "rank."

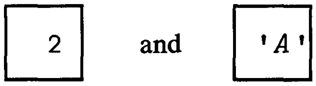
The following names apply to arrays:

	APL	**Equivalent**
Rank	**Name**	**to a**
0	Scalar	Point
1	Vector	Line or list
2	Matrix	Table

For this discussion, we will use a box notation to show the outermost structure of an array, and a linear notation to indicate an array as part of an APL expression.

Scalars

A *scalar* will be shown as a plain box:

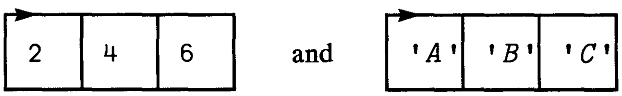


These arrays have one item (a single number and a single character, respectively) arranged along zero axes (no axes). When writing an expression, a single number is entered and displayed in its decimal representation, which may in general contain more than one digit. A single character is entered as that single character surrounded by single quotes. It is displayed by the system without the quotes. Those two scalars in the above example would be entered like this:

```
2    and   'A'
```

Vectors

A *vector* will be shown as a string of boxes with a single arrow on the top edge denoting the single axis of a vector:



These arrays have three items arranged along one axis. When writing an expression, a vector is entered by writing down each of the scalar values separated by a space. The two vectors in the example above would be entered like this:

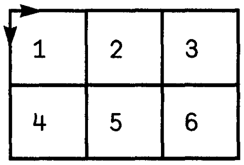
```
2 4 6   and    'A'  'B'  'C'
```

For conservation of symbols, a vector of single characters may be written with a single pair of enclosing quotes, like this:

```
'ABC'
```

Matrices

A *matrix* will be shown as a rectangular arrangement of boxes with *two* arrows, on the top and left edges, denoting the two axes of a matrix (notice that the vector picture had only one arrow):



1	2	3
4	5	6

This array has six items (each one a single number) arranged along two axes.

There is no linear form for writing a matrix constant as part of an expression. Rather, the "reshape" function (ρ) is used with a left argument giving the shape of the array, and the right argument giving a list of items. For example, the array shown in the previous example may be written like this:

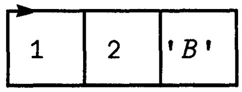
```
2 3 ρ 1 2 3 4 5 6
```

If *all* of the items of an array are single numbers or characters, then the array is called *simple*. Each of the arrays pictured above is simple. The set of simple arrays which are made up of only numbers or only characters are the arrays traditionally part of APL. Such arrays are used as items in further examples without further explanation.

The definition of array given above includes two extensions over the arrays of traditional APL. These extensions are "mixed arrays" and "nested arrays."

Mixed Arrays

Numbers and characters may appear in the same array. That's called a "mixed array." For example:



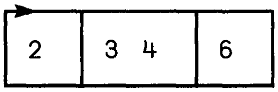
1	2	'B'

is an array (a vector) which contains two numbers and one character as items. It may be entered like this:

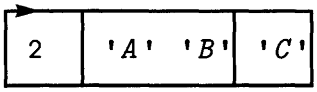
```
1 2 'B'
```

Nested Arrays

An item of an array may be another array. For example:



and



These arrays each have three items. The first and last items are simple scalars, and the center item is a length-two vector. Such arrays are called *nested arrays.* They are sometimes also called “nonsimple arrays.”; the terms are synonymous. These arrays may be entered like this:

```
2 (3 4) 6      and      2 ('A' 'B') 'C'
```

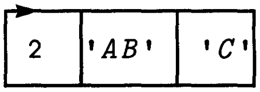
where the parentheses are used for grouping. Because the center item of the character example is made up of single characters, it may be written with a single pair of quotes. Thus, the example could also be written like this:

```
2 ('AB') 'C'
```

And, because the quotes already imply a grouping, the parentheses aren’t needed, and the array can be written like this:

```
2  'AB'  'C'
```

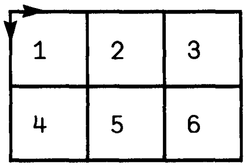
Those last two expressions are equivalent to the former expression, and may be shown pictorially like this:



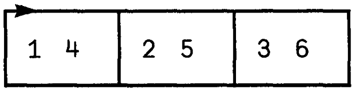
Enclose

A vector may be reshaped into a matrix by means of the reshape function (⍴), as shown in the previous example. In a similar way, the monadic function *enclose*, “⊂[*A*]”, may be used to transform a simple array into a nested array. *A* is a simple integer scalar or vector which identifies the axes of the argument used to form the items in the resultant array. The axes not mentioned define the outer structure of the result. For example, given this array with two rows and three columns:

⊂[1]

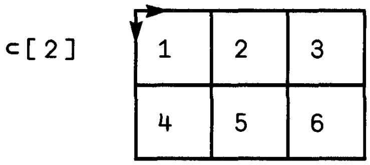


produces a vector whose items come from columns of the argument and whose outer structure is the left-over dimension “3”:

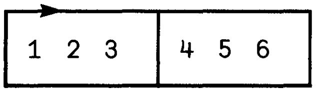


This is a *nested three-element vector of two-element vectors.*

Similarly applying enclose on the column axis gives an array whose items come from the rows of the argument leaving outer structure "2":

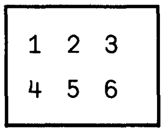


gives:



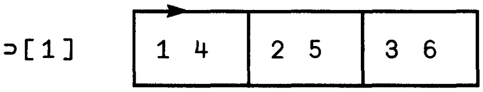
This is a *two-element vector of three-element vectors.*

Finally, ⊂[1 2] applied to the same matrix requests that both axes be used to make the items of the result leaving no axes for the outer structure — giving a scalar result:

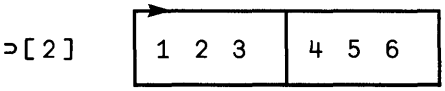


When *all* of the axes are to be enclosed, it is convenient to leave out the axes selection and the brackets. Therefore, the ⊂[1 2] of the above example could have been written as simply ⊂.

The *disclose* function (⊃[*A*]) is defined as the inverse to enclose, so:



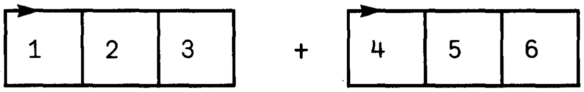
and:



both produce the original matrix as a result.

Scalar Functions

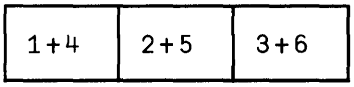
The "scalar" functions in APL are those functions which, when applied to scalars, produce scalars, and which extend to higher-rank arrays and nested arrays in an item-by-item manner. This can be pictured like this:



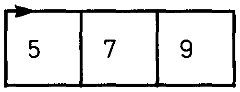
and written:

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

Applying "+" item-by-item gives:

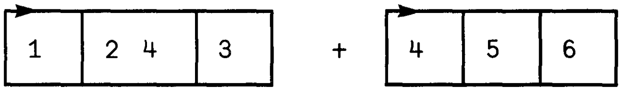


which evaluates to:



Note that there is an implicit requirement that the lengths of the arguments must match. However, if one argument is a scalar,[8] it is extended to be the same shape as the nonscalar argument.

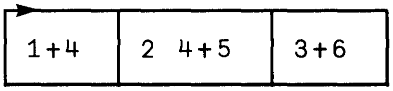
If the arguments are *not* simple arrays, the analysis still holds. For example:



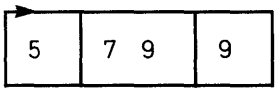
may be written:

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

Applying "+" item-by-item gives:



which evaluates to:



This implies a recursive[9] use of the definition of scalar functions on arrays. In more deeply nested arrays, this recursion persists until simple scalars are reached. This property of a function is called *pervasiveness*.

[8] For convenience, a relaxation of the rules sometimes permits a one-element vector to be used as though it were a scalar.

[9] By "recursive" we mean in this context applying the function to the outer structure,

Our examples have shown the use of "+" as a scalar function; there are, of course, many other functions that we could have used for the examples. Here's the entire set of scalar functions in APL2:

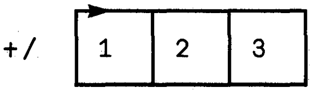
Monadic Scalar	**Function Symbol**	**Dyadic Scalar**
Conjugate	+	Add
Negative	-	Subtract
Direction	×	Multiply
Reciprocal	÷	Divide
Magnitude	|	Residue
Floor	⌊	Minimum
Ceiling	⌈	Maximum
Exponential	*	Power
Natural Log	⍟	Logarithm
Pi Times	○	Circular
Factorial	!	Binomial
Not	~	*{Nonscalar Function}*
Roll	?	*{Nonscalar Function}*
	∧	And
	∨	Or
	⍲	Nand
	⍱	Nor
	<	Less
	≤	Not Greater
	=	Equal
	≥	Not Less
	>	Greater
	≠	Not Equal

Note: All dyadic forms may take an axis.

For more information on these functions, refer to APL2 Programming: Language Reference.

Reduction

An *operator* in APL is applied to a *function* to produce a related *derived* function. The monadic operator "reduce" (/) may be applied to a dyadic scalar function, producing a monadic function (called a derived function), which is then applied between the items of its argument. For example:



then going inside the structure and performing the same operation on the inner structure, and then going inside *that* structure, and so forth, until we reach the bottom level (the "simple scalars").

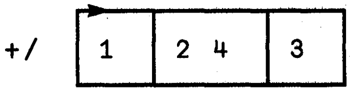
gives:

```
1 + 2 + 3
```

which evaluates to:

```
6
```

Notice that *the reduction of a vector gives a scalar* (...that's why it's called "reduction"). This same analysis holds if the vector argument of reduction is nested. For example:



may be written:

```
+/ 1 (2 4) 3
```

and gives:

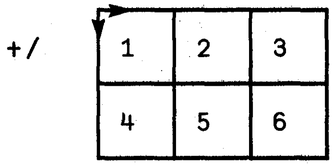
```
1 + 2 4 + 3
```

which evaluates to:

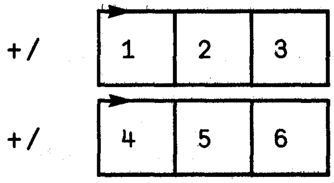
```
6 8
```

which is a *nonsimple scalar.*

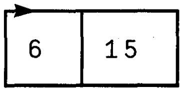
Reduction, when applied to arrays having more than one axis, implies a splitting of the array into vectors along one of the axes, and applying the reduction to each vector. Thus, *reduction reduces rank.* For example:



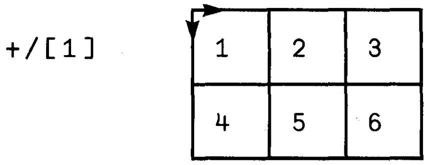
will split the array into vectors, like this:



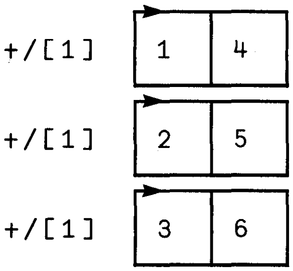
giving:



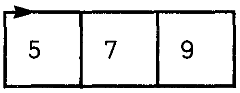
This example shows reduction applied to the second axis of a matrix. This could also be written as "+/[2]". Reduction applied to the first axis is written like this:



giving:



which evaluates to:



In general, for any rank n arrays, reduction causes the function to be applied to vectors. The rank of the result will be n-1.

The "+/" example showed summation applied to the *last* axis. In similar fashion, the "+/[1]" example showed summation applied to the *first* axis (in origin 1). A special short-hand notation is available for reduction, "+⌿", applying summation to the first axis (in either origin).

As you become familiar with APL, you'll learn to structure your data to take advantage of this ability to specify either the first or the last dimension without specifying a dimension number. This will frequently allow the same function to work on arrays of different rank without any change to your functions. Consider, for instance, a vector whose elements represent monthly sales or a matrix whose columns represent monthly sales and whose rows represent different products — "+/" gives the sum over time in both cases.

A further example of reduction is given on pages 43-44.

The "Each" Operator

A scalar function automatically extends to nonscalar arrays by applying the function to the items of its arguments, with each application being independent of the others. Such an item-by-item operation may be desired for nonscalar functions. For example, suppose that we were given the vectors "`2 3 2`", "`3 4 5`", and function "*f*" (where "*f*" represents any arbitrary function), and we wished to form this array:

2*f*3	3*f*4	2*f*5

If "*f*" is a scalar function, then this is what "*f*" gives by definition. Suppose that "*f*" were the reshape function (ρ). The expression:

```
2 3 2 ρ 3 4 5
```

would form a rank-three array — and that's *not* what we wanted. Therefore, the "each" operator (¨) is introduced, which, when applied to any function "*f*", produces a derived function "*f*¨" which applies "*f*" to the *items* of its arguments *independently.* That is to say, the each operator produces a scalar (but not necessarily pervasive) function. Now:

2	3	2	ρ¨	3	4	5

will produce:

2ρ3	3ρ4	2ρ5

giving:

3 3	4 4 4	5 5

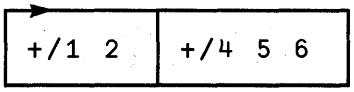
Finally, the each operator may be applied to the derived function that is the result of another operator. For example, suppose that we wanted to do a plus-reduction on each item of an array:

+/¨	1 2	4 5 6

which may be written:

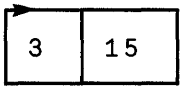
```
+/¨ (1 2) (4 5 6)
```

which becomes:



+/1 2	+/4 5 6

giving the result:



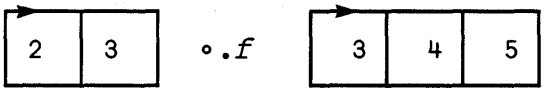
3	15

Because each of the above reductions gives a simple scalar result, the above is a simple vector.

No matter what function "*f*" represents, "*f*¨" is a scalar function. "*f*¨/", therefore, reduces rank, and (sometimes) increases depth. Therefore, "ρ¨/" would reduce rank, and you'd get back a nested array.

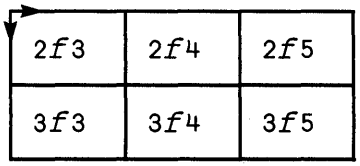
The "Outer Product" Operator

The "outer product" operator (∘.) is much like "each" except the function is applied between pairs of items, one from the left argument and one from the right argument, in *all combinations*. For example:



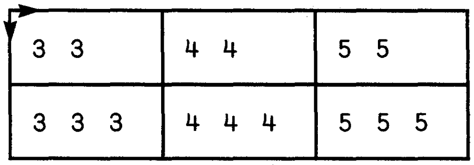
2	3	∘.*f*	3	4	5

will produce:



2*f*3	2*f*4	2*f*5
3*f*3	3*f*4	3*f*5

no matter what the function "*f*" is. If "*f*" is the function "reshape" (ρ) this gives:



3 3	4 4	5 5
3 3 3	4 4 4	5 5 5

In general, if the words "all combinations" occur in the problem description, you can expect to find an outer product in the solution.

* * *

An Example of the Use of Nested Arrays

Here's an example that shows the ability of nested arrays to represent data that is not so conveniently represented as a simple array.

Suppose that you want to keep track of your deposits and withdrawals for your bank account. You could do this with two variables; *D* for deposits and *W* for withdrawals, like this:

```
      D← 10 23 45
      W← 60 25
```

The total deposited is then "+/*D*", the total withdrawn is "+/*W*", and your total worth is "(+/*D*)-+/*W*". [Oh, oh... you're in trouble!]

This could be represented in a single variable:

```
      JAB←D W
```

This is a vector with "two items," where the first item represents the deposits, and the second item represents the withdrawals. Now the total deposited and withdrawn may be computed at once:

```
      +/¨ JAB
78 85
```

and, of course, your total worth is the difference between these two numbers:

```
      -/+/¨ JAB
¯7
```

[Not good.] Notice that this isn't any shorter than the original computation, and it's certainly doing computation on the same data as before — but it uses only one name!

Let's look at a computation involving more than two sets of data. Suppose that you are the bank. You *could* have one variable per bank account. The money on hand is then:

```
(-/+/¨JAB) + (-/+/¨JON) + (-/+/¨DAVE) +  ...
```

but this quickly gets out of hand. Instead, let's represent all of the accounts in a single vector:

```
      BANK← JAB JON DAVE  ...
```

This is a vector with one item per bank account. Each account is a two-element vector, just as before.

Now we can compute the total worth of each bank account *all at once* by applying the previous computation to each account:

```
      -/¨ +/¨¨ BANK
¯7 10 1000000  ...
```

There's an extra "each" on each function because in this case the accounts are more deeply nested.

The bank can compute *its* total worth by adding up the individual accounts, like this:

```
+/ -/¨ +/¨¨ BANK
```

This is *very* much shorter than the computation on many names. This had better not come out negative, but if it does, the bank can identify the offending accounts like this:

```
0 > -/¨ +/¨¨ BANK
```

Finally, the bank can also tell how many customers it has by simply entering:

```
ρBANK
```

Of course, you could represent withdrawals as negative numbers and pad accounts with zeros to the length of the most active account, and represent everything as a simple array. Doing it that way, some things would take more complicated expressions to compute and some would be easier. You may find that the nested representation fits *your* intuition better, or you may not. ***The point is that you have the choice.***

* * *

When To Use Nested Arrays (and When NOT to Use Them)

Eschew Obfuscation!

The use of simple arrays (as has always been done with traditional APL) is a powerful tool for the solution of problems. There will, no doubt, be a temptation to use nested arrays where they are not appropriate.

Problems that can be conveniently phrased in terms of simple (nonnested) arrangements of data should be so phrased.

Nested arrays, however, do provide an option for the representation of data and increase the set of applications that become simple APL2 programs. Consider the simple task of printing a message followed by the value of a variable. This might appear in a function as:

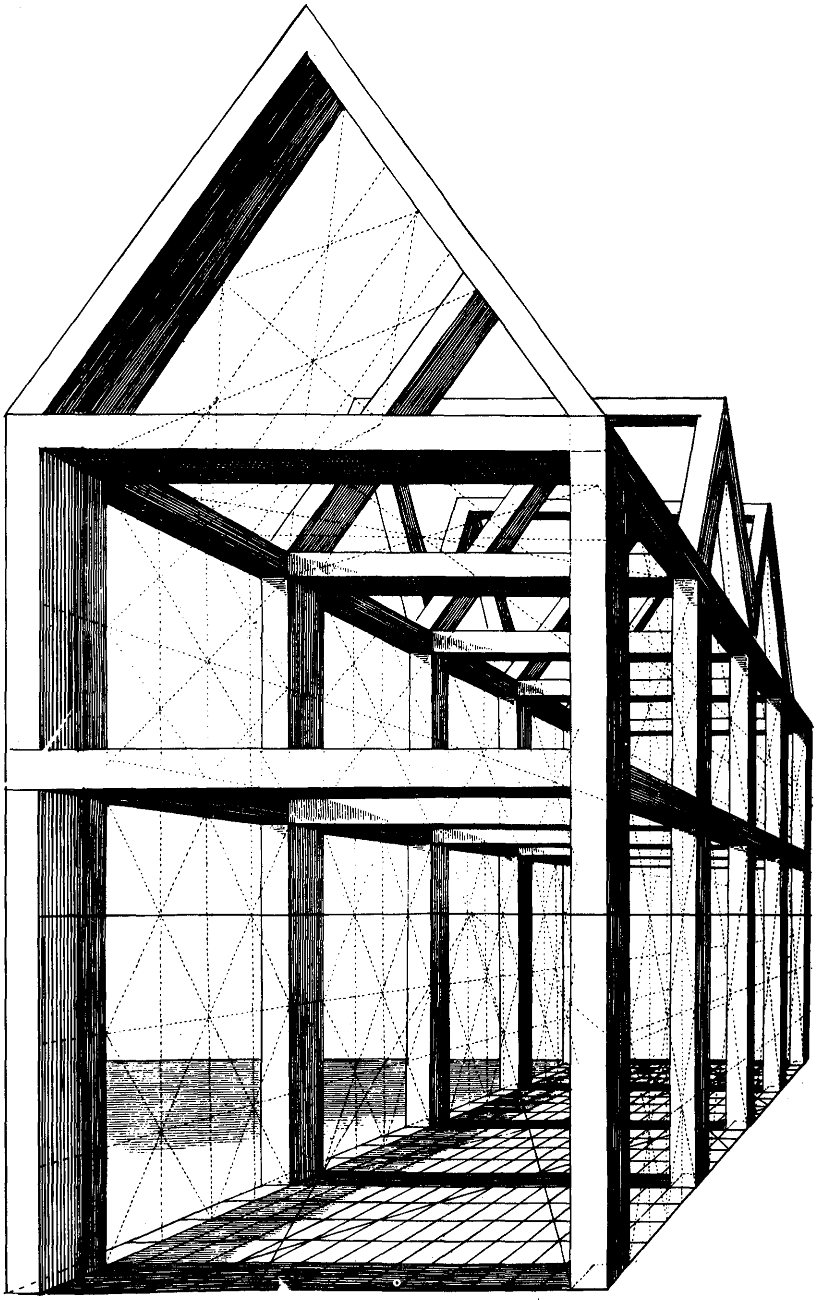
```
'THE ANSWER IS' A
```

This is a two-element vector, where the first element is a character vector, and the second element is the value of the array *A*. If *A* has the value `2 3 ρ ⍳6` this prints as:

```
THE ANSWER IS   1 2 3
                4 5 6
```

and this was accomplished *without* the application of any functions at all! Many APL programmers would have to think a while to figure out how to produce this result without nested arrays.

While this is a trivial example, it does show how something which is conceptually simple is also actually simple.[10]



Used with the permission of Dover Publications, Inc.

[10] It's also actually *nested*, of course....

Default Display of Output

By "default display," we're referring to the form in which APL2 displays arrays when no formatting functions are being used. This default display of output may be suitable for much of your formatting requirements, without even bothering to format data beyond that.

```
      (.1×⍳3)∘.×1 10 100 100000000000 1000
0.1 1 10 1E10 100
0.2 2 20 2E10 200
0.3 3 30 3E10 300
```

Notice that each column of the output was formatted *individually*, so that the inclusion of large values doesn't cause the entire display to be shown in "*E*-format."

```
      ⎕PW←53
      1 2 3∘.○⍳6
0.8414709848  0.9092974268  0.1411200081
0.5403023059 ¯0.4161468365 ¯0.9899924966
1.557407725  ¯2.185039863  ¯0.1425465431

 ¯0.7568024953 ¯0.9589242747 ¯0.2794154982
 ¯0.6536436209  0.2836621855  0.9601702867
  1.157821282  ¯3.380515006  ¯0.2910061914
```

Since APL wasn't able to fit all of the output within the confines of our page, the output was "scissored" at the printing width specified (53 characters), and the remaining right-hand portion of the data was placed below the left-hand portion, indented to indicate a continuation.

Notice that APL2 slices off *all* of the rows, and continues all of them below, rather than wrapping each line individually, as previous versions of APL did.

If you send this to a printer and want to create a display that's wider than the carriage on the printer, no problem — a small amount of cut-and-paste will produce the desired report.

Here's another example of the use of the default form of display, this time using a defined operator (which will be defined at length in an upcoming discussion):

```
      ∇ Z←L (F SEE) R
[1]    ⍝DISPLAYS ARGUMENTS TO AND OPERATION OF
[2]    ⍝ ANY FUNCTION SPECIFIED
[3]     →(0=⎕NC 'L')/MONADIC
[4]    DYADIC:Z←L F R    ⍝FUNCTION CALL IS DYADIC
[5]     ⎕←Z '←→' L 'f' R
[6]     ⎕←''
[7]     →0
[8]    MONADIC:Z←F R     ⍝FUNCTION CALL IS MONADIC
[9]     ⎕←Z '←→' 'f' R
[10]    ⎕←''
      ∇
```

```
     A←(3 4ρι12)+SEE 3 4ρ12?12
12  5  9  9   ←→   1  2  3  4   f   11 3 6  5
12 14  9 18        5  6  7  8        7 8 2 10
10 19 15 24        9 10 11 12        1 9 4 12


     B←1 2 3 ∘.×SEE 4 5 6
 4  5  6   ←→   1 2 3   f   4 5 6
 8 10 12
12 15 18


     C←2+SEE 2 2+SEE 2
 4 4  ←→  2 2  f 2

 6 6  ←→ 2 f  4 4
```

Report Formatting by Default Display

Display of arrays that contain both character and numeric data provides a convenient means for automatically formatting reports. This default output may be arranged such that visually pleasing reports may be produced with little or no additional programming.

Although the following rules for display may seem somewhat arbitrary, their main justification is to provide a visually pleasing display of tabular data.

The following features are available through default display:

- Automatic alignment of titles above numeric columns
- Automatic alignment of character data within otherwise numeric columns
- Easy control of display by selectively enclosing or formatting items (using the primitive format function)

The formatting rules for the columns of a matrix *appear* complex, but bear with us — they *do* make sense. These rules are:

1. Simple numeric scalars are formatted the same as if they were in a simple numeric matrix (that is, their decimal points are aligned, and so forth — a number is a number; it doesn't matter what else is in the column).
2. Simple character scalar or vector items are left-justified if there are *no* numeric simple scalars in that column.
3. Simple scalar or vector character items in a column with a (simple scalar) number are right justified.
4. Nonscalar items which are *not* simple character scalars or vectors are left-justified.

Higher-dimensional arrays follow these rules for each plane, and independently format their planes (except that the width of a column is the same throughout the array).

Throughout those rules, we spoke of *simple* scalars and vectors. Well, *non*simple items are formatted recursively, and are padded on their left and right by a blank (to show that they're nonsimple).

There are two key features that these rules provide: First, typical tables are automatically formatted in a visually pleasing fashion with *no additional programming* using default array display. And second, changes in the display can be made by selectively enclosing or formatting, using the primitive format function (which will be discussed in detail on pages 204-217). Here is a *very typical* example of report formatting:

```
      ∇ Z←NAMES REPORT DATA;MONTHS
[1]     MONTHS←'' 'JANUARY' 'FEBRUARY' 'MARCH'
[2]     Z←MONTHS,[1] NAMES,DATA
      ∇
```

Our data may represent a sales report; we'll enter it like this:

```
      SALES← 3 3 ρ 801 97 202 3 98 999 11 1089 'NONE'
```

This represents our three salesmen with three values for each person, showing January through March data

```
      SALES
 801   97  202
   3   98  999
  11 1089 NONE
```

Character data is okay to include, though it sure would make it hard to add them up; "0" would be a lot better if you're going to be doing calculations with it.

```
      ρSALES
3 3

      'BROWN' 'MCGREW' 'VAN DER MEULEN' REPORT SALES

                JANUARY FEBRUARY MARCH
BROWN               801       97   202
MCGREW                3       98   999
VAN DER MEULEN       11     1089  NONE
```

{Pretty erratic sales, eh?}

Notice that the word *NONE* is right-justified (rule 3).

The numeric data, of course, is right-justified (rule 1).

Notice that this field is *left*-justified, because it's *all* character data (rule 2).

The *REPORT* function contains only two trivial lines; yet, it can format a reasonable-looking report for us. If the numbers that we supply for the report are too large to fit under these headings, no problem — the columns will simply space themselves further apart — column headings and all. Likewise, if the names that we supply are longer or shorter than these, the spacing will automatically be adjusted. The *REPORT* function does this by taking advantage of the very good default formatting that's built right in to APL2. In the past, formatting a report like this could have required an entire workspace of specialized formatting functions.

Notice that the result that's produced by the *REPORT* function is a nested array:

```
      ρ'BROWN' 'MCGREW' 'VAN DER MEULEN' REPORT SALE
4 4
```

Perhaps this report needs to be used by some other process, such as being merged in with some text. A nested array isn't necessarily what we want. We're using nested arrays here simply because they automatically handle what would otherwise be rather complex data formatting. But let's say that we just want the output to be a simple character array. No problem. ...Let's just make one small change:

```
     ∇ Z←NAMES REPORT DATA;MONTHS
[1]    MONTHS←'' 'JANUARY' 'FEBRUARY' 'MARCH'
[2]    Z←⍕MONTHS,[1] NAMES,DATA
     ∇
```

The addition of this format function will change the result to a simple (nonnested) character matrix.

```
       ρ'BROWN' 'MCGREW' 'VAN DER MEULEN' REPORT SALES
4  39
```

See? ...Simple!

Even if you don't plan to construct complex data structures for storing your data in nested arrays, you may find yourself using nested arrays like the ones that we have here, simply for the advantages that they offer in the output display of the data.

* * *

By the way, if you're working with three-dimensional arrays (or higher), and have a need to create a simple character matrix that will look just like the original array, here's a function that can help:

```
     ∇ M←SIMPLE_MATRIX A
[1]    M←⍕1 1ρ⊂A          ⍝TURNS ANY ARRAY INTO A
[2]    →(0= ≡A)/0         ⍝SIMPLE CHARACTER MATRIX,
[3]    M←0 1↓0 ¯1↓M       ⍝WHICH DISPLAYS IDENTICALLY
     ∇
```

This may be needed for such tasks as sending data to a system printer.

Fill Elements, Empty Arrays, and Prototypes

Let's talk about some edge cases of APL functions. What happens when you take more than you have? What happens when you don't have anything to begin with? Such cases do show up sometimes in practical applications, but if you're just getting started in APL, you may want to skip this discussion.

What Is an Empty Array?

If you've read this far, you presumably know what an array is. [If you haven't read this far....] It's an ordered collection of items arranged along axes, where these items are numbers, characters, and other arrays. The shape function (monadic ⍴) measures the length of each dimension or axis, and the number of items in the array is the product of these lengths. Whenever the length of some axis is zero, the number of items in the array is zero and the array is called an *empty array.*

`0⍴0`	is an empty numeric vector
`⍳0`	is the same numeric empty
`''`	is an empty character vector
`5 0⍴''`	is an empty character matrix with five rows and no columns

Note that the empty arrays display much the same as nonempty arrays. The 5-by-0 array alone will display on five lines, each of which is empty. You may think that there's nothing left to say, but empty arrays have some interesting properties which are not immediately apparent. In the following pages, we will investigate these properties in an informal way by starting with nonempty arrays (vectors, in fact), where we already understand the properties, then reducing the number of elements gradually to zero. This will tell us what has to be true about empty arrays. ...All set?

Taking More Than You Have

We will now examine the "take" function (dyadic ↑) when used with a nonnegative left argument and when applied to various vector right arguments. For example,

```
      3 ↑ 1 2 3 4 5
```

gives:

```
1 2 3
```

...and in general, for nonnegative *N* and vector *V*,

```
      N↑V
```

gives a vector whose shape is " , *N*".

Suppose that we take more than we have, as in:

```
      N↑1 2 3
      N↑'ABC'                          ...for N>3
```

This is often called an "overtake." We know that the result will be a vector with *N* items. But what will the added elements be? Old APL has already answered these questions, and for *N*=5 the answers are:

```
N↑1 2 3 ←→ 1 2 3 0 0
N↑'ABC' ←→ 'ABC  '
```

For these examples, we'll write the result in the same way that you'd enter it, so that there's no question about the shape or values.

Thus, an all-numeric array is filled with zeros, and an all-character array is filled with blanks.

"What does this have to do with empty arrays?" you may ask. The answer is "a lot." Consider:

```
N↑⍳0
N↑''                                ...for N>0
```

What do you get when you "overtake" from an empty vector? Again, old APL answers this question, and for N=5 the answers are:

```
N↑⍳0 ←→ 0 0 0 0 0
N↑'' ←→ '     '
```

This tells us immediately that there is more to empty arrays than shape! They also contain information about what item to use as fill.

Let's look at a "well-behaved" nested example. Consider:

```
N↑ (1 2) (3 4) (5 6)          ...for N>3
```

The argument in this example is called a *uniform* array, because each item has the same structure — that is, each item is a two-element numeric vector. We certainly want the result to be an N-element vector, so the question is, "what do we fill with?"

Old APL gives us no help here.

Because the argument is uniform, it makes sense to fill with an item that looks like the others. Therefore, we choose as fill in this example a two-element vector of zeros. Thus, the answer (for N=5) is:

```
(1 2) (3 4) (5 6) (0 0) (0 0)
```

"Wait a minute!," you cry out. "You chose an easy example. What if the items in the vector are all different — *then* what do you do?"

Well... you're right. In the cases discussed so far, there is little doubt about what to do. But if the data is *not* well organized, there is *no* choice which will *always* match the intent of the user. We *can*, however, follow some general guidelines. They are:

1. Do what old APL does, where defined.
2. Simple arrays should remain simple.
3. Uniform arrays should remain uniform.
4. The result must be predictable.

A definition that follows these guidelines says to fill with the "*type*" of the first item, where "first" (monadic ↑) is the function which selects the first item of any array (in row-major order), and "type" is the scalar function which turns numbers into zeros and characters into blanks.

If we apply this to some of the examples that we discussed before, we get:

```
    0    ↔  TYPE ↑ 1 2 3
  ' '    ↔  TYPE ↑ 'ABC'
  0 0    ↔  TYPE ↑ (1 2) (3 4) (5 6)
```

What, you may ask, is this function *TYPE*? It's a scalar function that produces zeros from numbers and blanks from characters. Although it's a defined function, it's a very simple one, and we'll be using it almost as if it were a primitive function. Our definition looks like this:

```
      ∇ Z←TYPE R
[1]     Z←↑0⍴⊂R
      ∇
```

...So just think of this defined function as being "primitive."

For every nonempty array, *TYPE*↑ (type of the first) is well-defined, and the resulting array is called the *prototype* of the array ["prototype," after all, means "first type"].

The concept of a prototype is applied to empty arrays by defining the prototype of an empty array based upon the operation and data used to create the empty array — don't panic, you'll see some concrete examples of this in just a bit. When the "first" function (↑) is applied to an empty array, it returns the prototype of the array.

In summary:

- Every array has a prototype defined as the type of the first item (*TYPE*↑).
- First (↑) applied to an empty array returns the prototype.
- The prototype is used as the fill item for overtakes.

If you think that this is circular, think again (and again (and ...)). It *is* circular. This is because we are talking about the fundamental properties of data. For a more complete and *formal* treatment of this subject, see *APL2 Programming: Language Reference.*

Where does all of this lead? ...Several places. First, we've already answered the question of what to fill with in an overtake. The same also holds true for the "expand" function, and everything works on "empties" just as well. Second, because empty arrays have prototypes, *empty arrays can be nested!*

Consider again,

`N↑ (1 2) (3 4) (5 6)` ...but this time, for *N*<3.

- If *N*=2, the result is a two-element vector of two-element vectors.
- If *N*=1, the result is a one-element vector of two-element vectors.
- If *N*=0, the result is *a zero-element vector of two-element vectors.*

Thus, even empty arrays may have an arbitrary structure, and, as we said before, this structure (the prototype) is based upon the operations and the data used to create the empty array.

Thus, we may add the following to our previous examples of empty arrays:

`0↑⊂0 0`	is an empty vector of two-element vectors
`5 0⍴⊂0 0 0`	is an empty matrix of three-element vectors
`0⍴⊂⍳0`	is an empty vector of empty vectors

Let Me Count the Ways

We discovered the properties of empty arrays by looking at the "take" function (↑). Now we'll take a look at some functions applied to empty arrays and see what can be learned.

Consider the expression:

```
(N↑ (1 2) (3 4)) ,¨ (N↑ (1 2 3) (4 5 6))
```

The left argument of the catenation is a vector of two-element vectors, like the one we used with "take" in the previous discussion. Again, *N* says how long the vector is. We know from before that the result of the *N*↑ will be a uniform vector of two-element vectors, for any legal *N*. The right argument of the catenation is similar, except that each item is a three-element vector. The function being applied is "catenate each" (,¨). (The "each" operator was discussed back on page 41; you may want to refer back.) As you may recall, this will cause "catenate" to be applied between corresponding items of the two arguments, giving five-element vectors as items of the result. For example, if *N*=2 this becomes:

```
(2↑ (1 2) (3 4)) ,¨ (2↑ (1 2 3) (4 5 6))
   (1 2) (3 4)    ,¨     (1 2 3) (4 5 6)
  (1 2 , 1 2 3)           (3 4 , 4 5 6)
       (1 2 1 2 3) (3 4 4 5 6)
```

For N > 2 the arguments will be padded but the result will still be a vector of five-element vectors. For N = 1 we get a one-element vector of five-element vectors, and it's probably no surprise now that for N = 0 the result is *an empty vector of five-element vectors*. Nothing else would make sense. Try it:

```
      (0↑ (1 2) (3 4)) ,¨ (0↑ (1 2 3) (4 5 6))
```

↗

Well, it's *empty*, so it's tough to tell that it's an empty vector containing five-element vectors. But look at the prototype and recall that "first" on an empty returns the prototype.

```
      ↑(0↑ (1 2) (3 4)) ,¨ (0↑ (1 2 3) (4 5 6))
0 0 0 0 0 ←——— ...now, that's more like it.
```

Let's look at this from another angle. Suppose that N = 2. How many times is the catenate function applied? Well, since it's applied between corresponding items and there are two items in the arguments, catenate must get applied two times. Refer back to that last example and verify this. If N = 10000, then catenate would be applied 10,000 times. What if N = 1 ? ...No problem: one application. How about N = 0 ? ...Oh oh! The answer had *better* be zero times. Yet we know that the result has a prototype of five-element vectors. How can this be?

The answer is that when we reach the empty case in this situation, the function is *not* applied — instead, a related function, called the *fill function*, is applied. This function sees the prototypes as arguments, and its result will determine the prototype of the result of the derived function. In our example, the fill function would see "0 0" as the left argument and "0 0 0" as the right argument, so its result would be "0 0 0 0 0" — which is, as we have already seen, the prototype of the result of the "catenate each" (,¨).

In this case, the fill function for catenate was still a catenate, it just applied to prototypes. This is generally true of the primitive functions.

APL2 does not provide a way to define the fill function associated with a defined function.

A New Way to Reduce

We're not talking about a new diet plan, we're going to take another look at reduction. We saw before that:

```
      ×/1 2 3    ←→    1×2×3    ←→    6
```

Now consider:

```
      ×/ N↑ 1 2 3
```

For N > 3 we will get 0 for every result, because we'll be multiplying by zero. For N = 2 we get 2, for N = 1 we get 1, and for N = 0 we *also* get 1. This is not new in APL2. Reduction of empty vectors has always given the identity-element of the function (0 for +, 1 for ×, and so forth).

If we have uniform nested arrays, everything continues to work fine:

```
×/ N↑ (1 2) (3 4) (5 6)
```

For *N*=3 we get "⊂15 48". The result is an enclosed two-element vector, and so is a scalar *containing* a two-element vector. [Whenever you reduce a vector with a pervasive function (also known as a "scalar" function), you get a scalar.] For *N*=2 we get "⊂3 8". For *N*=1 we get "⊂1 2". And for *N*=0 we get "⊂1 1".

So, as before, even when the argument becomes empty, the result has the same structure — in this case, the result is always a scalar containing a two-element vector.

Let's look at this from another angle. If *N*=3, how many times does the "×" function get applied? Looking at the example, it obviously gets applied two times. (Of course, it might be applied between array arguments.)

If *N*=2 the function is applied one time. If *N*=1 the function is not applied at all, and the result is that one item, now packaged in a scalar rather than in a one-element vector.

If *N*=0 the function is not applied at all — yet we get a nonempty result, which is clearly related to the function (because we get ones — the identity of times), and to the empty data (because we get *two* ones).

What happens is that in the reduction of an empty, a related function called the *identity function* is executed.

The identity functions for all of the primitive functions that have them defined can be found in *APL2 Programming: Language Reference.*

* * *

This completes the discussion of fill elements, empty arrays, and prototypes. Empty cases *do* arise in real applications, and when they do, you'll find that *the* APL *primitives behave in expected ways.*

Using Defined Operators

What's an operator, and why would I want to define one? That's a good question. Before we try to answer it, however, let's review what a function is. Understanding operators requires an understanding of functions. Understanding functions, in turn, requires an understanding of arrays. If you need help with *that,* go back to "Understanding Arrays" on pages 33-45.

One Man's Ceiling Is Another Man's Floor

This discussion will be limited to monadic and dyadic functions that return explicit results. They are, after all, the most useful ones, because their results can be directly used as the argument to the next function on the same line, at which point one function's result is another function's argument.

A monadic function that returns an explicit result typically modifies its array argument. The result might be completely different from the argument, or it might be very similar.

```
      ∇ Z←NEGATIVE N
[1]     Z←-N
      ∇

        NEGATIVE 20
¯20
```

Similarly, a dyadic function that returns an explicit result typically combines its two array arguments in some way to make a new array. The result might be completely different from, or very similar to, either or both of its arguments.

```
      ∇ Z←A PLUS B
[1]     Z←A+B
      ∇

        10 PLUS 20
30
```

What's an Operator?

An operator is to a function what a function is to an array. Operators can be used to study and manipulate functions, just as functions can be used to study and manipulate arrays. Functions are sometimes likened to verbs, as arrays are sometimes likened to nouns. If we were to continue that analogy, operators would become the adverbs.

In APL2, you can create your own operators as well as functions. An operator can be created with one of the system editors, just like defining a function.

One difference between the definition of this operator and a regular APL2 function is that one or two of the parameters in the operator's header may be functions as well as variables (which, after all, are only named arrays). The *derived function* defined by the operator can thus control the execution of its function operands on its array arguments.

Another difference between a defined operator and a defined function is the way they are invoked by an APL2 expression. Operators are invoked only when they are found in the syntactical context of an operator. We'll look at some examples of this context in just a moment.

Suppose that we have an operator called *REDUCTION*, which, for purposes of illustration, will do the same thing as the primitive "reduce" operator:

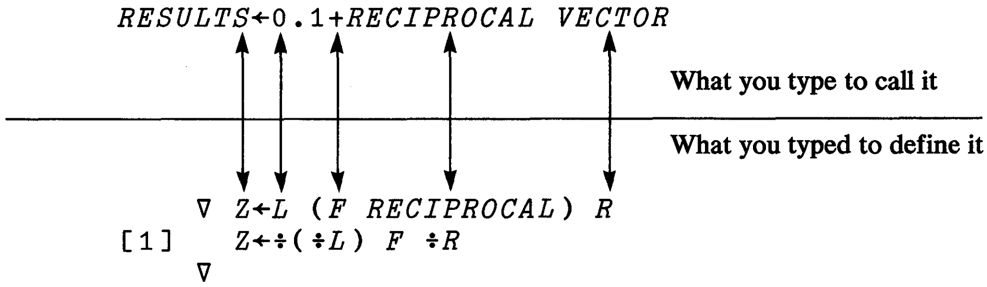
```
    ∇ RESULT←(FUNCTION REDUCTION) ARGUMENT
[1]   RESULT←FUNCTION/ARGUMENT
    ∇

      +REDUCTION 10 20 30
60
```

And, sure, you can use a defined function with a defined operator... here's the *PLUS* function from the previous page:

```
      PLUS REDUCTION 10 20 30
60
```

This *REDUCTION* operator is monadic, because its only operand (between the parentheses) is *FUNCTION*. You can think of *REDUCTION* as the name of the operator, and (*FUNCTION REDUCTION*) as the name of the derived function that it represents. There is an explicit *RESULT*, and the derived function is monadic because there is only one *ARGUMENT*.

An operator always has one set of parentheses in its header to indicate the operator context in which it is to be recognized. A monadic operator takes a single function or array operand on the *left* of the operator name. A dyadic operator, of course, takes its function or array operands on both sides of the operator name. Outside the parentheses, the header of a defined operator is exactly the same as for a monadic or dyadic function. It must have a right argument, and it *may* have a left argument, an explicit result, and local variables.



```
RESULTS←0.1+RECIPROCAL VECTOR

    ∇ Z←L (F RECIPROCAL) R
[1]   Z←÷(÷L) F ÷R
    ∇
```

An operator takes as its left operand the function or array to its left, which may itself be a derived function produced by another operator.

Let's poke around a bit inside this next operator, and see what that function looks like:

```
      ∇ Z←(F SHOW) R
[1]     ' ⎕NC F:' (⎕NC 'F')   )
[2]     ' ⎕CR F:' (⎕CR 'F')   }  Let's just display
[3]     '⍴⎕CR F:' (⍴⎕CR 'F')  )  some information[11]
[4]     ''
[5]     Z←F R ←———— ...before we apply the function.
      ∇

        ×REDUCTION SHOW 10 20 30
  ⎕NC F: 3 ←——————— "F" is indeed a function
  ⎕CR F: ←————————— but we can't display it...
 ⍴⎕CR F:   0 0 ←——— ⎕CR returns an empty display.

6000
```

In this case, *F* is a *derived function* ("×*REDUCTION*"). We can take the canonical representation of only a defined function, not a primitive function or a derived function. So, let's try it with a defined function — we can use that *NEGATIVE* function that we showed on page 57:

```
        NEGATIVE SHOW 10 20 30
  ⎕NC F: 3
  ⎕CR F:   Z←NEGATIVE N
           Z←-N
 ⍴⎕CR F:   2 12

¯10 ¯20 ¯30
```

...In fact, we can go a step further....

```
        S∆SHOW←1 ←——————————— Set "Stop Control"[12]

        NEGATIVE SHOW 10 20 30
SHOW[1]                       If the function is a
        ∇F[⎕]∇ ←————————————  defined function, it
      ∇ Z←NEGATIVE N          can be displayed by
[1]     Z←-N                  using ⎕CR, ⎕TF, or
      ∇                       one of the system's
                              del-editors.
```

[11] "⎕*CR*" and "⎕*NC*" are "system functions"; we'll be discussing them later (on pages 125 and 130, respectively).

[12] "*S*∆-name" is "stop control," causing the specified defined function or operator to halt execution at the indicated line number. Stop and Trace are both described in *APL2 Programming: Language Reference*.

Just as a defined function can be written to examine an array, a defined operator can be written to examine the behavior of a function. For example,

```
      ∇ Z←L (F TRACE) R
[1]     'RIGHT ARGUMENT:  ' R
[2]     ' LEFT ARGUMENT:  ' L
[3]     Z←L F R
[4]     '         RESULT:  ' Z
[5]     ''
      ∇

        (2×3) +TRACE 4×5
 RIGHT ARGUMENT:  20
  LEFT ARGUMENT:  6
         RESULT:  26

26
```

We could even use it to tell us something about how another operator works!

```
        +TRACE/ 1 2 3 4
 RIGHT ARGUMENT:  4
  LEFT ARGUMENT:  3
         RESULT:  7

 RIGHT ARGUMENT:  7
  LEFT ARGUMENT:  2
         RESULT:  9

 RIGHT ARGUMENT:  9
  LEFT ARGUMENT:  1
         RESULT:  10

10
```

Using an Operator to Modify a Function

More often, a defined operator can be used to *modify* the behavior of a function in some systematic way. For example, we could use a dyadic operator to control the index origin during the execution of an arbitrary function — here is such an operator, called "*IN_ORIGIN*":

```
      ∇ Z←L (FUNCTION IN_ORIGIN IO) R;⎕IO
[1]     ⎕IO←IO
[2]     Z←L FUNCTION R
      ∇

        10 20 30 ⍳ 40 20 10
4 2 1

        10 20 30 (⍳IN_ORIGIN 0) 40 20 10
3 1 0
```

Notice that in this last example, we used parentheses around the derived function. Without parentheses, there may be some visual ambiguity to a newcomer in APL2 circles as to whether the "0" would have been considered part of a four-item vector "0 40 20 10" (*all four* elements of which would have been taken as the right *operand* of "*IN_ORIGIN*", with nothing left for the right *argument*), or whether the "0" is the right operand of "*IN_ORIGIN*" and the "40 20 10" vector is the right argument. In fact, the latter case is the way APL2 evaluates it, but the point is, if you feel that clarity is enhanced by the use of the parentheses, by all means, use them. A simple rule of thumb is: If you write the calling expression so that it looks like the header of the operator, with the parentheses in the same places, it will always work. They are often not needed in the calling expression — like each of the preceding examples — but they are always allowed.

Dyadic operators taking functions as both operands often combine them in some way to produce a new derived function. The primitive inner product operator is a good example. By defining a dyadic operator, we have the opportunity to create similar types of things in endless variety.

For example, we can write a defined operator which, instead of combining two functions, just joins their results and returns the two results as a two-element vector, so that you can visually compare them.

```
          ∇ Z←(F AND G) R
     [1]    Z←(F R)(G R)
          ∇
```

Notice that "*AND*" is a dyadic operator, because it has both left and right operands, "*F*" and "*G*", but its derived function is monadic (because it takes only a right argument, "*R*").

```
            ⍋AND⍒ 1 2 4 3 5
       1 2 4 3 5  5 3 4 2 1
```

We can even use it to compare two derived functions!

```
            ≠\ AND (<\) 0 0 1 1 1 0
       0 0 1 0 1 1  0 0 1 0 0 0
```

If we write another defined operator which combines a dyadic function and its left argument to make a monadic function, then we could compare the result of a monadic function with the result of a dyadic function.

```
          ∇ Z←(F GLUE L) R
     [1]    Z←L F R
          ∇
```

Looks too simple to be useful? Let's use it —

```
            *GLUE 2 AND* 0 1 2
       1 2 4  1 2.71828182 7.389056099
```

(This example compares powers of two with powers of "e".) Let's try that with parentheses around the operands, just to show how the evaluation proceeds:

```
            ((*GLUE 2) AND*) 0 1 2
       1 2 4  1 2.71828182 7.389056099
```

This is the same as the former example. The parentheses are not needed here, but they can always be used.

Another application of defined operators is to use them in combination with event controls (see "Event Handling" on pages 158-189). For example, we can write a defined operator that attempts to execute a function normally, but returns a "*?*" if anything goes wrong:

```
      ∇ Z←L (F TRAP) R
[1]     Z←'''?''' ⎕EA 'L F R'
      ∇


        1 2 3 ÷TRAP 4 5 6
0.25 0.4 0.5
```

Nothing went wrong, so we get the right answer — this time. But maybe we were just lucky that time. Let's try it again:

```
        1 2 3 ÷TRAP 4 0 6
?
```

This time, the "0" in the right argument of the derived function would normally have caused a *DOMAIN ERROR*, but the *TRAP* operator circumvented it. ...And if we use the "each" operator, we can discover *where* the problem occurred:

```
        1 2 3 ÷TRAP¨ 4 0 6
0.25 ? 0.5
```

* * *

These have been a few scattered examples of how defined operators can be used. Try them out — their uses are as unlimited as the uses of defined functions.

Chapter 3: The External Environment

This chapter is going to tell you about the details of the environment in which APL operates. No, we aren't going to talk about the details of the operating system or the computer that APL is running on. We're going to talk about APL workspaces, what the rules are for defining your own functions, and how to use the built-in function editors to define your own functions.

System Commands

An APL system recognizes two broad classes of instructions, *expressions* and *system commands*. System commands control the initiation and termination of a work session, saving and reactivating copies of a workspace, transferring data from one workspace to another, and a variety of tasks within the workspace.

System commands can be thought of as being a sort of link to the outside world, allowing the users of APL functions and variables access to the environment which may not be defined as properly being a part of the APL Language.

System commands can be invoked *only by individual manual entries* from the keyboard and *cannot be executed dynamically* as part of a defined function. They are distinguished from APL statements in that they are always prefixed by a right (closing) parenthesis.

A command that is not recognizable, or is improperly formed, is rejected with the report *INCORRECT COMMAND*. Certain commands may also result in more specific trouble reports; these are discussed in the appropriate context. Both the system commands and their trouble reports are available in several national languages. Refer to the discussion of ⎕*NLT* (National Language Translation) on pages 143-145. English will be used for all of our examples.

Since these commands are separate from the rest of the APL Language, and are never under the control of APL functions, they are *not* subject to Event Handling; none of the error messages are trappable, and none of the system commands will set ⎕*EM* or ⎕*ET* (to be discussed later).

The Categories of System Commands

System commands fall into four broad categories:

1. The active workspace

   a. Action
   b. Inquiry

2. Workspace storage and retrieval

   a. Action
   b. Inquiry

3. Access to the system

4. Communication with other users

Just for reference, here's a list of the commands that fall into each of these categories (don't worry about how to use all of these commands yet; we'll be getting into the details a bit further on):

The Active Workspace

These commands assist in programming or application usage within the active workspace:

Command	***Page***	***Description***
)*EDITOR*	(85)	Specifies which system editor you wish to use when you edit objects in the workspace
)*ERASE*	(82)	Discards selected objects
)*FNS*	(74)	Lists the names of the functions in the workspace
)*MORE*	(85)	Provides additional error information if available
)*NMS*	(74)	Lists the names of all of the objects in the workspace and tells what type of object each one is
)*PBS*	(87)	Specifies and reports the setting of the "printable backspace" character, for entering certain characters not otherwise available on the terminal.
)*RESET*	(84)	Empties the state indicator of all entries or a selected number of entries
)*QUOTA*	(86)	Displays various limits imposed upon your use of APL
)*SI*	(83)	Displays the state indicator —)*SINL* and)*SIS* are variations
)*SYMBOLS*	(86)	Reports and sets the size of the symbol table
)*TIME*	(87)	Displays the date and time.
)*VARS*	(75)	Lists the names of the variables in the workspace
)*WSID*	(75)	Reports the name of the workspace, and sets both the name and password of the workspace

Workspace Storage and Retrieval

These commands put workspaces onto permanent storage devices and get them back again for you, they help you to move workspaces between systems, and they help you keep track of what you have stored:

Command	***Page***	***Description***
)*CLEAR*	(72)	Discards the active workspace, giving you a fresh start
)*CONTINUE*	(82)	Saves the active workspace under the name "*CONTINUE*," and signs you off of APL
)*COPY*	(77)	Combines selected objects from another workspace with the active workspace
)*DROP*	(76)	Discards a stored workspace
)*IN*	(80)	Moves transfer forms from a transfer file into the active workspace
)*LIB*	(75)	Lists the names of stored workspaces
)*LOAD*	(77)	Brings a stored workspace into the active workspace
)*MCOPY*	(80)	Migrates workspaces from VSAPL

)OUT	(81)	Converts objects to transfer form, and stores them on a transfer file
)PCOPY	(79)	Like)COPY, but ensures that you are protected against damage from name conflicts
)SAVE	(76)	Stores a copy of the active workspace on a permanent storage device for future use

Access to the System

These commands let you sign on to and off from APL:

Command	***Page***	***Description***
)123456:ABCDEF	(68)	On some systems, a command of this format signs you on to APL
)CONTINUE	(82)	Saves the active workspace under the name "CONTINUE," and signs you off of APL
)HOST	(85)	Executes host system commands
)OFF	(82)	Discards the active workspace and signs you off of APL

Communication with Other Users

These commands let you send messages to the terminal of another user who is currently signed on to the same APL system:

Command	***Page***	***Description***
)MSG	(84)	Sends a message to another APL user —)MSGN is a variation
)OPR	(85)	Sends a message to the APL System Operator —)OPRN is a variation

Access to the System

Each user of the system is assigned by the system manager an *account identification* used to identify data storage and charges for use of the system. The account identification is required in order to sign on.

Each user is also assigned a *quota*, indicating the maximum number or combined capacity of saved workspaces, and a *shared variable quota*, indicating the maximum number of variables that may be shared simultaneously.

Sign On

Before work can be started, a physical connection to the computer must first be established. This may require as little as turning a switch, or may require establishing a link from a dial-up terminal to a central computer, possibly passing through intermediary computing systems which are host to APL, depending on the system employed and the type of terminal device employed.

When you begin the APL session, either a clear workspace or the *CONTINUE* workspace is normally activated, depending on the condition that terminated the preceding session, and the system in use. If *CONTINUE* was activated, the system reports the date and time at which it was saved.

For more information, refer to *APL2 Programming: System Services Reference.*

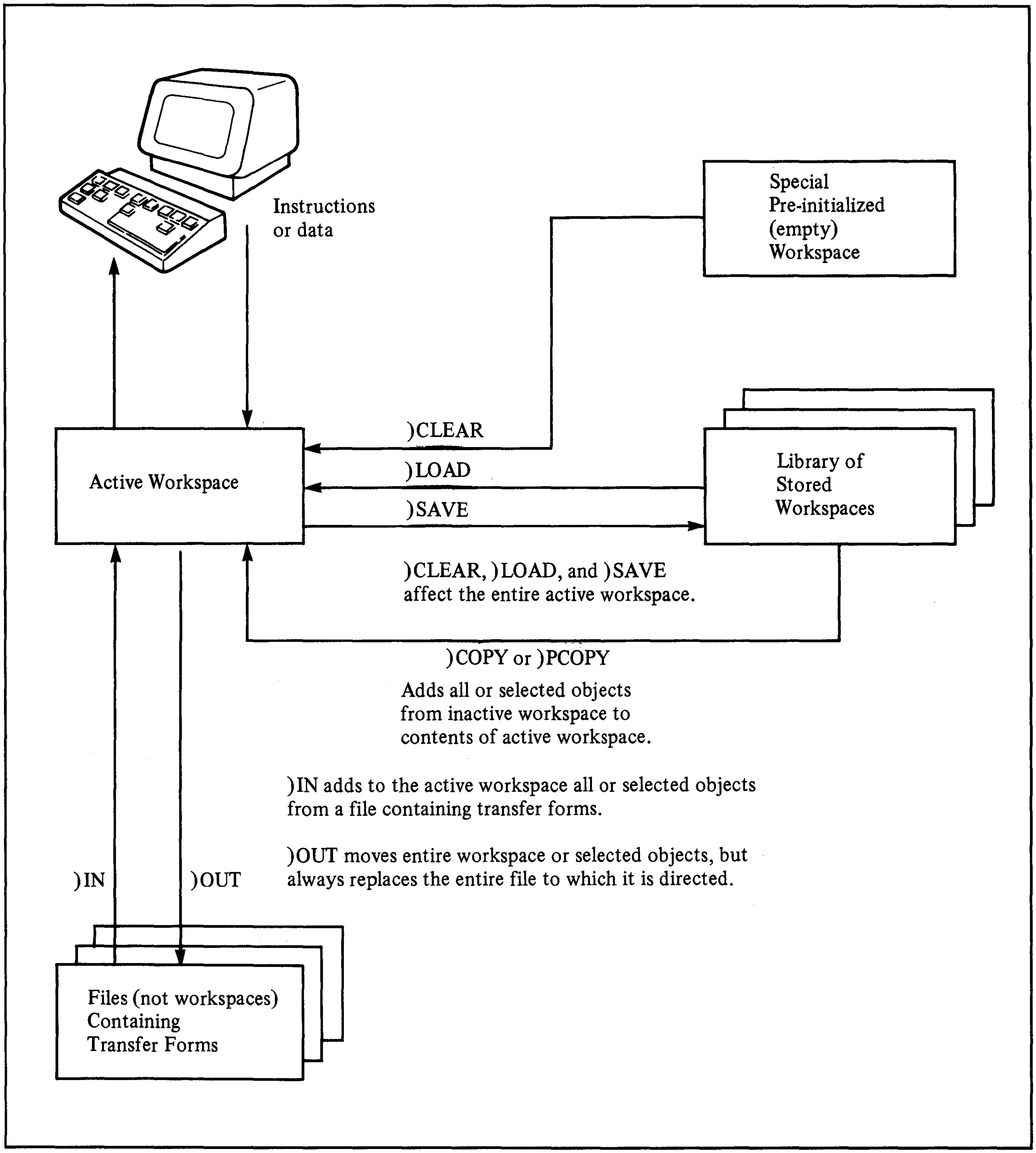
Workspaces and Libraries

The common organizational unit in an APL system is the *workspace*. When in use, a workspace is said to be *active*, and it occupies a block of working storage in the central computer. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, for storing items of information and for containing transient information generated in the course of a computation.

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

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

The Effects of Selected System Commands



Instructions or data
Special Pre-initialized (empty) Workspace
)CLEAR
)LOAD
)SAVE
Active Workspace
Library of Stored Workspaces
)CLEAR,)LOAD, and)SAVE affect the entire active workspace.
)COPY or)PCOPY
Adds all or selected objects from inactive workspace to contents of active workspace.
)IN adds to the active workspace all or selected objects from a file containing transfer forms.
)OUT moves entire workspace or selected objects, but always replaces the entire file to which it is directed.
)IN
)OUT
Files (not workspaces) Containing Transfer Forms

Commands for Workspace Storage and Retrieval

You may request that a duplicate of your currently active workspace be saved for later use. When a duplicate of a saved workspace is subsequently reactivated, the entire environment of computation is restored as it was, except that variables which were shared in the active workspace are not automatically shared again when the workspace is reactivated.

Libraries of Saved Workspaces

The set of workspaces that you have saved is called your *library.* Each workspace is identified by your account identification and the name that you assign to it. However, in referring to workspaces in your own library, the account identification may be omitted; your own identification is then supplied automatically.

In systems with multiple users, it is often convenient to use functions or variables contributed by others. You can activate an entire workspace saved by someone else, or may copy from it selected items. To do so both the library number and the name of the desired workspace must be supplied. However, APL provides no way of learning either the account identification or the names of workspaces belonging to other users. Thus, you may make use of material from the libraries of others only if they supply that information.

Certain libraries (usually identified by a particular group of library numbers) are not assigned to individual users, but are designated as *public* libraries. Any user may obtain a list of workspaces in a public library, and may use public workspaces. However, there are generally restrictions on who can save, drop, or modify a workspace in a public library.

Passwords

Stored workspaces and the information they hold can be protected against unauthorized use by associating a *lock*, comprised of a colon and a *password* of the owner's choice, with the name of the workspace, when the workspace is stored. In order to activate a locked workspace using APL or copy any information it contains, a colon and the password must again be used, as a *key*, in conjunction with the workspace name. Listings of workspace names, including those in public libraries, never display the keys, and do not overtly indicate the existence of a lock.

System Commands and Their Syntax

Command	Pg	Command Syntax
CLEAR	72	)*CLEAR* [size]
NMS	74	)*NMS* [first [last]]
FNS	74	)*FNS* [first [last]]
OPS	74	)*OPS* [first [last]]
VARS	75	)*VARS* [first [last]]
LIB	75	)*LIB* [library] [:[password]] [first [last]]
WSID	75	)*WSID* [[library] workspace [:[password]]]
SAVE	76	)*SAVE* [[library] workspace [:[password]]]
DROP	76	)*DROP* [library] workspace [:[password]]
LOAD	77	)*LOAD* [library] workspace [:[password]] [size]
COPY	77	)*COPY* [library] workspace [:[password]] [names
PCOPY	79	)*PCOPY* [library] workspace [:[password]] [names]
MCOPY	80	)*MCOPY* [library] workspace [:[password]] [names]
IN	80	)*IN* filename [list]
OUT	81	)*OUT* filename [list]
ERASE	82	)*ERASE* names
OFF	82	)*OFF* [*HOLD*]
CONTINUE	82	)*CONTINUE* [*HOLD*]
SI	83	)*SI*
SINL	83	)*SINL*
SIS	83	)*SIS*
RESET	84	)*RESET* [number]
MSG	84	)*MSG ON* or *MSG OFF*
MSG[*N*]	84	)*MSG*[*N*] user message
OPR[*N*]	85	)*OPR*[*N*] message
HOST	85	)*HOST* [command]
EDITOR	85	)*EDITOR* [number] or [name]
MORE	85	)*MORE*
QUOTA	86	)*QUOTA*
SYMBOLS	86	)*SYMBOLS* [number]
TIME	87	)*TIME*
PBS	87	)*PBS* or)*PBS ON* or)*PBS OFF*

Notes:

Items within brackets [ ] are optional.

For further discussion on all of the system commands, please refer to *APL2 Programming: Language Reference.*

All of the system commands are available in several national languages. Please refer to ⎕*NLT* (National Language Translation) on pages 143-145 for details.

Details of Usage

Here are the details of the usage of each of the system commands, which we're sure you'll enjoy perusing over supper tonight, or riffling through in your copious free time. The allowable syntax is shown with each command. Items shown within brackets, "[ ]," are optional. These brackets are shown only to illustrate the syntax... the brackets themselves are *never* meant to be entered with any of the system commands.

The system commands shown here are part of the distributed version of APL2. Some systems may have additional system commands to handle local requirements.

The system commands are available in several languages; refer to the discussion of ⎕*NLT* (National Language Translation) on pages 143-145 for information on usage, and refer to the Appendix of *APL2 Programming: Language Reference* for a complete list of both the system commands and their related trouble reports in each of the available languages. The commands will be shown here only in English.

* * *

)*CLEAR* [size]

This command is used to make a fresh start, discarding the contents of the active workspace, and resetting the environment to standard initial values (as shown in the table on the next page). At sign-on to APL, the user receives a clear workspace characterized by these same initial values (unless the workspace "*CONTINUE*" was automatically loaded).

You can include an optional parameter to indicate the *size* of the workspace that you wish to create. Where applicable, the size is to be stated in bytes (characters); the actual workspace size that you are given will match the requested size as closely as possible.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*CLEAR* command.

Environment within a Clear Workspace

These values are present at sign-on time (unless reset through the automatic loading of a *CONTINUE* workspace), and will then be carried forward following a)*LOAD* or)*CLEAR*:

⎕*HT*	Horizontal Tabs	⍳0
⎕*NLT*	National Language Translation[1]	' ' (The initial language depends upon the country where the system is located)
⎕*PW*	Printing Width[1]	(Depends upon the type of terminal being used)
⎕*TZ*	Time Zone[1]	(Depends upon the location of the system)
)*PBS*	Printable Backspace	_ (that is, *ON*)
)*EDITOR*	System Editor	1

These values are present following)*CLEAR*; the settings of the entries in the upper chart will be carried over from the previous workspace:

⎕*L*	Left Argument	No value
⎕*R*	Right Argument	No value
⎕*CT*	Comparison Tolerance[1]	1*E*¯13
⎕*EM*	Event Message	3 0ρ' '
⎕*ET*	Event Type	0 0
⎕*FC*	Format Control	.,*0_¯
⎕*IO*	Index Origin	1
⎕*LC*	Line Counter	⍳0
⎕*LX*	Latent Expression	''
⎕*PP*	Printing Precision[1]	10
⎕*PR*	Prompt Replacement	' '
⎕*RL*	Random Link[1]	7*5 ←→ 16807
⎕*SVE*	Shared Variable Event	0
⎕*WA*	Workspace Available[1]	(Depends upon the local installation, and in some systems, upon options selected by the user)
)*WSID*	Workspace name	None (*CLEAR WS*)
	Workspace password	None
)*SI*	State indicator	Cleared

[1] These items have values which may vary from system to system.

For entries where values are shown, those values were chosen as being widely-used values.

(For your convenience, this table will be repeated in the section "System Variables" on page 135)

)*NMS* [first [last]]

This command displays a list of all of the global objects in the active workspace, in alphabetical order, starting with the letter indicated by "first," and continuing through the letter indicated by "last." If "last" is elided, the list starts from the point indicated by "first" and continues to the end. If *both* "first" and "last" are elided, the list starts from the beginning and continues to the end.

Beside each name is a number indicating which class the name belongs to; these classes are the same ones that are used by ⎕*NC* (Name Classification) and ⎕*NL* (Name List). They are:

2. Variable	[A value of "1" refers to
3. Function	labels, which cannot be
4. Operator	global objects.]

For example:

```
      )NMS
ADDRESSES.2     BEGIN.3 FN1.3   FOO.3   FORMAT.2
GOO.3   OP.4    REPORT.3        SEE.4   TABLE.2
TABLES.3        VAR1.2  V2.2

      )NMS G
GOO.3   OP.4    REPORT.3        SEE.4   TABLE.2
TABLES.3        VAR1.2  V2.2

      )NMS F R
FN1.3   FOO.3   FORMAT.2        GOO.3   OP.4
REPORT.3
```

)*FNS* [first [last]]

This command is similar to the)*NMS*-command, except that it lists only the global functions in the active workspace. As with the)*NMS*-command,)*FNS* will accept beginning and ending letters:

```
      )FNS
BEGIN   FN1     FOO     GOO     REPORT  TABLES

      )FNS F R
FN1     FOO     GOO     REPORT
```

)*OPS* [first [last]]

This command displays a list of the global operators in the active workspace. It is similar to the)*NMS*-command, but shows only the operators. As with)*NMS* and)*FNS*,)*OPS* will accept beginning and ending letters:

```
      )OPS
OP      SEE

      )OPS F R
OP
```

)VARS **[first [last]]**

This command displays a list of the global variables in the active workspace. It is similar to the *)FNS*-command, but for variables. As with *)FNS*, *)VARS* will accept beginning and ending letters:

```
      )VARS
ADDRESSES       FORMAT  SALES   TABLE   VAR1
V2

      )VARS G
SALES   TABLE   VAR1    V2

      )VARS F R
FORMAT
```

)LIB **[library] [:[password]] [first [last]]**

This command displays, in alphabetical order, the names of the workspaces in your private library or in an indicated public library. The list starts with the letter indicated by "first," and continues through the letter indicated by "last." If "last" is elided, the list starts from the point indicated by "first" and continues to the end. If *both* "first" and "last" are elided, the list starts from the beginning and continues to the end.

On some systems, a password is required to display the contents of some libraries.

For example:

```
      )LIB
ANOTHER CONTINUE        DEMO    DISPLAY EXAMPLE
EXAMPLES        MIGRATE REPORT1

      )LIB G
MIGRATE REPORT1

      )LIB N R
REPORT1
```

An attempt to display the list of workspaces in another user's private library or in a non-existent public library is rejected with the report
IMPROPER LIBRARY REFERENCE.

)WSID **[[library] workspace [:[password]]]**

This command assigns to the active workspace the name indicated, and (optionally) the library number or the password indicated. Use of the colon with nothing following it assigns an empty password; that is, it removes a former password if there was one. If the active workspace is subsequently saved, future use of the saved workspace will require use of the password set here.

Setting of the active workspace's identification is acknowledged by the report "*WAS* ..." followed by the former name (but *not* the former password).

)*SAVE* **[[library] workspace [:[password]]]**

A duplicate of the active workspace is saved (optionally, in the indicated public library, otherwise in the user's own library) under the indicated name, and (optionally) with the new password indicated. If the library number, workspace name, or password are omitted, they are supplied from the workspace identification. After saving, the active workspace has the same identification (including library number, name, and password) as the saved workspace.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*SAVE* command.

Although saving does not affect the state of sharing in the active workspace, current values of the shared variables are saved in the stored copy even if you haven't actually referenced them yet.

Saving is acknowledged by a report showing the date and time at which the workspace was saved.

The command to save the active workspace may be rejected, with trouble reports as follows:

IMPROPER LIBRARY REFERENCE
The system does not permit a workspace to be saved in the private library of another user, or in a non-existent public library, nor does it permit a workspace named *CONTINUE* to be saved in a public library.

NOT SAVED, THIS WS IS ...
Saving is not permitted when the name given in the command matches the identification of an existing saved workspace but does not match the identification of the active workspace. This restriction prevents the user from inadvertently overwriting one workspace with another.

LIBRARY FULL
Saving is permitted only while the number of workspaces currently saved or the space used by the saved workspaces is within the user's allocation. Quotas are set by the system administrator.

WS FULL
The workspace contains a shared variable whose value, when brought in to the workspace, would require for its storage more work area than is available in the workspace.

)*DROP* **[library] workspace [:[password]]**

This command discards the saved copy of a specified workspace, with no effect on the active workspace.

On some systems, the workspace password is needed to drop the workspace.

)*LOAD* **[library] workspace [:[password]] [size]**

This command causes a duplicate of the indicated workspace (including its entire computing environment) to become your active workspace. The original copy of the workspace on the permanent storage device remains intact and in place. Everything that had previously been in your active workspace is discarded. Shared variable offers in the former active workspace are retracted. Following a successful)*LOAD*, the system reports the time and date at which the loaded workspace was last saved. This time stamp is subject to the time zone variable, ⎕*TZ* (see page 148). The system then immediately executes the latent expression (⍎⎕*LX*). A description of the latent expression may be found on page 142.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*LOAD* command.

Invalid requests to load a workspace may result in the following trouble reports:

WS NOT FOUND
The indicated workspace cannot be found.

WS LOCKED
The password supplied in the command does not match the password of the saved workspace, or is missing from the command when required.

IMPROPER LIBRARY REFERENCE
The user is ineligible to use the indicated library.

Realize that)*LOAD*ing a workspace and)*COPY*ing everything from a workspace are *not* the same thing. ...Read on.

)*COPY* **[library] workspace [:[password]] [names]**

The)*COPY* command is used to bring any APL *global* objects (functions, operators, and variables) from any selected workspace into the active workspace, and merge them with the existing objects already in your active workspace. If a name conflict occurs, the previously-existing object will be replaced by the new object. This point will be discussed in a little more detail below.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*COPY* command.

If no "library" is specified, your own library is assumed. The "names" parameter is to be a list of the names of the desired objects, separated by blanks. For example:

Copy from this workspace...

....these objects

```
      )COPY REPORT1 PRINT PRINTER TERMINAL SALES
SAVED   5/19/1984   7.24.39 (GMT-5)
```

The names of functions, operators, and variables may be intermixed in any order.

You may also copy "groups" of objects... the facility for this has changed quite a bit from earlier implementations. Refer to the discussion of "Grouping to Facilitate Copying and Erasing" on page 89 for a discussion of "Indirect Copy."

If there is *no* list of "names" following the *)COPY* command, the entire contents of the workspace specified will be copied into the active workspace.

Copy from this workspace...

....*all* of the objects
that exist in it.

```
      )COPY REPORT1
SAVED    5/19/1984    7.24.39 (GMT-5)
```

If there is insufficient space to copy in one of the objects that you asked for, the copy command will continue to the next object, and so forth, until all of the objects that will fit are brought in to the active workspace. If some of them could *not* be brought in because of a space problem, you may receive a message of *WS FULL.*

If a name conflict occurs, the previously-existing object will be replaced by the new object. That is, if you already have an object called (for example) "*PRINT*" in your workspace, and you copy in another object of the same name (through either of the two forms of the copy command), the old use of the name will be abandoned, and the new object will be established in the workspace. If you wish to circumvent this possibility of damage to existing objects, refer to the *)PCOPY* (Protected Copy) command, following. It is possible that the name classification will have changed in the process; that is, you can replace a function with a variable of the same name, and so forth.

Now, what happens if the active workspace is *empty*, and you copy in the *entire contents* of another workspace? Okay, let's consider...

Copying an Entire Workspace Into a Clear Workspace

Are these two sequences synonymous?? NO!!

```
)CLEAR
)COPY REPORT1
)WSID REPORT1
```

```
)LOAD REPORT1
```

If you clear the active workspace, copy an *entire* workspace into it, and then name it with the same name as the workspace from which you copied, you might suppose that that sequence would give you the same end result that you'd get if you had simply loaded that workspace. ...Sorry, you didn't win the steak knives this time. There are some differences which, on occasion, may be useful. They depend upon what is *not* copied when you copy an entire workspace. Let's review that a bit:

What is copied:

- All global variables, functions, and operators from the source workspace.

What isn't copied:

- Any local variables, functions, or operators.
- The state indicator (the list of where execution had been halted in the source workspace, described more fully under the)*SI* command on page 83).
- The symbol table (an internal dictionary by which APL identifies and locates each of the names that you have used in the workspace).
- The system variables: Distinguished names (those which start with a "⎕") are never copied. Many of the system variables are set only by the system, and a few will persist throughout the session, regardless of when you)*LOAD* or)*CLEAR*. (These distinctions are discussed on page 136.) But a few of them are ones for which you may have set up some special values; don't forget about them if you ever copy the entire workspace. They are:

⎕*CT*	—	Comparison Tolerance
⎕*FC*	—	Format Control characters
⎕*IO*	—	Index Origin
⎕*LX*	—	Latent Expression
⎕*PP*	—	Printing Precision
⎕*RL*	—	Random Link
⎕*SVE*	—	Shared-Variable Event

In addition to these differences, it should be pointed out that)*COPY* for an entire workspace causes the system to do *much* more work {*huff*puff*} than a simple)*LOAD* requires. If all you want to do is get access to the objects in the workspace, and you aren't trying to combine it with another workspace, use)*LOAD*.

)*PCOPY* [library] workspace [:[password]] [names]

)*PCOPY* (Protected Copy) works in just the same way as the)*COPY* command (discussed on page 77), *except* that)*PCOPY* will alert you to name conflicts and refuse to overlay an existing object with another object of the same name. If that situation arises, you will be given a report of "*NOT COPIED:*", followed by a list of the names under concern.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*PCOPY* command.

)*MCOPY* [library] workspace [:[password]] [names]

The "*M*" in)*MCOPY* stands for "migrate," so as you might expect, the command provides a means of migrating data into APL2 from other versions of APL. The)*MCOPY* command is just like)*COPY* except that the objects copied come from a VSAPL workspace. If you specify)*MCOPY* without a list of objects, you will get the entire workspace, just as with)*COPY*. In that case, the following system variables are also copied: ⎕*IO*, ⎕*RL*, ⎕*CT*, ⎕*PP*, and ⎕*LX*.

)*IN* filename [list]

The)*IN* command reads a transfer file, reconstructs each of the items on the file into APL objects (functions, operators, and variables), and establishes them in the active workspace. It merges them with the existing objects already in your active workspace. If a name conflict occurs, the previously-existing object will be replaced by the new object. This point will be discussed in a little more detail below.

For a discussion of what a transfer file *is* (and why you might want one), read "What's a Transfer Form," under the description of the transfer form system function, ⎕*TF* , on pages 132-135.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*OUT* command.

A transfer file may be created by using)*OUT* (described on page 81), by using auxiliary processors, or by a process external to APL.

)*IN* is quite similar to the)*COPY* command (described on pages 77-79), except that the)*COPY* command moves APL objects from a stored workspace, and the)*IN* command moves transfer forms from a transfer file. There is *no* form of the)*IN* command which corresponds to the)*PCOPY* command.

If a "list" of APL names is supplied, only those objects will be brought into the workspace. If no "list" is supplied, everything on the file is retrieved.

Normally,)*IN* displays no response — there is, for example, no *SAVED* message, like the)*LOAD* command displays. Difficulties encountered during the execution of the command, however, may result in the following trouble reports:

> *NOT FOUND*
> The file doesn't exist (or it exists, but isn't a proper transfer file); no objects will be transferred.
>
> *NOT FOUND*: name list
> Some of the names in an explicitly-stated list of objects could not be found; the ones that were found were established in the workspace.
>
> *NOT COPIED*: name list
> Some of the names in an explicitly-stated list of objects were found, but were determined to not be valid transfer forms; the ones that were found to be acceptable were established in the workspace.

If there is insufficient space to establish one of the objects that you asked for, you may receive a message of *WS FULL*, causing the transfer of objects to be halted.

If a name conflict occurs, the previously-existing object will be replaced by the new object. That is, if you already have an object called (for example) "*PRINT*" in your workspace, and you use)*IN* to bring in another object of the same name, the old use of the name will be abandoned, and the new object will be established in the workspace. It is possible that the name classification will have changed in the process; that is, you can replace a function with a variable of the same name, and so forth.

The format of the *file name* is system-dependent. Refer to *APL2 Programming: System Services Reference* for specific information. And for assistance in migrating data into APL2 from other versions of APL, refer to *APL2 Migration Guide*.

)*OUT* filename [list]

The)*OUT* command converts APL objects (functions, operators, and variables) into their transfer form, and transfers duplicate copies of them from the active workspace to a transfer file. The original objects in the active workspace remain intact and unscathed.

For a discussion of what a transfer file *is* (and why you might want one), read "What's a Transfer Form," under the description of the transfer form system function, ⎕*TF*, on pages 132-135.

Refer to the diagram on page 69, explaining "The Effects of Selected System Commands," for a pictorial view of the action of the)*IN* command.

A transfer file may be read back in by using)*IN* (described on page 80).

)*OUT* is somewhat similar to the)*SAVE* command (described on page 76), except that the)*SAVE* command stores an entire APL workspace in a specific format meaningful only to APL, and the)*OUT* command stores transfer forms of APL objects on a transfer file.

If a "list" of APL names is supplied, only those objects will be stored on the file. If no "list" is supplied, all of the user-defined objects in the active workspace (functions, operators, and variables) except shared variables will be copied to the file. In either form, the most local version of each object that is currently active will be transferred.

If you explicitly state the names of system variables in the name list, their settings may also be stored on the file. Use of the command without the name list will not transfer system variables.

If any names are explicitly requested but are not found in the active workspace, or if they are found but are not appropriate for transfer (for instance, if they name a shared-variable), you will be given a message of "*NOT COPIED:* ", followed by a list of those names.

Entering the command either with or without a list of object names will cause a new file to be created, or will completely over-write an existing file. There is no provision for adding to an existing file.

In CMS, the proper format is ")*OUT* fn.ft.fm" (where fn = filename, ft = filetype, and fm = filemode). If filemode is not specified, *A* is assumed. If filetype is not specified, "*APLTF.A*" is assumed.

In TSO, "*)OUT A.B*" writes file "userid.*APLTF.A.B*", while "*)OUT 'A.B'*" writes file "*A.B*".

Refer to *APL2 Programming: System Services Reference* for more host system file naming information.

)ERASE **names**

The *global* objects named are expunged from the active workspace; shared variable offers with respect to any of them are retracted.

If an object named in the command cannot be found, the report "*NOT ERASED:*" is given, followed by a list of the objects not found.

If the name of a suspended or pendent function is specified with the *)ERASE* command, only its global definition is erased. The copy of the function that is currently halted is not affected, and execution may resume normally.

It may be instructive to compare the *)ERASE* command to the *⎕EX* system function (on page 127). One major difference (other than form) is that *)ERASE* discards only global references, while *⎕EX* discards *the most local* reference to a name that is currently active.

)OFF [*HOLD*]

This command discards the active workspace and terminates the current APL session. An accounting report is displayed, showing the connect time and computing time used during the current session. Some systems may include some additional accounting information.

The *HOLD* parameter has no effect and is included for compatability with older APL products.

)CONTINUE [*HOLD*]

This command provides the same effect as *)WSID CONTINUE*, followed by *)SAVE*, followed by *)OFF*... your active workspace is saved under the name "*CONTINUE*," and your APL session is terminated. When your next APL2 session is started, the *CONTINUE* workspace will be automatically loaded, unless the *INPUT* parameter was used in your invocation statement.

An accounting report is displayed, showing the connect time and computing time used during the current session. Some systems may include some additional accounting information.

The *HOLD* parameter has no effect and is included for compatability with older APL products.

The *)CONTINUE* command is often helpful when you need to end your current session, but wish to easily "continue" pursuing the same task when you return.

The *CONTINUE* workspace is otherwise much the same as your other workspaces. In particular, you may *)LOAD* and *)SAVE* and *)DROP* it in the same fashion as any other workspace.

When you resume work in your following session, it's recommended that you rename the *CONTINUE* workspace and re-*)SAVE* it under its original name whenever you modify the items that you were working on. In this way, the chance of accidentally overlaying it with unrelated material through the use of a later *)CONTINUE* command will be lessened.

)SI • *)SINL* • *)SIS*

These commands display the *state indicator*, showing the status of halted statements from immediate-execution mode, or from halted functions and operators, with the most recently halted one shown first. The list shows the name of the function or operator (where applicable), and the number of the statement at which work is halted.

Suspended statements are marked in the state indicator by an asterisk, while pendent statements appear on the state indicator list *without* an asterisk. Damage to the state indicator is shown by a statement number of ¯1 alongside the name of the affected function or operator name.

The *)SINL* (*"state indicator with name list"*) command displays the state indicator in the same way as *)SI* does, but in addition, with each function or operator listed, lists the names that are local to its execution.

The *)SIS* (*"state indicator with statements"*) command displays the state indicator in the same way as *)SI* does, but in addition, with each function or operator listed, it displays the statements that were halted, and the associated error carets showing where the halt occurred, and how far through the line the execution had proceeded.

A Mystification to Avoid

Every now and again, an APL user forgets to tell the system what should be done with a function whose execution has been suspended. If the suspended function uses a local variable whose name is *also* used for a global variable or function, you *may* think that you're referring to the *global* name, and instead, you're getting the value of the local variable from within that suspended function. Fortunately, the problem is easily avoided: don't leave suspended executions hanging around unresolved any longer than necessary.

Any time that you encounter an error, you should take *some* action. The whole purpose of the suspensions, of course, is to give you a chance to *fix* the errors and be able to resume the execution if you want to, without having to retype everything or re-execute everything.

To restart execution, enter "→" followed by the line number in the most recently-suspended function or operator that you wish to begin re-executing; for example, to restart the execution at line 5, type "→5". If you want to restart the same line on which the most recent problem had occurred (perhaps you've fixed the source of the problem), enter "→⎕*LC*", meaning restart the last line that had been executing. Alternatively, "→⍳0" means "resume from where you stopped," even within the middle of a line.

If you don't want to resume at all, that's fine too, but you should tell the system that.

Entry of a single right-arrow, "→", followed by ENTER, will *cancel* the most recent suspension, back to the point of the keyboard input that invoked it, and *)RESET* will cancel *all* of the suspensions in the workspace.

For some additional discussion of errors, refer to the "Display of Errors" portion of the "Event Handling" section, on pages 161-166.

)*RESET* [number]

The)*RESET* command with no number after it empties the state indicator of all entries. All suspensions are cleared, and ⎕*EM*, ⎕*ET*, ⎕*L*, and ⎕*R* are reset. For all of those people who kept meaning to get back to the state indicator and clear up those suspended entries, this should be easier than (but equivalent to) entering "→" and pressing "ENTER" for every suspension. A procrastinator's delight.

)*RESET* followed by a number *n* will cause the state indicator to be cut back *n* levels (that is,)*SI* will have *n* fewer lines). This lets you cut back the stack to the place *you* want.

)*MSG* message •)*MSG ON* •)*MSG OFF* •)*MSGN* message

This command sends a message to the terminal of the user indicated. The designated user must be currently signed on to the same APL system. The method of specifying the user differs between systems; it may consist of the user's sign-on or a port number (phone-line connection number). Refer to *APL2 Programming: System Services Reference* for specific information.

Proper transmission of the message will result in the message *SENT* being displayed at your terminal when the message is received.

On some systems,)*MSG* will lock your keyboard until a response is received from the intended recipient. To prematurely abandon the wait, press the ATTENTION key.

)*MSG OFF* specifies that no subsequent messages are to be received from any user. This may be helpful, for example, during the printing of a report, so that an incidental message can't disturb your output. Use this with caution, though, since)*MSG OFF* will prevent the reception of *all* messages, including broadcast messages from the system operator (perhaps warning of an impending unscheduled system shutdown!).

An attempt to send a message to a user who is not currently signed on or who has set)*MSG OFF* will result in a display of a *NOT SENT* message.

)*MSG ON*, of course, is used to reverse the action of)*MSG OFF* (...you probably could have guessed that).

The)*MSGN* form of the command is identical to)*MSG*, *except* that it does not attempt to lock the keyboard in anticipation of a reply. On some systems,)*MSG* and)*MSGN* are identical. Again, refer to *APL2 Programming: System Services Reference* for specific information.

The *ON* and *OFF* options are not used with)*MSGN*.

)*OPR* message •)*OPRN* message

This command sends a message to APL System Operator. Proper transmission of the message will result in the message *SENT* being displayed at your terminal when the message is received.

On some systems,)*OPR* will lock your keyboard until a response is received from the intended recipient. To prematurely abandon the wait, press the ATTENTION key.

The)*OPRN* form of the command is identical to)*OPR*, *except* that it does not attempt to lock the keyboard in anticipation of a reply. On some systems,)*OPR* and)*OPRN* are identical. Refer to *APL2 Programming: System Services Reference* for specific information.

)*HOST* [command]

The)*HOST* command is used to send a command to the operating system (CMS or TSO). For example, in CMS you could list the number of people using the computing system like this:

```
          )HOST Q U
537 USERS, 009 DIALED, 000 NET
```

)*HOST* with no argument reports which Operating System you are on:

```
          )HOST
IS CMS
```

| **)*EDITOR* [number] or [name]**

The)*EDITOR* command is used to select which of the system editors you prefer for editing functions, operators, and variables.

- ")*EDITOR* 1" specifies the standard "del"-editor. It is the default at sign-on time.
- ")*EDITOR* 2" specifies the extended full-screen editor.
- ")*EDITOR* **name**" specifies the use of a system editor (such as CMS XEDIT). "**name**" is the name of a CMS EXEC on a TSO CLIST.

Used without a number, this command can be used for inquiry; it will return the number of the editor currently selected. The editor number persists over a workspace)*LOAD* or)*CLEAR*.

For a complete discussion of Editors 1 and 2, refer to "Defining Your Own Functions" on pages 104-115.

)*MORE*

The)*MORE* command will provide additional information after an error occurs (if the additional information exists). Generally there will be more information after an error that occurs during a library operation such as)*LOAD* or)*SAVE*, or if an auxiliary processor cannot handle a request.

)*QUOTA*

Each APL user is given quotas controlling how much of the system's resources she can use. These quotas control such things as the total number of (or combined storage capacity of) workspaces that may be saved concurrently, and the number of variables that may be shared concurrently. On systems that support variable workspace sizes, there is generally also a quota controlling the maximum size of the active workspace.

The)*QUOTA* command displays these quotas. While the form of the display and the meaning of the quotas displayed varies from system to system, here is one form:

```
         )QUOTA
LIB        total  FREE remaining
WS       default  MAX    maximum
SV        number  SIZE      size
```

Meaning: [All of the space measurements are in bytes]

"*LIB*" is the total amount of space that's currently in use for storing workspaces in your library.

"*LIB FREE*" is the amount of space that's left in your library.

"*WS*" is the size of your active workspace at sign-on time.

"*WS MAX*" is the size of the largest workspace that may be requested (as with the)*LOAD* or)*CLEAR* commands).

"*SV*" is the maximum number of shared variables that you may tender simultaneously.

"*SV SIZE*" is the size of the shared memory.

Here's a sample display:

```
         )QUOTA
LIB      6420480    FREE      520192
WS        753664    MAX       753664
SV            88    SIZE       32768
```

)*SYMBOLS* **[number]**

This command sets the size of the symbol table; that is, the maximum number of names that may occur in the workspace. (Note that the occurrence of a name includes not only the names of functions, operators, and variables themselves, but also any names occurring within their definitions.) New values of the maximum may be overridden by the system, and increased automatically as needed. An attempt to set the maximum outside the range permitted by the system is rejected with the report "*INCORRECT COMMAND*." There is no response for a valid setting.

Used without specifying a number, the)*SYMBOLS* command will display the number of symbols that are currently in use.

)*TIME*

The)*TIME* command will display the current date and time of day and the offset from Greenwich Mean Time. The format of the response is established when APL2 is installed on your computing system. The three possible formats are:

`1984-02-17 11.39.51 (GMT-4)`	The ISO standard
`02/17/1984 11.39.51 (GMT-4)`	The U.S. convention
`17.02.1984 11.39.51 (GMT-4)`	The European convention

)*PBS* [*ON*/*OFF*]

APL2 has seven new characters which weren't available in previous versions of APL. These are:

Character Name	Symbol	Formed by overstriking
Quad Slope	⍂	⎕ \
Quad Jot	⌻	⎕ ∘
Left Bracket Right Bracket	⌷	[ ]
Equal Underbar	≡	= _
Epsilon Underbar	⍷	∊ _
Iota Underbar	⍸	⍳ _
Dieresis Dot	∵	¨ .

On terminals that support overstrikes, these seven characters may be entered simply by overstriking the appropriate characters. On a 2741, for instance (remember those?), the “"” character may be formed by entering “⎕”-backspace-“\”. However, not all terminals allow backspacing and overstriking to produce compound characters. [For example, one terminal which cannot form the new characters directly is the IBM 3277.] Therefore, a new system command has been provided so that users of these terminals can have some means of entering these new characters.

The)*PBS* command is used to enable a *P*rintable *B*ack *S*pace character, “_”. This character can then be used as a logical backspace so that the new characters may be entered or displayed. This logical backspace has meaning *only* with the seven characters shown in the table.

With)*PBS ON*, one of the new characters may be specified as a three-character sequence consisting of the two existing APL characters with the character “_” inserted between them.

With)*PBS OFF*, the new characters are also displayed by APL2 as a sequence of three characters: the two characters which are overstruck to produce the new character, separated by the character “_”.

For example:

```
      )PBS ON
      A←'⎕_∘∊__ι__[_]⎕_\=__._¨'
      ⍴A
7
      ⎕AV⍳A
116 118 117 205 207 226 237
      A
⎕_∘∊__ι__[_]⎕_\=__._¨
```

Either of the two characters which make up each overstrike may come first.

```
      '⎕_∘' = '∘_⎕'
1
```

You can "undefine" the)*PBS* character by entering ")*PBS OFF*". If you attempt to display the new characters when there is no *PBS* character defined, the characters may display as other, terminal dependent, characters:

```
      A
⎕_∘∊__ι__[_]⎕_\=__._¨

      )PBS OFF
      A
*******
```

Grouping to Facilitate Copying and Erasing

Indirect *COPY* and Indirect *ERASE*:

It is frequently convenient to copy into the active workspace several related functions and variables, and to erase them when they are no longer needed. To facilitate such transfers, a *group* may be defined by supplying a list of names that are to be copied or erased together. The definition of a group consists of a list of names. It is not necessary that objects having those names exist in the workspace.

In previous versions of APL, groups were a special type of object, requiring the use of several system commands to list and update them. Now, in APL2, they no longer require such special handling — any APL variable which consists of a simple character matrix containing a list of names (with one name per line) may be treated as a group. (Also, in keeping with the rules for such functions as ⎕*NC*, the list may be a simple character vector or scalar containing just one name.) This allows you to use all of the power of APL to create and maintain these lists dynamically.

For example, let's assume that we have a workspace that looks like this:

```
      )WSID
REPORT1

      )FNS
DISPLAY NEWPAGE PRINT   PRINTER REPORT  TERMINAL

      )VARS
HSPTR   DATA    GRPPRINT        SALES   TABLES
```

Now, let's look at that variable called "*GRPPRINT*":

```
      ⍴GRPPRINT
6 8
      GRPPRINT
GRPPRINT
PRINT
PRINTER
TERMINAL
NEWPAGE
HSPTR
```

In this case, the variable named "*GRPPRINT*" contains the names of six objects— not quite all of the functions, but of course, it could contain whatever you wish.

Now let's load another workspace, and see how that group can help us:

```
        )LOAD ANOTHER
SAVED    5/23/82  19.16.43 (GMT)
```

Now we want to copy some of those functions from that *REPORT*1 workspace into this workspace; in particular, we want the functions that make up the "print group," along with a couple of other stray objects... that's easy:

Copy from this workspace...

....each of these APL objects....

```
        )COPY REPORT1 SALES DATA (GRPPRINT) TABLES
SAVED    5/19/82  12.24.39 (GMT)
```

...and the *objects named in the list* called "*GRPPRINT*"; the parentheses cause this to be an *indirect copy.*

When a *)COPY*, *)PCOPY*, or *)ERASE* command mentions a name that is contained within parentheses, then all global objects whose names appear in the membership list are copied (or erased, as the case may be).

If the membership of a list *A* includes the name *B* of such list, the act of copying *A* causes the membership list of *B* to be copied, but copying does not extend further. That is, the objects referred to in the membership list of *B* are not copied. The same applies to erasure.

Those items that we copied into the active workspace could be *erased* with this command:

```
        )ERASE DATA SALES (GRPPRINT)
```

By the way, APL2 makes it easy to create the character matrix for the group — just use disclose on a vector of vectors:

```
        GRPLIST←⊃'NAME1' 'ANOTHER' 'LAST'
        ⍴GRPLIST
 3 7
```

It's also easy to modify a group definition — just use *)EDITOR* 2 to add, delete, or change names in the character matrix.

* * *

For additional information on system commands, refer to *APL2 Programming: System Services Reference*.

Used with the permission of The Dick Sutphen Studio.

Function and Operator Definition

While this section will speak of "functions" throughout, these discussions apply equally to defined operators.

There are three ways in which a defined function can be established in an APL workspace:

1. It can be loaded or copied from a stored workspace using a system command (see pages 65-90).

2. It can be established in execution mode, using the system function "fix" (⎕*FX*), either by a direct keyboard entry or by another defined function (see page 128).

3. It can be established in function definition mode, using one of the system's "del"-editors (see pages 104-115).

Regardless of which facility has been used for establishing a function, its definition can be displayed or modified in either the function definition mode, in which certain editing capabilities are built-in, or by the combined use of the system functions "Canonical Representation" (⎕*CR*) and "fix" (⎕*FX*).

Canonical Representation and Fix

The *character representation* of a function is a character matrix satisfying certain constraints: the first row of the matrix represents the *function header* and must be one of the forms specified in the discussion of function headers on page 93. The remaining rows of the matrix, if any, constitute the *function body*, and may be composed of any sequence of characters. If the character representation satisfies additional constraints (such as no redundant spaces and left justification of the nonblank characters in each row), it is said to be a *canonical representation*. The canonical representation of a function, then, is the minimum form of display which will completely define a function; it is devoid of such decorations as line numbers.

Applying ⎕*CR* to the character array (scalar or vector) representing the name of an already established function will produce its canonical representation.

For further discussion of the use of ⎕*CR*, see the discussion of "Canonical Representation" on page 125.

The "fix" function, ⎕*FX*, provides a facility for dynamically creating APL functions and operators. ⎕*FX* is the inverse of ⎕*CR*. A typical input (right argument) to ⎕*FX* would be a matrix of characters such as ⎕*CR* produces. When ⎕*FX* is applied to a character matrix which is the canonical representation of some function, it will cause that function to be established in the workspace. The explicit result of ⎕*FX* is the name of the function (as a character vector) or a number (representing an invalid line in the matrix) if the function couldn't be created.

For a discussion of the rules governing the use of ⎕*FX*, see the discussion of "fix" on page 128.

Function and Operator Headers

The *valence of a function* is defined as the number of explicit arguments that it takes. A defined function may have any one of six forms of header, as follows:

Function Header:	*FN* ←→	the function name
	L ←→	the left argument
	R ←→	the right argument
	Z ←→	the explicit result

Type	Valence	With Result	No Result
Niladic	0	*Z←FN*	*FN*
Monadic	1	*Z←FN R*	*FN R*
Dyadic	2	*Z←L FN R*	*L FN R*

The *valence of an operator* is defined as the number of explicit operands which it takes. Its derived function, however, may have a different valence. A defined operator may have any one of eight forms of header, as follows:

Operator Header:	*OP* ←→	the operator name
	L ←→	the left array argument
	R ←→	the right array argument
	F ←→	the left function or array operand
	G ←→	the right function or array operand
	Z ←→	the explicit result

Number of Arguments/Operands		With Result	No Result
1	1	*Z←(F OP) R*	*(F OP) R*
1	2	*Z←(F OP G) R*	*(F OP G) R*
2	1	*Z←L (F OP) R*	*L (F OP) R*
2	2	*Z←L (F OP G) R*	*L (F OP G) R*

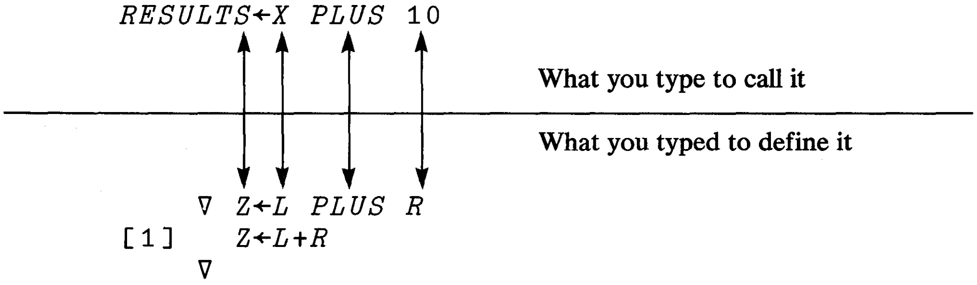
Arguments and Operands

The names in the header of a defined function which pass data into that function are called its *arguments*. The names in the header of a defined operator which pass functions or data into that operator (entered within parentheses in the header) are called its *operands*.

The names used for the arguments and operands of a function or operator become *local* to that function or operator, and additional local names may be designated by listing them after the function name and argument, with a semicolon preceding each local name; the name of the function is *global.* The significance of these distinctions is explained in "Local and Global Names" on page 97.

A name may not be meaningfully repeated in the header, except for the function name itself, which may be repeated in the list of local names. It is not obligatory either for the arguments of a defined function to be used within the body, or for the result variable to be specified in the course of function execution. Although it would be unusual for either of those cases to occur, and would probably offer no advantages, those cases do not result in errors.

The names of the arguments have meaning only within the function. Outside of the function definition, their only significance is a positional one.



```
      RESULTS←X PLUS 10
```

What you type to call it

What you typed to define it

```
    ∇ Z←L PLUS R
[1]   Z←L+R
    ∇
```

When this *PLUS* function is invoked with the top line, its left argument, *L*, takes on the value of the data in *X*, and its right argument, *R*, takes on the value 10.

Explicit Results

At the conclusion of its execution, whatever value that had last been placed in its local result, *Z*, will be passed out of the function to the variable *RESULTS*. When a name is assigned (using a specification arrow, "←") *both* in the header of a function *and* somewhere within the body of that function, the function is said to have an *explicit result...* "explicit" in that it is explicitly available for use as part of a larger expression. *Only* if the function is defined with an explicit result may the function be used with more of an APL statement to its left:

```
    ∇ Z←L PLUS R
[1]   Z←L+R
    ∇

      10 PLUS 20
30

      2×10 PLUS 20
60
```

```
    ∇ L PLUS R
[1]   L+R
    ∇

      10 PLUS 20
30

      2×10 PLUS 20
30
VALUE ERROR
      2×10 PLUS 20
       ∧   ∧
```

In the right-hand example, the only thing that line 1 of the function defined was the addition of the two variables, *L* and *R*. Since there was no assignment of the result to another variable, the default was to print it, which it did (making the two forms shown look identical for trivial cases). Since there is no explicitly available result from the *PLUS* function, there's nothing to pass along to be used as the right argument for the "×" function. The explicit result in the left-hand example makes that value available. The value that gets passed along as the result of the function is whatever value was last assigned to the explicit result variable (in this case, *Z*) when the function has finished its execution. The assignment does not have to be made at any special time during the function's execution.

Ambi-Valent Defined Functions

The *valence* of a function is a count of its explicit arguments: a monadic function has a valence of one, and a dyadic function has a valence of two. An *ambi*-valent function, then, is one which may be used with *both* valences.

A dyadic user-defined function may be invoked either with or without its left argument. This allows you to write functions that more closely resemble the operation of primitive functions. Within the function, you must then check to see if the left argument was supplied before you reference its value; you can do this with ⎕*NC* (name classification, described further on page 130).

To illustrate how (and why) ambi-valence might be used, let's start with a simple function for finding the n-th root of a number:

```
      ∇ Z←N ROOT A
[1]     Z←A*÷N
      ∇

        2 ROOT 64 729 4096
8 27 64

        (2 ROOT 64 729 4096)*2
64 729 4096

        3 ROOT 64 729 4096
4 9 16

        2 3 ROOT 16 125
4 5
```

Perhaps a form that would be more convenient to use would be one which would use a common default value for the left argument if no value is supplied; let's assume, for instance, that we would usually use this for finding square roots. If the function is called *without* a left argument, the name used for the left argument ("*N*") would have no value. This can be checked using ⎕*NC*, like this:

```
     ∇ Z←N ROOT A
[1]    →(0≠⎕NC 'N')/RN
[2]    N←2
[3]   RN:Z←A*÷N
     ∇
```

{or}

```
     ∇ Z←N ROOT A
[1]    ⍎(0=⎕NC 'N')/'N←2'
[2]    Z←A*÷N
     ∇
```

```
      2 ROOT 64 729 4096
8 27 64

      ROOT 64 729 4096
8 27 64

      3 ROOT 64 729 4096
4 9 16
```

Notice that *ROOT* now works with or without a left argument, and uses 2 as a default if the left argument is elided.

Ambi-valence can be used to supply a commonly-used value by default, as we did here, or it can supply an argument which would otherwise require cumbersome entry. For example, assume that you are using a *FIND* function which will look through selected functions (whose names are listed in its left argument) for a character string (which is specified in its right argument). To tell it that you want to look through *all* of the functions in the workspace, it may be necessary to enter something like `(⎕NL 3) FIND 'CHAR STRING'`. Most APL authors have long recognized common cases like this, and have provided a shorthand notation: `'' FIND 'CHAR STRING'`. Now, using ambi-valence, the notation becomes one step easier: `FIND 'CHAR STRING'`. Ambi-valence, then, is a useful tool for situations where you may frequently want to indicate *all values.*

No Two Ways About It, ALL Functions Are Ambi-Valent

If you accidentally call a dyadic defined function without its left argument, it *used* to respond with a *SYNTAX ERROR* in previous versions of APL. Now it will invoke the function, and (if you haven't provided for it) will produce a *VALUE ERROR* the first time that the left argument is referenced. Recognizing this change may save you some time during troubleshooting.

Previous behavior:

```
      ∇ Z←A PLUS B
[1]     Z←A+B
      ∇

      3 PLUS 5
8

      PLUS 5
SYNTAX ERROR
      PLUS 5
      ^

      )SI


      PLUS
SYNTAX ERROR
      PLUS
      ^
```

APL2 behavior:

```
      ∇ Z←A PLUS B
[1]     Z←A+B
      ∇

      3 PLUS 5
8

      PLUS 5
VALUE ERROR
PLUS[1]  Z←A+B
           ^^

      )SI
PLUS[1]  *

      PLUS
SYNTAX ERROR
      PLUS
      ^
```

Notice, in the last example, that ambi-valence does *not* allow the function to be called niladically (that is, with *no* arguments).

Local and Global Names

During the execution of a defined function, it's often necessary to work with intermediate results or temporary functions that have no significance either before or after the function is used. The use of *local* names for these purposes, so designated by their appearance in the function header, avoids cluttering the workspace with a multitude of objects introduced for such transient purposes, and allows greater freedom in the choice of names. Names used in the function body, and not so designated, are said to be *global* to that function.

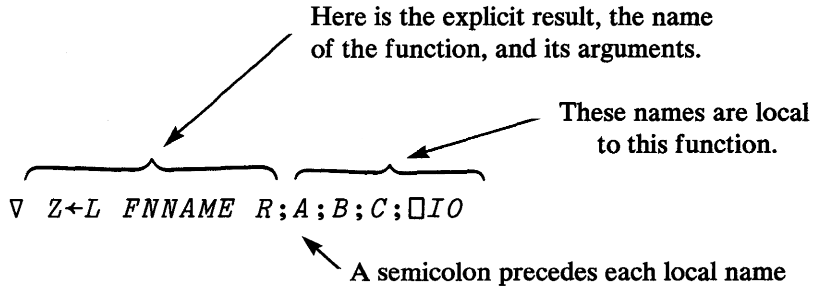
A local name may be the same as that for a global object, and any number of names local to different functions may be the same. During the execution of a defined function, a local name will temporarily exclude from use a global object of the same name. If the execution of a function is interrupted (leaving it either *suspended* or *pendent*), the objects retain their dominant position, during the execution of subsequent APL operations, until such time as the *halted* function is completed. However, system commands continue to reference global objects under these circumstances. (The system editors, by the way, always reference the *most local* copy of an object that's currently active — they're described on pages 104-115.)

Local names in suspended functions can sometimes be a source of confusion. The shadowing of names can alter the operation of a workspace, and leave the user perplexed about what happened. For some discussion of what to do to overcome or prevent this confusion, see "A Mystification to Avoid" on page 83.

The localization of names is dynamic, in the sense that it has no effect except when the defined function is being executed. Furthermore, when a defined function uses another defined function during its execution, a name localized in the first (or outer) function continues to exclude global objects of the same name from the purview of the second (or inner) function. This means that a name localized in an outer function has the significance assigned to it in that function when used without further localization in an inner function. The same name localized in a sequence of nested functions has the significance assigned to it at the innermost level of execution.

The *shadowing* of a name by localization is complete, in the sense that once a name has been localized its global and outer values are temporarily hidden, even if no significance is assigned to it during execution of the function in which it is localized.

Names are made local to a function by placing them on the header line, to the right of one of the eight forms of headers shown above. A semicolon separates the names of the local objects from each other and from the rest of the function header. For example:



Here is the explicit result, the name of the function, and its arguments.

These names are local to this function.

```
∇ Z←L FNNAME R;A;B;C;⎕IO
```

A semicolon precedes each local name

"Semi-Global" Names

This is a contrived term, but it's one that seems to work itself into lots of literature, so let's discuss it a bit. In the most strict sense of the term, a "global name" is one that's not localized at any level; it's active with nothing suspended. To a function that's executing, though, anything that it is able to reference that isn't in its own header is considered to be *global to that function.* A name that has been localized to a function, which is subsequently accessed "globally" by a subfunction, is sometimes referred to as a "semi-global" name.

Branching and Statement Numbers

Statements in a function are normally executed sequentially, from top to bottom, and execution terminates at the end of the last statement in the sequence. This normal order can be modified by *branches*. Branches are used in the construction of iterative procedures, in choosing one out of a number of possible continuations, or in other situations in which decisions are made during the course of function execution.

To facilitate branching, the successive statements in a function definition have reference numbers associated with them, starting with the number one for the first statement in the function body and continuing with successive integers, as required. Thus, the expression →4 denotes a branch to the fourth statement in the function body, and when executed, causes statement 4 to be executed next, regardless of where the branch statement itself occurs. [In particular, →4 may be statement 4, in which case the system will simply execute this "tight loop" indefinitely, until interrupted by an action from the keyboard. This is a trap to be avoided.]

A branch statement always starts with the *branch arrow* (or *right arrow*) on the left, and this can be followed by any expression. For the statement to be effective, however, the expression must evaluate to an integer, to a vector whose first element is an integer, or to an empty vector; any other value results in a *DOMAIN* or *RANK* error. If the expression evaluates to a valid result, then the following rules apply:

1. If the result is an empty vector, the branch is "vacuous" (or empty, meaning that *no* branching occurs), and execution continues with the next statement in the function if there is one, or else the function terminates.

2. If the result is the number of a statement in the body of the function, then that statement is the next to be executed.

3. If the result is a number out of the range of statement numbers in the function, then the function terminates. The number 0 and all negative integers are outside of the range of statement numbers for any function.

Because zero is often a convenient result to compute, and because it is not the number of a statement in the body of any function, it is often used as a standard value for a branch intended to end the execution of a function. It should be noted that in the function definition mode described below, zero is used to refer to the header. This has no bearing on its use as a target for a branch.

An example of the use of a branch statement is shown in the following function, which computes the greatest common divisor of two scalars:

```
      ∇ Z←M GCD N
[1]     Z←M                 ⍝GREATEST COMMON DIVISOR
[2]     M←M|N
[3]     N←Z
[4]     →(M≠0)/1
      ∇
```

The compression function (really a derived function of the slash operator), when it's used in the form "condition/target," or "(M≠0)/1", returns 1 if the condition (M≠0) is true (that is, if *M* is *not* equal to zero), and an empty vector if the condition is false (that is, if *M* *is* equal to zero). Thus, the fourth statement in *GCD* is a branch statement which causes a branch to the first statement (line 1) when the condition *M*≠0 is true (that is, when *M* is *not* equal to zero), and a branch with an empty vector argument (falling through to the next statement), when the condition is false. In this case, there is no next statement and so execution of the function ends.

This form of branching is not the only way to write a branch statement. We'll talk about a couple of different forms, and make a recommendation on style, right after we look into labels.

Labels

If a statement occurring in the body of a function definition is prefaced by a name and a colon, then the name is assigned a value equal to the statement number. A name used in this way is called a *label.* Labels are used to advantage when it is expected that a function definition may be changed for one reason or another, since a label automatically assumes the new value of its statement's line number as other statements are inserted or deleted.

Notice in the *GCD* example, above, that statement 4 branches to line 1 if the condition (*M*≠0) is met. That's okay initially, but if we ever add some additional code to the function *prior* to the line that reads "*Z*←*M*", what's now line one will be automatically given a new line number. That would mean that we would have to remember to change that last line every time that we added something to the top of the function; if the function was very big, that could quickly get out of hand. Therefore, a *much better* approach is to always branch to a line label. Using a label, that preceding function would look like this:

```
      ∇ Z←M GCD N
[1]  LOOP:Z←M          ⍝GREATEST COMMON DIVISOR
[2]    M←M|N
[3]    N←Z
[4]    →(M≠0)/LOOP
      ∇
```

Now if we add some additional lines to the top of the function, the loop will continue to execute in exactly the same way that it does now; whenever the condition for branching is met, the branch at the end of the function will always cause execution to go to the line with the "*LOOP*" label, regardless of what line number it's associated with.

Notice also that line labels are automatically *exdented* by the system editors; that is, they are moved out into the margin by one space, so that you can more easily spot them when you're scanning through the display of a function.

The name of a label is local to the function in which it appears, and must be distinct from other label names in the same function and from the local names in the header.

Note also that a label name may not appear immediately to the left of a specification arrow; that is, it may not be the target of an assignment. In effect, a label acts like a (local) constant.

A Recommended Form of Branching

There are many ways that a branch statement can be written. People have come up with lots of different styles... after all, that's what programming style is all about. Different solutions from different people, for different problems. However, some forms — even some of the common ones — have built-in limitations. For example, a form that seems to get used a lot is of the form "→*LOOP*×⍳*A*=*B*"; in origin 1, this equates to "→*LOOP*" if *A*=*B*, and "→⍳0", or fall through to the next line if *A*≠*B*. Unfortunately, if the function is ever run in origin 0, you're in trouble, because the *A* = *B* case will evaluate to "→0", meaning "exit from the function". The "× ι" form of branching is *not recommended.* You will hear people make statements that this form (or whatever form) is "faster than other forms." That's a laudable goal, but it's also strictly implementation dependent. What's fast on this system may not be fast on another system, and so forth.

A *recommended form* of branching is the form used in the example above; that is, "→(condition) / label"... for instance, "→(*A* = *B*) / *LOOP*". If *A* = *B*, this statement becomes "→1 / *LOOP*", or "→*LOOP*". If *A* ≠ *B*, this statement becomes "→0 / *LOOP*", or "→ι 0", meaning "fall through to the next line."

One advantage of this form over some of the other forms is that this one is origin-independent; you can use origin 1 or origin 0 without changing the statement. Another advantage of this form over *many* of the other forms is that this one form will handle both the simple branching cases like we have just discussed, and "n"-way branching with equal ease.

Writing n-Way Branch Statements

Sometimes it's desirable to be able to let a function choose *which* label it should branch to; there may be several potential paths. This is easily done with compression branching, like this:

```
[7]   →( (A<B) , (A=B) , A>B ) / LESS , EQUAL , MORE
```

This line produces a three-way branch. In this particular statement, one of the conditions will be true, and the rest will be false. The true condition (expressed as a 1) will select the corresponding label. Obviously, the list of labels is expected to be the same length as the list of conditions.

```
[12]  →( N≥90 70 60 50 ) / SUPER , GOOD , FAIR , POOR
[13] FAIL:
```

This line can branch to any of four labels, or can fall through to the "*FAIL*" label. In this statement, *one or more* of the various conditions listed may be true (yielding a one). The resultant vector of ones and zeros will be used to compress the list of labels. Typically, this statement will evaluate a branch to a *list* of labels; that is, it may evaluate like this:

```
This line:     →(N≥90 70 60 50)/SUPER,GOOD,FAIR,POOR
evaluates to:  →(75≥90 70 60 50)/SUPER,GOOD,FAIR,POOR
evaluates to:  →     0  1  1  1 /SUPER,GOOD,FAIR,POOR
evaluates to:  →                      GOOD,FAIR,POOR
evaluates to:  →                      17   19   27
evaluates to:  →17
```

As shown here, a branch to a list of labels (or a vector of numbers) is the same as a branch to the *first* (left-most) value in the list. Therefore, an n-way branch will always branch to the label (or line number) that corresponds to the *left-most* "true" condition.

If *none* of the conditions were true (as would be the case here if *N* were less than 50), the execution would fall through to the next line (in this case, at the *FAIL* label).

Doing the branching with *compression* allows you to use one simple form for *all* of your branching requirements. That will save you time in writing — and reading — those statements later on.

Comments

The lamp symbol, ⍝ (the cap-jot), signifies that what follows it is a comment, for illumination only and not to be executed; it may appear on a line by itself, or may appear to the right of any expression or label.

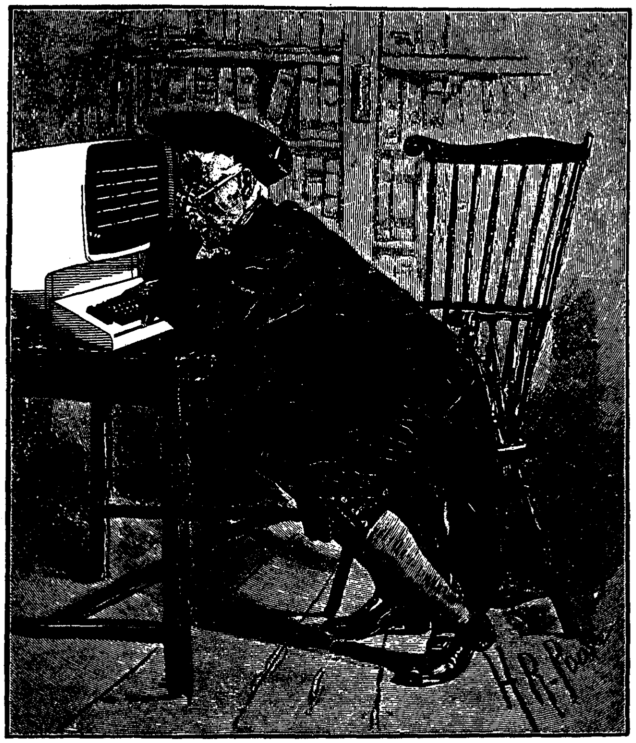
Comments are visible only when the function or operator is displayed through the use of one of the system editors or ⎕*CR*; they do not display when the function is executed.

Spaces within comments are left just where you enter them; APL2 doesn't touch them at all. Also, the *leading* blanks between the code and the comment are significant. These leading blanks are considered to be part of the comment (even though they appear before the lamp symbol), and APL2 leaves them alone:

```
      ∇ FOO
[1]   ⍝COMMENTS WITH NO CODE TO THE LEFT GET EXDENTED
[2]   ⍝(MOVED LEFT BY ONE POSITION) --JUST LIKE LABELS
[3]   ⍝DO-- SO THAT YOU CAN FIND THEM EASILY WHEN YOU
[4]   ⍝READ THE FUNCTION.
[5]   ⍝
[6]    C←0            ⍝ SPACES ON BOTH SIDES OF A COMMENT
[7]    H←⍳1↑⍴M        ⍝ SYMBOL ARE SIGNIFICANT, SO THAT
[8]   LOOP:C←C+1      ⍝ YOU CAN LINE UP YOUR COMMENTS.
```

A statement may be composed of a *label*, an *expression*, and a *comment*. Any of these three items may be elided; however, when they are present, they must appear in that order.

If you feel up to wading through some editorial comments on comments, see the discussion entitled "Where's This Function Going?" on pages 228-229.



Used with the permission of The Dick Sutphen Studio.

Defining Your Own Functions

Now that you know how defined functions can be used, you'll probably want to write your own functions. You can always do this by building a character matrix that looks like the definition of the function and applying the function ⎕*FX* to "fix" or define it [...you may want to sneak a look at this; it's on page 128]. You, of course, have all the power of APL to apply to the matrix when you want to make changes to your function. There is, however, an escape to a more specialized environment designed especially for editing of functions. In APL2 there are two similar system editors. These are called the "del" editors because the symbol "∇" (called "del") is used to request them.

If you don't say anything special (or if you enter ")*EDITOR* 1") you get the default del editor the next time you attempt to edit a function. This editor is designed for use at a line oriented terminal. If you enter ")*EDITOR* 2" then you get the extended editor the next time that you attempt to edit a function. This editor is designed for use at a screen-oriented terminal where more than one line can be processed at a time. Such editors are sometimes called "full screen editors." The extended editor allows you to make additions, deletions, and changes to a function directly often without the use of editing commands.

If you're not sure *which* editor you are set up to use, just type ")*EDITOR*"... with *no* editor number specified, it will return the number that's currently in effect.

The editors are designed primarily for the editing of APL functions and operators, although the extended editor also permits editing of character arrays. Since well-written APL functions tend to be small, only a small set of commands are needed. Also —in the spirit of APL— the commands are symbolic rather than English words. This eliminates any confusions with national languages when using the editors.

Using the Default Editor (*EDITOR* 1)

Throughout this discussion, the examples will form a *cumulative session*, just the same as you would see at your terminal if you entered all of these commands.

You enter the default editor for a new function by entering a del and a valid header (line zero) for the function. The system responds by prompting for line one of the function.

```
      ∇ Z←F X
[1]
```

You can then enter line 1 and the system will respond by prompting for line 2.

```
      ∇ Z←F X
[1]     ⍝ line 1
[2]
```

You can continue to enter lines as long as you want.

```
      ∇ Z←F X
[1]     ⍝ line 1
[2]     ⍝ line 2
[3]     ⍝ line 3
[4]
```

If you want to insert a line between two existing lines you must choose a number between the two line numbers and enter that number between brackets and follow the brackets with the line to be inserted. The system will respond by prompting for the next insert.

```
      ∇ Z←F X
[1]     ⍝ line 1
[2]     ⍝ line 2
[3]     ⍝ line 3
[4]     [2.1] ⍝ inserted line
[2.2]
```

You can insert as many lines as you want, subject to the system limitation of numbers with up to four digits to the right of the decimal point. If possible, the system will never prompt with the number of an existing line. Should the system ever prompt with the number of an existing line or should you manually enter such a line number then the new line you type will replace the existing line with that number. For example, to replace line 1 with some other line you could enter:

```
      ∇ Z←F X
[1]     ⍝ line 1
[2]     ⍝ line 2
[3]     ⍝ line 3
[4]     [2.1] ⍝ inserted line
[2.2] [1] ⍝ new line 1
[1.1]
```

In addition to entering line numbers in brackets you may also enter editing commands in the brackets. The available commands are:

Command Symbol	Meaning
⎕	Display
∆	Delete
→	Quit

The display and delete commands have five forms by which the lines acted upon are specified. They are:

[command]	apply to entire function
[command v]	apply to lines in vector v
[command f-l]	apply to lines numbered f through l
[command -l]	apply to lines numbered 0 through l
[command f-]	apply to lines numbered f through end

Given the function we defined above we can display the entire updated function by entering [⎕] after the line-number prompt:

```
      ∇ Z←F X
[1]     ⍝ line 1
[2]     ⍝ line 2
[3]     ⍝ line 3
[4]     [2.1] ⍝ inserted line
[2.2] [1] ⍝ new line 1
[1.1] [⎕] ←——————————— Here's our command
      ∇
[0]     Z←F X
```

```
[1]   ⍝ new line 1
[2]   ⍝ line 2
[2.1] ⍝ inserted line
[3]   ⍝ line 3
   ∇ 1984-02-16 11.08.18 (GMT-8)
[4]
```

When you are finished defining the function you can close it (that is, establish it as a function in your active workspace) by typing a del:

```
    ∇ Z←F X
[1]   ⍝ line 1
[2]   ⍝ line 2
[3]   ⍝ line 3
[4]   [2.1] ⍝ inserted line
[2.2] [1] ⍝ new line 1
[1.1] [⎕]
    ∇
[0]   Z←F X
[1]   ⍝ new line 1
[2]   ⍝ line 2
[2.1] ⍝ inserted line
[3]   ⍝ line 3
    ∇ 1984-02-16 11.08.18 (GMT-8)
[4]   ∇
```

The closing del may also be typed after any line entered in the function or after any editing command.

Once a function has been defined, you can edit it again by entering only a del and the function name.

If you are using a line-oriented terminal:

Alterations to existing lines of a function are achieved by means of the detailed editing request (also called super-edit). The command:

```
[n⎕m]
```

requests a display of line "n" with the cursor (or type element) positioned at column "m" of the display. Then, under the line to be altered you may then enter a "/" for any character to be deleted and a digit (0−9) under any character before which you wish to insert characters. After you press ENTER, the system will respond by re-displaying the line with the requested characters deleted and with blanks inserted where you want to add characters. You may then type into the blank spaces to make the additions. For example, to make an insert in line one of the function *F* defined above, you would enter:

```
      ∇F[1⎕9]
[1]   ⍝ new line 1
```

The cursor would be aligned under the "*w*". You then enter "/" to delete characters and numbers to add spaces for insertions:

```
      ∇F[1⎕9]
[1]   ⍝ new line 1
          /2    4
[1]   ⍝ ne   line     1
```

The cursor will be positioned under the first inserted blank, and you can type in the additions.

```
      ∇F[1⎕9]
[1]   ⍝ new  line 1
           /2     4
[1]   ⍝ next line for 1
[1.1]
```

When you have finished making all the modifications to the function you establish the changed function in the workspace by using the closing del, "∇".

Using the Full Screen Editor (*EDITOR* 2)

The full screen editor can be used to edit defined functions, defined operators, and character arrays. Throughout this discussion, the examples will each show what you would see at your terminal if you had entered all of these commands; each illustration represents the current screen-load of information.

Although the extended editor can be used at a line oriented terminal, the following discussion is restricted to screen-oriented terminals such as the IBM 3276, 3277, 3278, 3279, 3290, or 8775.

Here are a few principles that make using the extended editor easy:

- A line erased from the screen is not erased from the function if you also delete the line number
- The cursor never moves unless you move it.
- Whenever you press ENTER, only the lines you have actually changed are processed.

You enter the extended editor in much the same way as you entered the default editor:

```
      )EDITOR 2
      ∇Z←F X
```

The response, however, is different. The screen is cleared and replaced by the following display:

```
[⍝]∇ F.3  0000-00-00  00:00:00   ⍴: 1
[0] Z←F X
```

The first line is an information line that tells you the name of the object being edited (*F* in this case), the class of the object being edited (3 for function in this case — it could also be 4 for defined operator or 2 for a variable), the time and date of the last change to this function or operator (or zeros here, because *F* is a new function), and the number of lines in the function (1 in this case because we only defined the header). In the following discussion it's assumed that a function is being edited. Nothing would change if it were an operator or a variable, except that there's no timestamp shown if you're editing a variable (because APL2 doesn't keep track of the last time that a variable was updated, but it *does* keep track of it for a function or operator).

At this point the screen is said to have one segment — that is, anything that you type anywhere on the screen applies to the function *F*. A little later we'll see how to specify and use multiple screen segments.

You can then add lines to the function by entering them just below line zero as follows:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 1
[0] Z←F X
⍝ line 1
⍝ line 2
⍝ line 3
```

Note that you don't need to type in the line numbers (although you can if you want to). When you press ENTER, the system will respond like this:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 4
[0] Z←F X
[1] ⍝ line 1
[2] ⍝ line 2
[3] ⍝ line 3
```

And again, if you want to add more lines, you type them in just below the function:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 4
[0] Z←F X
[1] ⍝ line 1
[2] ⍝ line 2
[3] ⍝ line 3
Z←(4 5 6+2)×7 8
```

And, as before, pressing ENTER causes this response:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 5
[0] Z←F X
[1] ⍝ line 1
[2] ⍝ line 2
[3] ⍝ line 3
[4] Z←(4 5 6+2)×7 8
```

If you want to insert a line between two existing lines you just type over the second of them. For example, to add a line between lines 2 and 3, you just type over line 3 (...be sure to type over the brackets and line number so that APL knows that you're not just entering some new text for line 3):

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 5
[0] Z←F X
[1] ⍝ line 1
[2] ⍝ line 2
⍝ inserted line
[4] Z←(4 5 6+2)×7 8
```

Even though you typed over line 3, you'll find that line 3 is still part of the function, unchanged. This is because *no* line is *ever* changed unless you explicitly change or delete it. Erasing a line from the screen does not erase it from the function (as long as the brackets and line number are also erased). Therefore, upon pressing ENTER, the system will respond like this:

```
[⍝]∇ F.3  0000-00-00  00:00:00   ⍴: 6
[0   ]  Z←F X
[1   ]  ⍝ line 1
[2   ]  ⍝ line 2
[2.1]  ⍝ inserted line
[3   ]  ⍝ line 3
[4   ]  Z←(4 5 6+2)×7 8
```

In general, whenever you type in lines without line numbers, they are inserted into the function just after the last line back that does have a line number. In this example, we could have inserted more than one line by typing over line 4 as well, and continued with as many insertions as we wanted, all of which would have been placed between lines 2 and 3.

If you want to alter an existing line, you can just type over the old text of the line with the change. As long as the line number remains on the line, it will alter that line rather than insert a new one. For example, to add a comment to line 1:

```
[⍝]∇ F.3  0000-00-00  00:00:00   ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
```

Upon pressing ENTER, the display will not change in this case.

The extended editor includes the same editor commands as the default editor:

Command Symbol	**Meaning**
⎕	Display
∆	Delete
→	Quit

and the same forms for the commands:

[command]	apply to entire function
[command v]	apply to lines in vector v
[command f-l]	apply to lines numbered f through l
[command -l]	apply to lines numbered 0 through l
[command f-]	apply to lines numbered f through end

In addition, the extended editor has these commands:

[/ /]	search and display
[/ / /]	change

These commands have the same five forms as above, with the additional option of putting the characters *N* or ¨ (or both) after the last slash. "*N*" means that the search argument is an APL *name* and only names should be found. (They will be found even if they are inside quotes.) "¨" in a change command means that *every occurrence* of the search argument should be changed (within the specified line limits; for example, from lines 3-9). If no "¨" is entered, only the *first* occurrence in each line (or each specified line) will be changed.

In the extended editor, these commands may be entered on any line. In the case of the display or search commands, the display will begin on that line and continue on following lines until all are displayed or until the end of the screen segment.

The following are some examples of search and change commands:

`[/ABC/]`	Search the entire function for the *characters* "*ABC*".
`[/ABC/ N 5-]`	Search from line 5 to the end of the function for the *name* *ABC*. Note that if the name *ABC*1 appeared in the function it would not be found, but if *ABC* appears to be a name it will be found even if it is in a comment or a character constant.
`[/ABC/XYZ/ ¨ N]`	Change the name *ABC* to *XYZ* everywhere it appears in the function.

Two more commands provide a means to manipulate blocks of lines:

[∨ name]	put lines
[∧ name]	insert lines

These commands have the same five forms as the others that have been presented except they include a name.

The put command causes the selected lines to be saved in the active workspace as a character matrix with the given name. It's sort of like a save command [∇], except (in most cases) only part of the object is saved. (Notice that the symbol ∨ is only part of the symbol ∇, which makes it easy to remember.)

The insert command causes the selected lines from the named object to be inserted into the object at the spot where the command is entered. The symbol ∧ is often used to mean insert and is even on the insert key of some terminals. For example, all or part of a second function could be inserted into the one currently open. If no name is specified, then lines from the current function are inserted (that is, it becomes a *move* operation).

Here are some examples:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
[∨ QQ 2 4]
```

This causes line 2 and 4 to be put into a two-row matrix named *QQ*. Then:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[∧ QQ]
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
```

causes the contents of *QQ* to be inserted after line 0:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[0.1] ⍝ line 2
[0.2] Z←(4 5 6+2)×7 8
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
```

Since this function will be used in subsequent examples, let's delete those two lines that we just inserted:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[0.1] ⍝ line 2
[0.2] Z←(4 5 6+2)×7 8
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
[∆ .1 .2]
```

There are eight other commands that do not follow the above patterns. You have already seen the "∇" command, which closes a function if it is used at the end of a line. In the extended editor it is normally entered on a line by itself somewhere below the last line of the function. You have also seen the "→" command for exiting from the editor without fixing the function.

You can enter any of the eight commands, although normally the first six are requested by use of a PF key. The seven commands are:

Command	Definition	PF key
∇	Close definition	PF3
[↑]	Scroll up	PF7
[↓]	Scroll down	PF8
[⊤]	Scroll to cursor	PF9
[⍳]	Renumber	PF2
[∇]	Save	PF6
[⍝]	Do nothing	none
[→]	Quit	none

∇ *Close definition* ends the editing session for the current object that you're working on. It establishes the object in the active workspace and (if no other editing is being performed) returns you to immediate-execution mode.

[↑] *Scroll up* causes the first line in the segment in which the cursor is positioned to become the last line displayed in that segment of the screen. If this command is executed on the first screen of the definition, no action is taken.

[↓] *Scroll down* causes the last line in the segment in which the cursor is positioned to become the first line displayed in that segment of the screen.

[⊤] *Scroll to cursor* causes the line that the cursor is on to move to the top of the segment. Note that the editor *never* moves the cursor. It will always remain where *you* last placed it.

[⍳] *Renumber* causes the lines to be renumbered with integers starting with zero. In any case, when the function is closed the lines are renumbered.

[∇] *Save* causes the function to be established in the workspace but without terminating editing of the function. This may be helpful if you are creating several similar functions. You can define the function under one name and save it, then make changes to it (including a change to the name), and save it again as a second function. Realize that the function is *not permanently saved* on disk (as it would be with the ")*SAVE*" command — it's simply established in the workspace. If you want to ensure that it's really permanently saved, you'll still have to close the definition and)*SAVE* the workspace.

[⍝] *Do nothing* is the command that the editor puts at the top of a segment to identify the function being edited. You can enter it, too, to mark off portions of the function while you're editing... as soon as you leave the editor, these lines will disappear.

[→] *Quit* causes editing to be abandoned *without* establishing the function in the workspace.

* * *

One characteristic of a well-written APL application is the use of many small functions rather than a few large ones.[13] Therefore, it's common when defining a function to identify some computation that belongs in a subfunction. The extended editor allows you to leave the first function open for editing while opening a second function on the same screen.

A second screen segment is defined by using the "∇" to open a new function in the same way that you opened the first one. Wherever you type a new del, that is where the new screen segment will begin. For example, to open definition for function *G*:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1     ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8

∇Z←G X                        Here's our command
```

When you press ENTER, the system will respond like this:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1     ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8

[⍝]∇ G.3   0000-00-00   00:00:00    ⍴: 1
[0] Z←G X
```

You can now add, insert, alter, and delete lines in this function just as you did in the first one. You can make changes to both functions at once and they will both be processed the next time you press ENTER. You may open as many functions as you want and each time you enter an opening "∇" you cause another screen segment to be created. Of course you are limited by the size of the screen.

In addition to using a screen segment for a function definition you can define a segment for use to do immediate calculations. To execute an APL2 expression without leaving the editor, use the [⍎] command. The expression that follows this command is executed and its result is displayed in the segment.

For example, suppose we had not opened function *G*. We could open an execution segment as follows:

[13] It's hoped, of course, you won't break things down *so* far that folks will get lost in the forest of functions when they load the workspace. It's difficult to judge how big functions should be for the greatest ease of use, and where the breaks should occur. That sort of judgment simply comes from usage and personal preference. For some of *our* thoughts on the subject, though, you can refer to the discussion of "APL Building Blocks" on pages 221-226.

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[⍎]    2 3⍴⍳6
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
```

That will result in this:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[⍎   ] 2 3⍴⍳6
1 2 3
4 5 6
```

If you press ENTER again the executed expression and its results will disappear and the original function will re-appear. Remember — only the lines which you change are processed. If instead of just pressing ENTER, you touch (alter somehow) selected lines of the result that had been displayed, those lines will be processed as though you had entered them.

You can execute the function you are defining if you save [∇] it first:

```
[⍝]∇ F.3   0000-00-00   00:00:00    ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×7 8
[∇]

[⍎]    F 3
LENGTH ERROR
F[5] Z←(4 5 6+2)×7 8
      ^         ^
     F 3
     ^
```

Notice that the editor does not let the function be suspended if an error occurs. Also notice the silly error in the function. That's why you'll want to try out the function to see if it works. Since the editing session for *F* is still on the screen, it's a simple matter to move up and repair the offending line, save it again (with [∇]) and try it again.

Here it is after it has been corrected:

```
[⍝]∇ F.3   1984-02-16   13:12:02   ⍴: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×2
```

The system's response to this last change will be:

```
[⍝]∇ F.3   1984-02-16   13:13:48   ρ: 6
[0   ] Z←F X
[1   ] ⍝ line 1    ⍝ NEAT FUNCTION
[2   ] ⍝ line 2
[2.1] ⍝ inserted line
[3   ] ⍝ line 3
[4   ] Z←(4 5 6+2)×2

[⍎]   F 3
12 14 16
```

Now that the function works, we can *exit* from the editor in any of four ways:

- Enter a closing ∇ in each screen segment to establish the new definitions.
- Press PF3 once per segment to establish the new definitions.
- Enter [→] once per segment to abandon the editing *without* establishing the new definitions.
- Press PA2 twice to interrupt the editing — *none* of the new definitions will be established.

Note that if you are not currently in the editor, the line:

```
        ∇F[⎕]∇
```

will cause a display of the entire function in your normal session *without* entering full-screen mode.

More detail on both of the APL2 editors is available in *APL2 Programming: Language Reference.*

Using a System Editor (*EDITOR* name)

You may edit an APL defined function, defined operator, or character array using any of the system editors.

You enter the system editors in the same way you entered the "del" editors:

```
        )EDITOR XEDIT
        ∇Z←F X
```

In response, a sequential file will be written containing the definition of the object, and a named editor will be invoked on that file. When you close the editing session (how you do that is up to the editor you're using), the resulting file is read and the definition is reestablished in the workspace.

Chapter 4: The Quads

This chapter will discuss the topics of System Functions, System Variables, Shared Variables, and Event Handling. What do each of these topics have in common? All of them rely heavily upon the use of "quad"-names...which we'll define here.

System Functions and System Variables

Although the primitive functions of APL deal only with abstract objects (arrays of numbers and characters), it is often desirable to bring the power of the language to bear on the management of the environment of the system in which APL operates. This can be done within the language by identifying certain variables as elements of the interface between APL and its host system, and using these variables for communication between them. While they remain abstract objects to APL, the values of such *system variables* may have some required significance to the host system.

In principle, all necessary interaction between APL and its environment could be managed by use of a complete set of system variables, but there are situations in which it is more convenient, or otherwise more desirable, to use functions based on the use of system variables which may not themselves be made explicitly available. Such functions are called, by analogy, *system functions*.

System variables and system functions are denoted by *distinguished names* that begin with a quad ("⎕"). The use of such names is reserved for the system and cannot be applied to user-defined objects. These objects cannot be copied; those that denote system variables can appear in function headers, but only to be localized. These localization rules may at times differ from the rules that govern user-defined variables (taking into account their special requirements). Within APL statements, distinguished names are subject to all the normal rules of syntax.

System Functions

Like the primitive abstract functions of APL, the system functions are available throughout the system, and can be used in defined functions. They are monadic or dyadic, as appropriate, and have explicit results. In most cases they also have implicit results, in that their execution causes a change in the environment. The explicit result always indicates the status of the environment relevant to the possible implicit result.

Altogether, eighteen system functions are provided. Five of these are concerned with the management of the shared-variable facility and are described starting on page 149. Eleven system functions are described here.

List of System Functions

	Monadic usage	Dyadic usage		
Symbol			Function Name	Pages
⎕*AF*	×		Atomic Function	120
⎕*AT*		×	Attributes	121-125
⎕*CR*	×		Canonical Representation	125
⎕*DL*	×		Delay	127
⎕*EA*		×	Execute Alternate	172-176
⎕*EC*		×	Execute Controlled	176-177
⎕*ES*	×	×	Event Simulation	183-188
⎕*EX*	×		Expunge	127
⎕*FX*	×	×	Fix	128-130
⎕*NA*	×	×	Name Association	130-130
⎕*NC*	×		Name Classification	130
⎕*NL*	×	×	Name List	131
⎕*SVC*	×	×	Shared Variable Control	152
⎕*SVO*	×	×	Shared Variable Offer	150
⎕*SVQ*	×		Shared Variable Query	156
⎕*SVR*	×		Shared Variable Retraction	156
⎕*SVS*	×		Shared Variable State	155
⎕*TF*		×	Transfer Form	132-135

Note: *For formal definitions of all of the system functions, please refer also to APL2 Programming: Language Reference.*

⎕*AF*: Atomic Function

The atomic function maps integers to characters and characters to integers. The input may be either of those data types; the output will be the other. This function is therefore its own self-inverse.

It is the means by which four-byte characters may be accessed for such applications as Kanji. The atomic vector contains a copy of the first 256 characters from the list of the four *billion* characters that the atomic function can access.

The atomic function uses an index origin of zero, regardless of which origin setting the application has established. With origin set at zero, ⎕*AF* is analogous to "⎕*AV*⍳*R*" for character to numeric conversion and "⎕*AV*[*R*]" for numeric to character conversion.

The atomic vector is discussed on page 140.

⎕*AT*: Attributes

The attributes function, ⎕*AT*, returns information about each of several properties of APL objects. Its right argument is the object name (or a matrix of names), and its left argument shows which of the attributes you wish to interrogate.

Here are the three possible left arguments for ⎕*AT*, with a list of the results shown for each one:

Valence		
1 ⎕*AT* namelist	1	Explicit result (1=yes, 0=no)
	2	Function valence (number of arg uments)
	3	Operator valence (number of operands)
Fix Time (time and date of last update, in ⎕*TS* format)		
2 ⎕*AT* namelist	1	Year
	2	Month
	3	Day
	4	Hour
	5	Minute
	6	Second
	7	Millisecond
Execution Properties		
3 ⎕*AT* namelist	1	Nondisplayable
	2	Nonsuspendable
	3	Ignores weak interrupts (ATTENTION)
	4	Converts errors to *DOMAIN ERROR*s
Object Size (when written to a file)		
4 ⎕*AT* namelist	1	Bytes that the data and descrip tion require
	2	Bytes that the data-only requires

1. Valence

It may be desirable to determine under program control what the header of a defined function or operator looks like. Rather than having to analyze the character representation of it, you have the option of simply asking for the number of arguments and operands, and receiving an indication of whether or not the object is defined as having an explicit result.

```
      ∇ A PLUS B
[1]     A+B
      ∇

        1 ⎕AT 'PLUS'
0 2 0

        )ERASE PLUS
        1 ⎕AT 'PLUS'
0 0 0

        VAR←10 20 30
        1 ⎕AT 'VAR'
1 0 0
```

```
∇ Z←SORT M
[1]    Z←M[ALF⍋M;]
      ∇

         1 ⎕AT 'SORT'
1 1 0

      ∇ Z←L (F OP) R
[1]     Z←R F L
      ∇

         1 ⎕AT 'OP'
1 2 1
```

Explicit result (yes, 1)

The function valence (two arguments)

The operator valence (one operand)

```
           1 ⎕AT 4 5 ⍴ 'PLUS VAR SORT  OP    '
      0 0 0
      1 0 0
      1 1 0
      1 2 1
```

2. Fix Time

One piece of information that's often useful is the date that a function was last modified. This date is displayed automatically by the system editors when you display the function:

```
        ∇SORT[⎕]∇
      ∇ Z←SORT M
[1]     Z←M[ALF⍋M;]
      ∇  1984-05-01 15.30.37 (GMT-5)
```

Here's the time and date of last update (the "*GMT*-5" means five hours less than *GMT* time)

The attribute function can supply that same information *dynamically*, even if the function in question is locked.

```
      2 ⎕AT 'SORT'
1983 5 1 15 30 37 390
```

Human nature being what it is (and programming schedules being what *they* are), you probably just would *not* update these dates every time that you made some minor change to the function. But if you didn't change them, the whole point of the dates would be lost. Fortunately, the system does all of the footwork in maintaining these dates for you. Any time that the function is edited with one of the system editors or established with ⎕*FX*, the *fix-time* time stamp is updated.

Using the ⎕*AT* facility, those functions can then be examined under program control to find functions which have been updated since a particular date. For example, here's an expression which will list the names of functions and operators, sorted by the date of their last update (with most recently changed objects first):

```
LIST[ALF⍒2 ⎕AT LIST←⎕NL 3 4;]
```

("*ALF*" is a variable containing an alphabet that contains your desired collating sequence.) If you wanted to get a bit fancier, you could write a small function which would format the time stamps and display them along with the names of the functions, like this:

```
    ∇ Z←SHOW_DATES;F;I;T
[1]   ⍝DISPLAY FNS AND TIMESTAMPS; LATEST ON TOP
[2]    F←F[I←⍒T←2 ⎕AT F←⎕NL 3;]
[3]    Z←F,'  56:06 06/06/00'⍕100|T[I;4 5 2 3 1]
    ∇

      SHOW_DATES
SHOW_DATES  17:37 06/15/83
MIOTA       18:11 04/15/83
SQUAD       18:10 04/15/83
K           17:59 04/15/83
KWIC        14:26 04/14/83
DISPLAY      8:51 04/06/83
```

The time stamp returned by ⎕*AT* is affected by the setting of the time zone variable, ⎕*TZ*, described on page 148.

The fix-time is always all zeros for a variable — in other words, APL2 doesn't keep track of when each variable is changed. Similarly, any undefined name will return a time stamp of all zeros:

```
      VAR←10 20 30          |          )ERASE VAR
                            |
      2 ⎕AT 'VAR'           |          2 ⎕AT 'VAR'
0 0 0 0 0 0 0               |0 0 0 0 0 0 0
```

3. Execution Properties

The execution properties that are available with a left argument of 3 are the properties which are set either through use of dyadic ⎕*FX*, or by locking a function or operator with a . If *all four* of the execution properties are invoked, a function or operator is said to be "locked."

(For a discussion of *dyadic* ⎕*FX*, see page 129.)

```
      0 0 0 1 ⎕FX 'Z←L (F OP) R'  'Z←R F L'
OP

                                  Operator is now
      3 ⎕AT 'OP'                  nondisplayable
0 0 0 1

      ∇OP                         Operator is now
      3 ⎕AT 'OP'                  "locked"
1 1 1 1
```

The execution properties for a variable or an undefined name are always all zeros.

Any form of ⎕*AT* will, of course, accept a matrix of names as its right argument ...or even a nested vector of names, if you care to pair the "each" operator from page 41 with it:

```
      2 ⎕AT 2 5 ρ 'SORT OP
1984 5  1 15 30 37 390
1984 7 21  9 16  2  28

      ρ⎕←2 ⎕AT¨ 'SORT' 'OP'
 1984 5 1 15 30 37 390  1984 7 21 9 16 52 128
2

      A←1 2 3 ⎕AT¨ ⊂'OP'
      ρA
3

      ρ¨A
 3  7  4

      3 1 ρ A
1 2 1
1984 7 21 9 16 2 28
1 1 1 1
```

4. Object Size

If you are writing arrays to files, you often need to know how much external storage space the array will take. A file processor has to store both the data and a description of the data (so that a matrix can still look like a matrix when it's read back in from the file). Therefore, "4 ⎕*AT* name" gives you two numbers. The first number tells you how much space the data description and the data would take, and the second number tells you how much space the data would take by itself. "-/4 ⎕*AT* name" tells you how much space the data description takes (but you almost never want to know this).

⎕*CR*: Canonical Representation

The canonical representation of a defined function or operator is obtained as a result of applying the system function ⎕*CR* to the character vector (or scalar) representing the name of the function or operator. First, let's understand just what is meant by *canonical representation.* The *character representation* of a function is a character matrix satisfying certain constraints: the first row of the matrix represents the *function header* and must be one of the forms specified in the discussion of function headers on page 93. The remaining rows of the matrix, if any, constitute the *function body*, and may be composed of any sequence of characters. If the character representation satisfies additional constraints (such as no redundant spaces and left justification of the nonblank character in each row), it is said to be a *canonical representation.* The canonical representation of a function, then, is the minimum form of display which will completely define the function; it is devoid of such decorations as line numbers.

So, now that we understand what a canonical representation is, let's discuss a bit more about what ⎕CR does for us. Applying ⎕CR to the character vector (or scalar) representing the name of an already existing function will produce its canonical representation. For example, if we have a function called OVERTIME in the workspace, then:

```
      DEF←⎕CR 'OVERTIME'

      DEF
PAY←R OVERTIME H;TIME
TIME←0⌈H-40
PAY←R×1.5×TIME

      ⍴DEF
3 21
```

The use of ⎕CR does not change the status of the function OVERTIME. It still exists, and can be used for calculations. Thus:

```
      7 5 8 OVERTIME 35 40 45
0 0 60
```

If OVERTIME should be expunged:

```
      ⎕EX 'OVERTIME'
1
```

it is erased from the workspace, and is no longer available for use:

```
      7 5 8 OVERTIME 35 40 45
VALUE ERROR
      7 5 8 OVERTIME 35 40 45
            ^
```

Because the variable "DEF" is still in the workspace and is a canonical representation of "OVERTIME", we can reestablish the function using ⎕FX:

```
      ⎕FX DEF
OVERTIME
```

When ⎕CR is applied to an otherwise valid argument which does not represent the name of an unlocked defined function or operator, it returns a matrix of shape 0 0.

Possible error reports for ⎕CR are RANK ERROR if the argument is not a vector or a scalar, or DOMAIN ERROR if the argument is not a character array.

⎕*DL*: Delay

The delay function, ⎕*DL*, evokes a pause in the execution of the statement in which it appears. The argument of the function determines the duration of the pause, in seconds, but the accuracy is limited by possible competing demands on the system at the moment of release. Moreover, the delay can be aborted by a strong interrupt. The explicit result of the delay function is a scalar value equal to the actual delay. If the argument of ⎕*DL* is not a scalar with a numeric value, a *RANK* or *DOMAIN* error will be reported.

Generally speaking, the delay function uses only a negligible amount of computer time (as opposed to elapsed time). It can therefore be used freely in situations where repeated tests may be required at intervals to determine whether an expected event has taken place. This is useful in work with shared variables as well as in certain kinds of interactions between users and programs.

⎕*EX*: Expunge

The expunge function, ⎕*EX*, provides the facility to dynamically eliminate an existing use of a name. By "existing use," we mean the current, most local copy. Thus, ⎕*EX* '*PQR*' will erase the most local copy of the object *PQR* *unless* it is a label; labels may not be expunged. As one example of its use, certain name conflicts can be avoided by using this function.

The function returns an explicit result of 1 if the name is now unencumbered, and a result of 0 if it is not, or if the argument does not represent a well-formed name. A result of 1, therefore, signifies that the name is available for use, whereas a 0 signifies that it may not be used.

Expunging a shared variable will retract the sharing and erase the variable. If a function or operator is expunged while it is active or halted, the copy that is listed on the state indicator will persist until the execution has completed. When the stack entry is cleared the function or operator will cease to exist.

The expunge function applies to a matrix of names and then produces a logical vector result. ⎕*EX* will report a *RANK* error if its argument is of higher rank than a matrix, or a *DOMAIN* error if the argument is not a character array. A single name may also be presented as a vector or scalar, or the "each" operator may be employed, as in ⎕*EX*¨ '*ABC*' '*DEF*' to erase the two names, "*ABC*" and "*DEF*" (refer to the description of the "each" operator on page 41).

You may wish to compare the operation of ⎕*EX* with that of the ")*ERASE*" system command, discussed on page 82. One major difference (other than form) is that ⎕*EX* discards *the most local* reference to a name that is currently active (which *may* be a global object), and)*ERASE* discards *only global* references.

⎕*FX*: Fix

The definition of a function or operator can be established, or "fixed," by applying the system function ⎕*FX* to its character representation. The function ⎕*FX* produces as an explicit result the character vector which represents the name of the function being fixed, while replacing any existing definition of a function with the same name. A halted function or operator may be replaced using ⎕*FX*, but this will *not* replace the active copy on the stack.

The fix function has both a monadic and a dyadic form. Let's look at the monadic case first.

An expression of the form ⎕*FX M* will establish a function (or operator) if *both* of the following conditions are met:

1. *M* is a valid representation of a function. It may be a matrix (with each row representing a function line), or it may be a nested vector of vectors (with each element representing a function line). Any matrix which differs from the canonical representation only in the addition of nonsignificant spaces is a valid representation.

2. The name of the function to be established does not conflict with an existing use of the name for a variable or label.

The first row of the matrix (or first item of the nested vector) represents the *function header* and must be one of the forms specified in the discussion of function headers on page 93. The remaining rows (or items), if any, constitute the *function body*, and may be composed of any sequence of valid APL characters (refer to the table on page 11).

If the expression fails to establish a function then no change occurs in the workspace and the expression returns a scalar index of the row in the matrix argument or the element in the nested vector argument where the first fault was found. If multiple errors are present, only the first will be reported. (This value is origin dependent.) If the argument of ⎕*FX* is not a matrix or a nested vector, a *RANK ERROR* will be reported, and if it is not a character array, a *DOMAIN ERROR* will result.

For example,

```
      M← 2 8 ρ 'Z←ROOT NZ←N*.5   '

      M
Z←ROOT N
Z←N*.5

      ⎕FX M
ROOT ◄────────────  The name of the function is returned
                    if ⎕FX completed successfully

      ∇ROOT[⎕]∇ )
    ∇ Z←ROOT N  }   newly-created function
[1]   Z←N*0.5   )
    ∇
```

```
       ⎕FX 'Z←N ROOT A' 'Z←A*÷N'
ROOT

       ∇ROOT[⎕]∇
     ∇ Z←N ROOT A      function has been replaced
[1]    Z←A*÷N
     ∇


       ⎕FX 1 2 3
DOMAIN ERROR ←—— argument doesn't match
       ⎕FX 1 2 3  conformability rules
       ∧

       ⎕FX 'Z←A*÷N' 'Z←N ROOT A'
1 ←———————————————————
                     The first item isn't valid as a header
                     line, so the function can't be created.
```

Now let's look at the dyadic form of ⎕*FX*. A left argument may optionally be supplied to control the *execution properties* of the newly-created function or operator. If present, the argument must be composed of four boolean elements. If there is *no* left argument, the action will be the same as `0 0 0 0 ⎕FX M`. A `1` specifies the inclusion of one of the properties; a `0` declines use of the property.

These four properties are:

1. Prevent the display of the function or operator (use of the built-in editors will return *DEFN ERROR*, and ⎕*CR* will return a `0` by `0` matrix).

2. Prevent suspensions, as is done with primitive functions.

3. Ignore weak interrupts (attention) during its execution.

4. Convert error messages to *DOMAIN ERROR*, to further hide its internal workings. [Resource and environment errors, such as *WS FULL*, will continue to be reported.]

If *all four* of these properties are specified, the resultant function or operator is said to be "locked." The setting of these properties (among other things) may be read through the use of the attributes function, ⎕*AT*, described on pages 121-125.

If the left argument to ⎕*FX* is a single `1` or `0`, the normal rules of scalar extension will logically replicate that value to apply to all four properties. Therefore, "`1 1 1 1 ⎕FX` name" and "`1 ⎕FX` name" will each create a locked function.

⎕*NA*: Name Association

Normally when you enter a name, its value is determined within the workspace. You can use ⎕*NA* to declare that the value of the name is determined outside the workspace.

The left argument is a two-item vector giving the desired name class and the number of the name-resolving processor.

```
      3 11 ⎕NA 'DAN'
1
```

This declares that '*DAN*' is a function whose value is provided by associated processor 11. Thus using ⎕*NA*, programs written in other languages can be executed as though they were locked functions. For more information, see *APL2 Programming: System Services Reference*. The processors supplied with APL2 are:

11	FORTRAN, Assembler Language
10	REXX

Here's an example of some functions available with ⎕*NA*:

```
      A←1 1 DAN 'NOW IS THE TIME'
      A
NOW IS THE TIME
      ⍴A
4
```

⎕*NC*: Name Classification

The monadic function ⎕*NC* accepts a matrix of characters and returns a numeric indication of the class of the name represented by each row of the argument. Alternatively, the "each" operator may be employed, as in ⎕*NC*¨ '*ABC*' '*DEF*' to return the classification of the two names, "*ABC*" and "*DEF*" (refer to the description of the "each" operator on page 41). A single name may also be presented as a vector or scalar, as in ⎕*NC* '*FN*'.

The result of ⎕*NL* is a suitable argument for ⎕*NC*, but other character arrays may also be used, in which case the possible results are integers ranging from ¯1 to 4:

Result	Meaning
¯1	Invalid name
0	Unused (but valid) name
1	Label
2	Variable
3	Function
4	Operator

The significance of 1, 2, 3, and 4 are as for ⎕*NL*; a result of 0 signifies that the corresponding name is available for any use; a result of ¯1 signifies that the argument is not available for use as a name. The latter case may arise because the name is not a valid name at all (as in ⎕*NC* '*A.B*').

Once a name has been accepted for sharing via ⎕*SVO*, its classification becomes 2 (variable), even if no value has yet been assigned.

For discussion of a related function, see ⎕*NL* (name list), following.

⎕*NL*: Name List

The dyadic function ⎕*NL* yields a character matrix, each row of which represents the name of an object in the dynamic environment. The rows of the result are sorted into ⎕*AV* order.

The right argument is an integer scalar or vector which determines the classes of the names desired, with the following permissible values:

Right Argument	Result
1	Labels
2	Variables
3	Functions
4	Operators

The left argument is a scalar or vector of alphabetic characters which restricts the names produced to those with an initial letter occurring in the argument. For example, '*ABC*' ⎕*NL* 3 4 returns the names of all functions or operators whose names begin with either *A*, *B*, or *C*.

The monadic function ⎕*NL* behaves analogously with no restriction on initial letters. For example, ⎕*NL* 2 produces a matrix of all variable names, and either ⎕*NL* 3 4 or ⎕*NL* 4 3 produces a matrix of all function and operator names.

The uses of ⎕*NL* include the following:

- In conjunction with ⎕*EX*, all the objects of a certain class can be dynamically erased; or a function can be readily defined that will clear a workspace of all but a preselected set of objects.

- In conjunction with ⎕*CR*, functions can be written to automatically display the definitions of all or certain functions in the workspace, or to analyze the interactions among functions and variables.

- The dyadic form of ⎕*NL* can be used as a convenient guide in the choice of names while designing or experimenting with a workspace.

For discussion of a related function, see ⎕*NC* (name classification) on page 130.

⎕*TF*: Transfer Form

Before we discuss how the transfer form function works, perhaps we should discuss what "transfer form" is.... Okay:

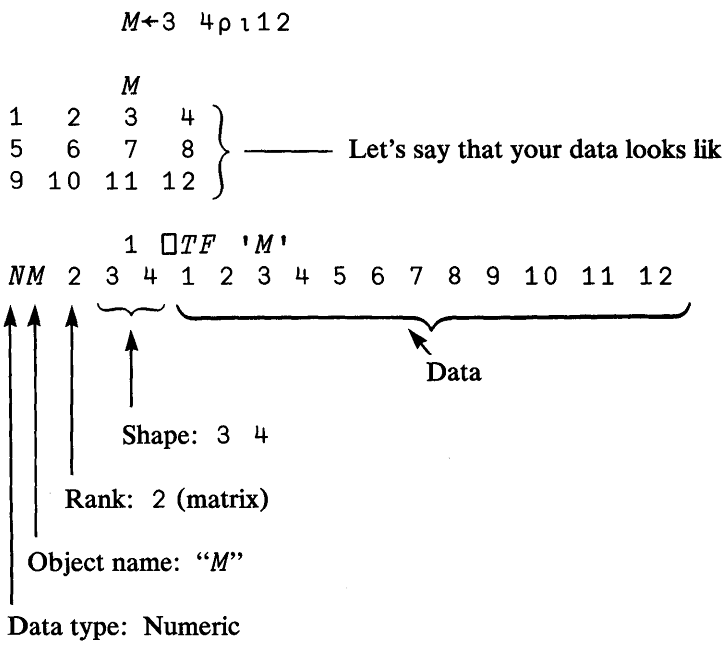
What's a Transfer Form?

Transferring workspaces between different versions of APL has always been a difficult procedure. Transferring workspaces between systems is fine, but if the two systems are *not* running the same version of APL, all bets are off. The problem is that the internal format of the workspace differs wildly between different implementations. Sure, the workspaces look pretty much the same to *you*, as a user, but to the system, foreign workspaces look... — well, pretty foreign.

Several years ago, a Standards organization met to discuss this problem. What resulted was a recommended convention for formatting APL objects (such as functions and variables) so that they could be moved between systems. To accomplish this, each system would need to have a means of transferring objects from APL workspaces to a transfer file. (The file would be in an agreed-upon format.) The file would then be moved between systems, and the contents of the file would be moved back into a new workspace, where the objects would be reestablished as APL objects.

At the heart of this procedure is the need for a function that can convert the APL objects from the system-dependent format to the standard interchange format and back again. That function can be (and, in the past, has been) a defined function. It also can be (and is now) a primitive function. ...Introducing ⎕*TF* (applause).

There are several forms to discuss, but as a starting point, let's say that the transfer form function takes as its right argument the *name* of the object that you wish to convert into the agreed-upon format. This standard interchange format may be produced using "`1 ⎕TF` objectname". That format looks like this:



```
        M←3 4ρι12

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

        1 ⎕TF 'M'
NM 2 3 4 1 2 3 4 5 6 7 8 9 10 11 12
```

This information is sufficient to recreate that numeric matrix on another system, regardless of what version of APL exists there. A similar format, of course, exists for other data types, and for functions.

When the interchange format was put together, nested array systems and user-defined operators hadn't yet been developed. The interchange format, therefore, didn't address these items at all. So, APL2 has an alternate format that is suitable for transferring any APL2 objects onto and off of a transfer file. This alternate format is available by using a left argument of 2:

```
      (1 1⍉M)←'NESTED' 'ARRAY' 'EXAMPLE'

      M
NESTED       2         3  4
     5  ARRAY          7  8
     9      10 EXAMPLE 12

      ρM
3 4

      2 ⎕TF 'M'
M←3 4ρ'NESTED' 2 3 4 5 'ARRAY' 7 8 9 10 'EXAMPLE' 12
```

"Gee, that looks *really familiar!* Isn't that displayed in just the same way that you'd enter it?" Well, actually, now that you mention it, it does look "coincidently" quite similar. And, yes, it *happens* to be directly executable on APL2.

Let's try a defined function this time:

```
        ∇Z←A PLUS B
[1]     Z←A+B∇ ◄──────── Good, not too complicated

        1 ⎕TF 'PLUS'
FPLUS 2 2 10 Z←A PLUS BZ←A+B

        A←2 ⎕TF 'PLUS'
        A
⎕FX 'Z←A PLUS B' 'Z←A+B'

        ⍴A
24
```

Rebuilding Transfer Forms Back Into APL Objects

Now that you've got all of the APL objects into transfer form, how do you convert them back? ...What's the *inverse* of ⎕*TF*?

Well, it's ⎕*TF*. That's right, ⎕*TF* is its own self-inverse. If the right argument to ⎕*TF* is a valid APL name, it performs the APL-to-transfer-form conversion that we've just seen. If, on the other hand, the right argument is *not* a valid APL name, ⎕*TF* attempts to perform a transfer form-to-APL conversion, and, in true inverse fashion, returns the *name* of the newly-created APL object. If no object could be created (perhaps due to an error in the data), ⎕*TF* returns an empty character vector.

```
        )ERASE PLUS

        PLUS
VALUE ERROR
        PLUS
        ^

        A
⎕FX 'Z←A PLUS B' 'Z←A+B'

        2 ⎕TF A
PLUS

        2 PLUS 2
4
```

Fine. Now we can convert the APL objects both to and from the transfer form. But what do we do from here? How does the transfer form get out to a file, or back again?

Moving APL Objects To and From Transfer Files

There are two system commands built-in to APL2 for moving these converted objects in and out of the workspace. They are ")*IN*" (to get the data into the workspace) and ")*OUT*" (to get the data from the workspace to the file). In fact, using)*IN* and)*OUT*, you don't even have to get involved with the nitty-gritty detail of running each of your functions and variables through ⎕*TF*. The system commands do all of that for you. But ⎕*TF* can always be employed for creating special files for special conversions. And with a little imagination, the "2 ⎕*TF*" form really opens up some possibilities for creative constructions. For an example of this, we'll be getting to an editing example a little later on (on pages 200-201).

Those two system commands are discussed at length in the system command section. Refer to:

-)*IN* (Page 80)
-)*OUT* (Page 81)

See also the pictorial diagram called "The Effects of Selected System Commands," on page 69.

And for more in-depth coverage of this subject, refer to *APL2 Migration Guide*.

System Variables

System variables are special instances of *shared variables*, which are discussed in depth on pages 149-157. The characteristics of shared variables that are most significant here are these:

- If a variable is shared between two processors, the value of the variable when used by one of them may well be different from what that processor last specified, and

- Each processor is free to use or not use a value specified by the other, according to its own internal workings.

System variables are shared between a workspace and the APL processor. Such sharing takes place automatically each time a workspace is activated and, when a system variable is localized in a function, each time the function is used.

The table on the next page lists the system variables and gives their significance and use. Three classes can be discerned:

1. With most of the system variables, the value specified by the user (or available in a clear workspace) is used by the APL processor during the execution of operations to which they relate. If this value is inappropriate, or if no value has been specified after localization, a ⎕-- *ERROR* (for instance, a "⎕*IO ERROR*") will be evoked at the time of execution. These variables are reset by the system to a default value following a)*CLEAR* operation.

Examples:	⎕*CT*	—	Comparison Tolerance
	⎕*EM*	—	Event Message
	⎕*ET*	—	Event Type
	⎕*FC*	—	Format Control characters
	⎕*IO*	—	Index Origin
	⎕*L*	—	Left Argument
	⎕*LX*	—	Latent Expression
	⎕*PP*	—	Printing Precision
	⎕*R*	—	Right Argument
	⎕*RL*	—	Random Link
	⎕*SVE*	—	Shared Variable Event

2. Some of the variables are treated the same as those in the first case, except that global values set by the user will persist across)*LOAD* and)*CLEAR* operations. These are referred to as *session variables*, since a single setting can persist for the duration of the terminal session.

Examples:	⎕*HT*	—	Horizontal Tabs
	⎕*NLT*	—	National Language Translation
	⎕*TZ*	—	Time Zone
	⎕*PW*	—	Printing Width

3. With the remainder of the system variables, localization or setting by the user are immaterial. The APL processor will always reset the variable before it can be used again.

Examples:	⎕*AI*	—	Accounting Information
	⎕*AV*	—	Atomic Vector
	⎕*LC*	—	Line Counter
	⎕*TC*	—	Terminal Control characters
	⎕*TS*	—	Time Stamp
	⎕*TT*	—	Terminal Type
	⎕*UL*	—	User Load
	⎕*WA*	—	Workspace Available

Environment within a Clear Workspace

These values are present at sign-on time (unless reset through the automatic loading of a CONTINUE workspace), and will then be carried forward following a)LOAD or)CLEAR:

⎕HT	Horizontal Tabs	ι0
⎕NLT	National Language Translation[1]	' ' (The initial language depends upon the country where the system is located)
⎕PW	Printing Width[1]	(Depends upon the type of terminal being used)
⎕TZ	Time Zone[1]	(Depends upon the location of the system)
)PBS	Printable Backspace	_ (that is, ON)
)EDITOR	System Editor	1

These values are present following)CLEAR; the settings of the entries in the upper chart will be carried over from the previous workspace:

⎕L	Left Argument	No value
⎕R	Right Argument	No value
⎕CT	Comparison Tolerance[1]	1E¯13
⎕EM	Event Message	3 0⍴' '
⎕ET	Event Type	0 0
⎕FC	Format Control	.,*0_¯
⎕IO	Index Origin	1
⎕LC	Line Counter	ι0
⎕LX	Latent Expression	''
⎕PP	Printing Precision[1]	10
⎕PR	Prompt Replacement	' '
⎕RL	Random Link[1]	7*5 ↔ 16807
⎕SVE	Shared Variable Event	0
⎕WA	Workspace Available[1]	(Depends upon the local installation, and in some systems, upon options selected by the user)
)WSID	Workspace name	None (CLEAR WS)
	Workspace password	None
)SI	State indicator	Cleared

[1] These items have values which may vary from system to system.

For entries where values are shown, those values were chosen as being widely-used values.

(For your convenience, this is a repeated copy of the same table that appeared in the section "System Commands" on page 65.)

Comparison of System Variable Characteristics
Sorted by Class

Global value persists over a)*CLEAR* or a)*LOAD*	Cannot be effectively localized	Ignores an assignment	Set by the system upon an error	**Symbol**	**Name**	**Pages**
•	•	•	•	⎕*CT*	Comparison Tolerance	*
•	•	•	•	⎕*FC*	Format Control Characters	216-217
•	•	•	•	⎕*IO*	Index Origin	19-20
•	•	•	•	⎕*LX*	Latent Expression	142
•	•	•	•	⎕*PP*	Printing Precision	*
•	•	•	•	⎕*PR*	Prompt Replacement	195-201
•	•	•	•	⎕*RL*	Random Link	*
•	⊗	•	•	⍞	Character Input/Output	195-201
•	⊗	•	•	⎕	Evaluated Input/Output	193, 8
•	⊗	•	•	⎕*SVE*	Shared Variable Event	156
•	⊗	•	⊗	⎕*L*	Left Argument	165
•	⊗	•	⊗	⎕*R*	Right Argument	165
•	⊗	⊗	⊗	⎕*EM*	Event Message	179-182
•	⊗	⊗	⊗	⎕*ET*	Event Type	177-179
•	⊗	⊗	•	⎕*AI*	Accounting Information	*
•	⊗	⊗	•	⎕*AV*	Atomic Vector	140
•	⊗	⊗	•	⎕*LC*	Function Line Counter	164
•	⊗	⊗	•	⎕*TC*	Terminal Control Characters	146
•	⊗	⊗	•	⎕*TS*	Time Stamp	147
•	⊗	⊗	•	⎕*TT*	Terminal Type	147
•	⊗	⊗	•	⎕*UL*	User Load	*
•	⊗	⊗	•	⎕*WA*	Workspace Available	*
⊗	•	•	•	⎕*HT*	Horizontal Tabs	141
⊗	•	•	•	⎕*NLT*	National Language Translation	143-145
⊗	•	•	•	⎕*PW*	Printing Width	*
⊗	•	•	•	⎕*TZ*	Time Zone	148

Not all of the system variables are discussed in this manual.

* For a description of these system variables, please refer to *APL2 Programming: Language Reference.*

Comparison of System Variable Characteristics
Sorted Alphabetically by Symbol

Global value persists over a *)CLEAR* or a *)LOAD*	Cannot be effectively localized	Ignores an assignment	Set by the system upon an error	Symbol	Name	Pages
•	⊛	•	•	⍞	Character Input/Output	195-201
•	⊛	•	•	⎕	Evaluated Input/Output	193, 8
•	⊛	⊛	•	⎕*AI*	Accounting Information	*
•	⊛	⊛	•	⎕*AV*	Atomic Vector	140
•	•	•	•	⎕*CT*	Comparison Tolerance	*
•	⊛	⊛	⊛	⎕*EM*	Event Message	179-182
•	⊛	⊛	⊛	⎕*ET*	Event Type	177-179
•	•	•	•	⎕*FC*	Format Control Characters	216-217
⊛	•	•	•	⎕*HT*	Horizontal Tabs	141
•	•	•	•	⎕*IO*	Index Origin	19-20
•	⊛	•	⊛	⎕*L*	Left Argument	165
•	⊛	⊛	•	⎕*LC*	Function Line Counter	164
•	•	•	•	⎕*LX*	Latent Expression	142
⊛	•	•	•	⎕*NLT*	National Language Translation	143-145
•	•	•	•	⎕*PP*	Printing Precision	*
•	•	•	•	⎕*PR*	Prompt Replacement	195-201
⊛	•	•	•	⎕*PW*	Printing Width	*
•	⊛	•	⊛	⎕*R*	Right Argument	165
•	•	•	•	⎕*RL*	Random Link	*
•	⊛	•	•	⎕*SVE*	Shared Variable Event	156
•	⊛	⊛	•	⎕*TC*	Terminal Control Characters	146
•	⊛	⊛	•	⎕*TS*	Time Stamp	147
•	⊛	⊛	•	⎕*TT*	Terminal Type	147
⊛	•	•	•	⎕*TZ*	Time Zone	148
•	⊛	⊛	•	⎕*UL*	User Load	*
•	⊛	⊛	•	⎕*WA*	Workspace Available	*

Not all of the system variables are discussed in this manual.

* For a description of these system variables, please refer to *APL2 Programming: Language Reference.*

⎕AV: Atomic Vector

The atomic vector, ⎕AV, is a 256-element character vector, containing all possible characters. Certain elements of ⎕AV may be terminal control characters, such as carriage return or linefeed, but other elements of ⎕AV may neither print nor exercise control. The indices of any known characters can be determined by an expression such as ⎕AV⍳'ABCA̲B̲C̲'.

The sequence of the atomic vector currently matches IBM's EBCDIC convention; however, the ordering of ⎕AV is always implementation dependent. Its order may occasionally be altered to provide different capabilities, and it frequently differs between different versions of APL. indexing selected characters out of ⎕AV in defined functions may lead to improper operation if the order of ⎕AV changes in the future. If you need to access certain characters from ⎕AV by means of indexing, you may want to consider the use of a variable containing the special characters that your application needs. You may also find that the use of ⎕TC could alleviate some of these problems.

The following chart represents the order of the characters in "16 16⍴⎕AV". Their positions are shown in both decimal (base 10) and hexadecimal (base 16). To find the position of any character in the chart, simply add its row and column positions together, and add the index origin in which you are working. For example, the "$" character is located at decimal position 80+11+⎕IO, or position 92 in origin 1.

```
          0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
Decimal   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5   Decimal
  +------                           ------+
  | Hex   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   Hex |
  |       0 1 2 3 4 5 6 7 8 9 A B C D E F       |
  |     +---------------------------------+
 00  00 |                                 | 00  00
 16  10 |                                 | 10  16
 32  20 |                                 | 20  32
 48  30 |                                 | 30  48
 64  40 |   A̲ B̲ C̲ D̲ E̲ F̲ G̲ H̲ I̲ ¢ . < ( + | | 40  64
 80  50 | & J̲ K̲ L̲ M̲ N̲ O̲ P̲ Q̲ R̲ ! $ * ) ; ¬ | 50  80
 96  60 | - / S̲ T̲ U̲ V̲ W̲ X̲ Y̲ Z̲ ¦ , % _ > ? | 60  96
112  70 |   ∧ ¨ ⌼ ⍸ ⍷     ∨ ` : # @ ' = " | 70 112
128  80 | ~ a b c d e f g h i ↑ ↓ ≤ ⌈ ⌊ → | 80 128
144  90 | ⎕ j k l m n o p q r ⊃ ⊂   ○   ← | 90 144
160  A0 | ¯ ~ s t u v w x y z ∩ ∪ ⊥ [ ≥ ∘ | A0 160
176  B0 | ⍺ ∊ ⍳ ⍴ ⍵   × \ ÷   ∇ ∆ ⊤ ] ≠ ∣ | B0 176
192  C0 | { A B C D E F G H I ⍲ ⍱ ⌷ ⌽ ⍂ ⍉ | C0 192
208  D0 | } J K L M N O P Q R ⌶ ! ⍒ ⍋ ⍞ ⍝ | D0 208
224  E0 | \ ≡ S T U V W X Y Z ⌿ ⍀ ∵ ⊖ ⌹ ⍕ | E0 224
240  F0 | 0 1 2 3 4 5 6 7 8 9   ⍫ ⍙ ⍟ ⍎   | F0 240
        +---------------------------------+
  |       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0       |
  | Hex   0 1 2 3 4 5 6 7 8 9 A B C D E F   Hex |
  +------                           ------+
Decimal   0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1   Decimal
          0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

Not shown in the chart, of course, are the terminal-control characters, such as backspace and new line.

Refer also to "⎕*AF*" (the atomic function) on page 120.

⎕*FC*: Format Control characters

Format control characters, available from ⎕*FC*, are used to control the formatting of data, primarily through the use of the format function (⍕).

For details of the use of ⎕*FC*, refer to the description of picture format on pages 209-217; a table showing the usage of ⎕*FC* appears on pages 216 and 217.

⎕*HT*: Horizontal tabs

The system variable ⎕*HT* is not currently supported on this version of APL2. ⎕*HT* is designed to tell APL where you have set the tabs on your terminal, so that APL can use automatic horizontal tabbing to speed up the printing of all of your output. Its *name* is supported on APL2 to provide compatibility with other versions of APL, so that existing functions will not get a *VALUE ERROR* on the name.

⎕*LX*: Latent Expression

The APL statement represented by the latent expression is automatically executed whenever the workspace is activated with a ")*LOAD*" command. Formally, ⎕*LX* is used as an argument to the execute function (⍎⎕*LX*), and any error message will be appropriate to the use of that function.

Here are some common forms of the latent expression:

To invoke an arbitrary function *F*:

```
    ⎕LX←'F'
```

To print a message upon activation of the workspace:

```
    ⎕LX←'''FOR NEW FEATURES IN THIS WS ENTER: NEW'''
```

To automatically restart a suspended function:

```
    ⎕LX←'→⍳0'
```

The variable ⎕*LX* may also be localized within a function and respecified therein to furnish a *different* latent expression when the function is suspended. For example:

```
        ⎕LX←'START'

        ∇START;⎕LX
[1]     ⎕LX←'→0ρ⎕←''WE CONTINUE FROM WHERE WE LEFT OFF'''
[2]     'WE NOW BEGIN LESSON 2'
[3]     DRILL_FUNCTION
[4]     ∇

          )SAVE ABC
      7/18/1984 12.21.39 (GMT-5)
```

On the first activation of workspace *ABC*, the function *START* would be automatically invoked; if it were later saved with *START* halted, subsequent activation of the workspace would automatically continue execution from the point of interruption.

⎕NLT: National Language Translation

The national language translation variable, ⎕NLT, handles translation of error messages and system commands into any of several natural languages.

⎕NLT is set by assigning to it the name of a national language, entered in that language. For example, to select German, set ⎕NLT←'DEUTSCH'.

The languages that are currently supported are:

Language	⎕NLT←
Danish	'DANSK'
English	''
Finnish	'SUOMI'
French	'FRANCAIS'
German	'DEUTSCH'
Hebrew	'HEBREW'
Italian	'ITALIANO'
Katakana	'KATAKANA'
Norwegian	'NORSK'
Portuguese	'PORTUGUES'
Spanish	'ESPANOL'
Swedish	'SVENSKA'

The initial default value of ⎕NLT (at sign-on time) is dependent upon the location of the system (so that a system in France can default to French, and so forth). In English-speaking countries, this initial value is an empty character vector (⎕NLT←''), causing the default language to be English. In other countries, the initial value may be one of the other entries from the preceding table. In any event, a null entry will always select English.

If ⎕NLT is set to anything invalid, the system will reset it to an empty vector, causing translations to be in English. For example, ⎕NLT←'ENGLISH' is invalid, so it causes the system to default to English (got that?)... but then, the same could be said of ⎕NLT←'PIG LATIN'.

Once a national language has been selected through the use of ⎕*NLT*, error reports will be displayed in that language, and system commands may be entered *either* in that national language *or* in English. For example:

```
      ⎕NLT←''                 |      ⎕NLT←'FRANCAIS'
                              |
      1 2 3+4 5               |      1 2 3+4 5
LENGTH ERROR                  |ERREUR DE DIMENSION
      1 2 3+4 5               |      1 2 3+4 5
      ∧    ∧                  |      ∧    ∧
                              |
      ⎕NLT←'DEUTSCH'          |      ⎕NLT←'NORSK'
                              |
      1 2 3+4 5               |      1 2 3+4 5
LAENGENFEHLER                 |LENGDE-KONFLIKT
      1 2 3+4 5               |      1 2 3+4 5
      ∧    ∧                  |      ∧    ∧
```

This setting also affects the display of the error messages in ⎕*EM*:

```
      ⎕NLT←'FRANCAIS'
      XYZ
VALEUR NON DEFINIE
      XYZ
      ∧

      ⎕EM
VALEUR NON DEFINIE
      XYZ
      ∧
      ρ⎕EM
3 18
```

```
      ⎕NLT←'DEUTSCH'
      ⎕EM
VALEUR NON DEFINIE
      XYZ
      ∧
      ρ⎕EM
3 18
```

The value of ⎕*EM* is based upon the setting of ⎕*NLT* at the time that the error occurred.

The setting of ⎕*NLT* also allows you to enter system commands in other than English. For example,

```
      ⎕NLT←''
      )LOAD DATA
SAVED    7/14/1984  12.11.19 (GMT-5)

      ⎕NLT←'DEUTSCH'

      )LADE DATA ◄────────── ")LOAD", in German.
GESPEICHERT    7/14/1984  12.11.19 (WEZ-5)
      ⎕NLT
DEUTSCH

      ⎕NLT←'SUOMI'

      )LATAA DATA ◄──────── ")LOAD", in Finnish.
TALLETETTU    7/14/1984  12.11.19 (GMT-5)
      ⎕NLT
SUOMI

      )TYHJENNYS ◄────────── ")CLEAR", in Finnish.
TYHJ# TY@TILA
```

By the way, the "#" and "@" symbols are national-use symbols, so they will have different graphics in Finland than they have in the United States.

Notice, however, that you can *always* enter the system commands in English:

```
      ⎕NLT
SUOMI
      )LOAD DATA
TALLETETTU    7/14/1984  12.11.19 (GMT-5)
```

Notice also that ⎕*NLT* is a *session variable*; thus, as shown in the examples above, its value will persist over a)*LOAD* or)*CLEAR*.

For a complete list of the system commands and the error messages in each of the supported national languages, refer to the appendix of *APL2 Programming: Language Reference*.

⎕*TC*: Terminal Control Characters

The Terminal Control variable, ⎕*TC*, is a character vector containing characters which control screen or page positioning on a terminal. These characters are:

Element	Character
1	Backspace
2	New line (cursor or carriage-return)
3	Line feed (feed up one line without returning to the left)

```
      '∇',⎕TC[1],'_'
∇̲
```

(realize that *not all* terminals are able to display lines with backspaces in them)

```
      'NEW',⎕TC[2],'LINE',⎕TC[2],'CHARACTERS'
NEW
LINE
CHARACTERS


      'LINE',⎕TC[3],'FEED',⎕TC[3],'CHARACTERS'
LINE
    FEED
        CHARACTERS
```

While ⎕*TC* currently contains three characters, it is recommended that you don't consider its length to be fixed. Future extensions could add additional elements.

⎕*TS*: Time Stamp

The time stamp variable, ⎕*TS*, is a seven-element numeric integer vector, containing the current system time and date, in this format:

Element	Meaning	Range
1	Year (Four digits)	(year)
2	Month	1-12
3	Day of the month	1-31
4	Hour (on a 24-hour clock)	0-23
5	Minute	0-59
6	Second	0-59
7	Millisecond	0-999

For example,

```
      ⎕TS
1984 6 21 14 15 31 127

      ⎕TS
1984 6 21 14 15 33 229

      ρ⎕TS
7
```

The hours position will roll over to 0 at midnight; therefore, one second past midnight will display like this:

```
      ⎕TS
1984 6 22 0 0 1 0
```

The default setting for the time stamp will usually indicate either GMT or the local time for the city where that system is located — either of which may not match your own local time. To reset it for your own local time, see ⎕*TZ* (time zone) on page 148.

⎕*TS* is respecified by the system every time it's referenced, so assigning a value to it or localizing it has no effect. ...You can *try* it, of course...

```
      ⎕TS←1946 5 16 10 30 0 0
      ⎕TS
1984 6 21 14 16 2 124
```

...but do you *feel* any younger??

⎕*TT*: Terminal Type

The Terminal Type variable, ⎕*TT*, is a numeric scalar showing the type of terminal that you are using for your APL session.

Here are its possible values:

Value	Terminal Type
0	Indeterminate
1	Correspondence
2	PTTC/BCD
3	(Not currently used)
4	3270 with APL feature
5	3270 without APL feature

⎕*TT* is respecified by the system every time it's referenced, so assigning a value to it or localizing it has no effect.

⎕*TZ*: Time Zone

The time zone variable, ⎕*TZ*, is a numeric scalar showing your local time displacement from Greenwich Mean Time (GMT). The initial setting (at sign-on time) is chosen by the system which you are using. In general, its initial value will usually indicate either GMT or the local time for the city where that system is located — either of which may not match your own local time.

To set it to indicate *your* time zone, specify the number of hours that must be added to Greenwich Mean Time to match your local time. For example, U.S. Eastern Standard Time would be set by ⎕*TZ*←¯5 (five hours *less* than GMT), and British Summer Time would be set by ⎕*TZ*←1 (as in "Spring forward, Fall back").

The setting of ⎕*TZ* controls the display of ⎕*TS* (time stamp), described on page 147, and workspace "*SAVED*"- dates:

```
          ⎕TZ
0
          )LOAD MYWORK
SAVED     7/19/84   4.13.27 (GMT)

          ⎕TZ←¯5
          )LOAD MYWORK
SAVED     7/18/84   23.13.27 (GMT-5)
```

⎕*TZ* is a session variable, so its value is carried over following a)*LOAD* or)*CLEAR* [...after all, you don't change time zones when you load a workspace.]

Shared Variables

Two otherwise independent concurrently operating processors can communicate, and thereby be made to cooperate, if they share one or more variables. Such *shared variables* constitute an interface between the processors, through which information may be passed to be used by each for its own purposes. In particular, variables may be shared between two active APL workspaces, or between an APL workspace and some other processor that is part of the overall APL system, to achieve a variety of effects including the control and utilization of devices such as printers, card readers, magnetic tape units, and magnetic disk storage units.

In use in an APL workspace, a shared variable may be either global or local, and is syntactically indistinguishable from ordinary variables. It may appear to the left of an assignment, in which case its value is said to be *set*, or elsewhere in a statement, where its value is said to be *used*. Either form of reference is an *access*.

At any instant a shared variable has only one value, that last assigned to it by one of its owners. Characteristically, however, a processor using a shared variable will find its value different from what it might have set earlier.

A given processor can simultaneously share variables with any number of other processors. However, each sharing is bilateral; that is, each shared variable has only two owners. This restriction does not represent a loss of generality in the systems that can be constructed, and commonly useful arrangements are easily designed. For example, a shared file can be made directly accessible to a single control processor which communicates bilaterally with (or is integral with) the file processor itself. In turn, the central processor shares variables bilaterally with each of the using processors, controlling their individual access to the data, as required.

It was noted on page 135 that system variables are instances of shared variables in which the sharing is automatic. It was not pointed out, however, that access sequence disciplines are also imposed on certain of these variables, although one effect of this was noted; namely, variables like the time stamp accept any value specified, but continue to provide the proper information when used. The discipline that accomplishes this effect is an inhibition against two successive accesses to the variable unless the sharing processor (the system) has set it in the interim.

When ordinary "undistinguished" variables are to be shared, explicit actions are necessary to effect the sharing and establish a desired access discipline. Six system functions are provided for these purposes: three for the actual management and three to provide related information. Hang on; we'll cover those in just a bit.

Distinguished Names for Controlling Shared Variables

Symbol	System variable	System function with Monadic usage	System function with Dyadic usage	Function or Variable Name	Page
⎕SVC		×	×	Shared Variable Control	152
⎕SVE	×			Shared Variable Event	156
⎕SVO		×	×	Shared Variable Offer	150
⎕SVQ		×		Shared Variable Query	156
⎕SVR		×		Shared Variable Retraction	156
⎕SVS		×		Shared Variable State	155

For formal definitions of all of the system functions and variables, refer also to *APL2 Programming: Language Reference.*

⎕SVO: Shared Variable Offer

A single *offer* to share is of the form P ⎕SVO N, where P is the identification of another processor and N is a character vector representing a pair of names. The first of this pair is the name of the variable to be shared, and the second is a surrogate name which is offered to match a name offered by the other processor. The name of the variable may be its own surrogate, in which case only the one name need be used, rather than two. For example, the three sets of actions shown below all have the same effect, which is to share one variable between two processors 1234 and 5678, the variable being known to the former as ABC, and to the latter as Q.

```
User 1234:                          User 5678:

       5678 ⎕SVO 'ABC Y'
1
                                           1234 ⎕SVO 'Q Y'
                                    2

       5678 ⎕SVO 'ABC Q'
1                                          1234 ⎕SVO 'Q'
                                    2

       5678 ⎕SVO 'ABC'                     1234 ⎕SVO 'Q ABC'
1                                   2
```

The surrogate names have no effect other than to control the matching, making it possible for one processor to operate with no direct knowledge of, or concern with, the variable name used by the other. The same surrogate can be used in a succession of offers to the same processor, in which case they are matched in sequence by appropriate counter-offers. The same surrogate may also be used for offers to any number of other processors at the same time. However, since a variable may be offered to (or shared with) only one other processor at a time, each coincident use of a particular surrogate name must be associated with a different variable name.

The explicit result of the expression *P* ⎕*SVO* *N* is the *degree of coupling* of the name or name pair in *N*: *zero* if no offer has been made, *one* if an offer has been made but not matched, *two* if sharing is completed. An offer to any processor (other than the offering processor itself) increases the coupling of the name offered if the name has zero coupling and is not the name of a label, function, or operator. An offer never decreases the coupling.

The *monadic* function ⎕*SVO* does not affect the coupling of the name represented by its argument, but does report the degree of coupling as its explicit result. If the degree of coupling is one or two, a repeated offer has no further implicit result, and either monadic or dyadic ⎕*SVO* may be used for inquiry. Advantage is taken of this in the following example of a defined function for establishing sharing with processor identification *PID* using a shared variable named in *NAME*:

```
      ∇ COUPLING←PID OFFER NAME
[1]     ⎕SVE←10              ⍝ START THE COUNTDOWN
[2]    ⍝                     ⍝ MAKE THE OFFER:
[3]    LOOP:→(2=COUPLING←PID ⎕SVO NAME)/OKAY
[4]     →(0≠⎕SVE)/LOOP       ⍝ WAIT FOR ACCEPTANCE
[5]     'NO DEAL'
[6]     →0
[7]    OKAY:'ACCEPTED'
      ∇
```

If the arguments of ⎕*SVO* fail to meet any of the basic requirements, the appropriate error report is evoked and the function is not executed. If a user attempts to share more variables than the quota allotted to him by the system administrator, the error report will be *SYSTEM LIMIT*. (If for any reason the shared variable facility itself is not available, the same report could be given.) An offer to a processor will be tendered, whether or not the processor happens to be available.

The value of a shared variable when sharing is first completed is determined thus: if both owners had assigned values beforehand, the value is that assigned by the first to have offered; if only one owner had assigned a value, that value prevails; if neither had assigned a value, the variable has no value. Names used in sharing are subject to the usual rules of localization.

A set of offers can be made by using a vector left argument (or a scalar or one-element vector which is automatically extended) and a matrix right argument, each of whose rows represents a name or name pair. The offers are then treated in sequence and the explicit result is the vector of the resulting degrees of coupling. If the quota of shared variables is exhausted in the course of such a multiple offer, as many of the offers as possible will be tendered. Monadic ⎕*SVO* can tell you which share offers were successfully tendered.

An offer made with zero as left argument is a *general offer*, that is, an offer to *any* processor. A general offer will be matched only with a counteroffer which is not general, that is, one that explicitly identifies the processor making the general offer. The processor identification associated with a user is the user's account number. Auxiliary processors are usually identified by integer numbers from 1 through 999.

For discussion of a related function, see ⎕*SVR* (shared variable retraction) on page 156.

⎕*SVC*: Shared Variable Control

Access control of shared variables is handled by ⎕*SVC*. Consider the following simple example of sharing the variable *V* between two users 1234 and 5678:

```
User 1234:                          │ User 5678:
                                    │
      5678 ⎕SVO 'V'                 │
1                                   │
                                    │       1234 ⎕SVO 'V'
                                    │ 2
                                    │
      V←5                           │
                                    │
                                    │       V←3×V*2
                                    │
      V                             │       V
75                                  │ 75
```

The relative sequence of events in the two workspaces, after sharing, is significant; for example, had that last access of *V* by 1234 in the foregoing example preceded the setting by 5678, the resulting value would have been 5 rather than 75.

In most practical applications it is important to know that a new value has been set between successive uses of a shared variable, or that use has been made of an assigned value before a new one is set. Since, as a practical matter, this cannot be left to chance, an *access control* mechanism is embodied in the shared variable facility.

The access control operates by inhibiting the setting or use of a shared variable by one owner or the other, depending upon the *access state* of the variable, and the value of an *access control matrix* which is set jointly by the two owners, using the dyadic form of the system function ⎕*SVC*. If, in the example above, one user (say 5678, for example) had followed his offer to share *V* by the expression `1 1 1 1 ⎕SVC 'V'`, then the desired sequence would have been enforced. That is, the use of *V* by 5678 would be automatically delayed until *V* is set by 1234, and the use by 1234 would be delayed until *V* is set by 5678.

The delay occasioned by the inhibition of any access uses only a negligible amount of computer time. Interruption by a strong interrupt signal during the period of delay aborts the access and unlocks the keyboard.

Let's briefly discuss an *access control matrix*, which is a two-by-two element binary matrix which controls the actions of shared variables. The meanings of the positions of this access control matrix for two users "A" and "B" are:

Set by A	Set by B
Use by A	Use by B

Notice that the first row of ACM is associated with setting of the variable by each owner, and the second with its use.

- If ACM[1;1]=1, then two successive sets by A require an intervening access (set or use) by B.
- If ACM[1;2]=1, then two successive sets by B require an intervening access by A.
- If ACM[2;1]=1, then two successive uses by A require an intervening set by B.
- If ACM[2;2]=1, then two successive uses by B require an intervening set by A.

The value of the access control matrix is available to the user through the monadic function ⎕SVC. A related access state representation is available to a user through the ⎕SVS function (which will be described on page 155). For a shared variable V the result of the expression ⎕SVC 'V' executed by user A is the *access control vector*, ",ACM" (the four-element ravel of ACM). However, if user B executed the same expression he would obtain the result ",⌽ACM". The reason for the reversal is that sharing is symmetric: neither owner has precedence over the other, and each sees a control vector in which the first one of each pair of control settings applies to his own accesses.

The setting of the access control matrix for a shared variable is determined in a manner which maintains the functional symmetry. An expression of the form L ⎕SVC 'V' executed by user A assigns the value of the logical left argument L to a four-element vector which, for the purposes of the present discussion, will be called QA. Similar action by user B sets QB. The value of the access control matrix is determined as follows:

```
      ACM←(2 2⍴QA)∨⌽2 2⍴QB
```

Since *ones* in ACM inhibit the corresponding actions, it is clear from this expression that one user can only increase the degree of control imposed by the other (although he can, by using ⎕SVC with a left argument of zeros, restore the control to that minimum level at any time).

Access control can be imposed only after a variable is offered, either before or after the degree of coupling reaches two. The initial values of QA and QB when sharing is first offered are zero.

Here are some settings of the access control vector which are of common practical interest:

Access Control Vector as seen by:		
User A:	User B:	Comments
0 0 0 0	0 0 0 0	No constraints.
0 0 1 1	0 0 1 1	Half-duplex. Ensures that each use is preceded by a set by the partner.
1 1 0 0	1 1 0 0	Half-duplex. Ensures that each set is preceded by an access by partner.
0 1 1 0	1 0 0 1	Simplex. Controlled communications from B to A (for card reader, etc.).
1 1 1 1	1 1 1 1	Reversing half-duplex. Maximum constraint.

A group of *N* access control matrices can be set at once by applying the function ⎕*SVC* to an *N* by 4 matrix left argument and an *N*-rowed matrix right argument of names. The explicit result is an *N* by 4 matrix giving the current values of the (ravels of) control matrices. When control is being set for a single variable the left argument may be a single 1 or 0 if all inhibits or none are intended.

⎕*SVS*: Shared Variable State

The state of any shared variable which you are currently using may be interrogated using the monadic system function ⎕*SVS*. The right argument to ⎕*SVS* is a the name of a variable (or a matrix of names, with one name per row). The result is a vector (or matrix) showing the state of that variable (or variables).

Result	Meaning
0 0 0 0	Not a shared variable
0 0 1 1	Set by one processor, and referenced by the other (the initial state)
1 0 1 0	Set by A, but not yet referenced by B
0 1 0 1	Set by B, but not yet referenced by A
Note:	You are processor A; the other processor (the one that you named with dyadic ⎕*SVO*) is processor B.

⎕*SVR*: Shared Variable Retraction

Sharing offers can be retracted by the monadic function ⎕*SVR* applied to a name or a matrix of names. The explicit result is the degree (or degrees) of coupling *prior* to the retraction. The implicit result is to reduce the degree of coupling to zero.

Retraction of sharing is automatic if the connection to the computer is interrupted or if the user signs off, loads a new workspace, erases (or expunges) the variable, or replaces the variable by copying an object of the same name into the workspace. Sharing of a variable is also retracted by its erasure (either through)*ERASE* or through ⎕*EX*), or if it is a local variable, upon completion of the function in which it appeared.

The nature of the shared variable implementation is often such that the current value of a variable set by a partner will not be represented within a user's workspace until actually required to be there. This requirement prevails when the variable is to be used, when sharing is terminated, or when a)*SAVE* command is issued (since the current value of the variable must be stored). Under any of these conditions it is possible for a *WS FULL* error to be reported. In all cases, the prior access state remains in effect and the operation can be retried after corrective action.

For discussion of a related function, see ⎕*SVO* (shared variable offer) on page 150.

⎕*SVQ*: Shared Variable Query

There are three monadic inquiry functions which produce information concerning the shared variable environment but do not alter it; the functions ⎕*SVO* and ⎕*SVC* already discussed on pages 150 and 152, and the function ⎕*SVQ*. A user who applies the latter function to an empty vector obtains a vector result containing the identification of each user making any sharing offer to him. A user who applies the function ⎕*SVQ* to a nonempty argument obtains a matrix of the names offered to him by the processor identified in the argument. This matrix includes only those names which have *not* been accepted by counteroffers.

To produce a character matrix whose rows represent the names of all shared variables in the dynamic environment, use either one of these two expressions:

```
          (0≠⎕SVO M)/[1] M←⎕NL 2
    or    (0≠⎕SVO M)⌿M←⎕NL 2
```

⎕*SVE*: Shared Variable Event

The system *variable* ⎕*SVE* (shared variable event) provides a facility for delaying until a shared variable that you are using changes state, so that you don't have to continually check to see if the state has changed. ⎕*SVE* also provides a means for stating that you only wish to wait for a maximum of "n" seconds, so that your APL function doesn't hang interminably if the change never occurs.

After ⎕*SVE* has been set with some nonnegative value, as in ⎕*SVE*←5, (indicating five seconds of delay), referencing it will cause a delay of either the number of seconds specified by its value or the time necessary until one of the shared variables that you are using changes state (whichever occurs first).

The value returned by ⎕*SVE* represents the approximate time remaining in the specified delay. This will usually be 0, because it will normally delay until the time has expired. Even if it's interrupted prematurely, when you reference it again (for instance, to check the delay), it just resumes the delay. The only time that the result will be other than 0 is when a shared variable event has occurred during the delay (or, of course, when you explicitly *set* it to 0). And, after all, knowing when a shared variable changes state is what this is all about. Realize that a 0 response may be *indeterminate*, in that a shared variable event *could* have occurred *at the same moment* that the time in ⎕*SVE* expired.

Since ⎕*SVE* is designed to be sensitive to the state of any shared variable under your control, it is not affected by localization. Its value following)*CLEAR* or)*LOAD* is 0.

Its action is somewhat similar to the action of the ⎕*DL* (Delay) function, *except* that it will terminate early if a shared variable changes state during the delay time, *and* the value returned indicates the time remaining rather than the time accrued. As with ⎕*DL*, this facility uses almost no compute time for delaying, and may be used freely to delay execution. Also, as with ⎕*DL*, the delay may be prematurely interrupted by means of a strong interrupt from the keyboard. For a discussion of ⎕*DL*, see page 127.

Event Handling

While we always try to "do things right the first time" (don't we?), there comes that inevitable time when we make a mistake. APL2 has planned ahead for this eventuality and provides several new facilities to help with problem determination and correction.

> *"I claim not to have controlled events,*
> *but confess plainly that events have*
> *controlled me."*
>
> —Abraham Lincoln,
> in a letter to A.G. Hodges,
> 4 April 1864

Sure, every APL programmer has been in this situation. Clearly, we need to let our functions *control* events. But before we learn how to handle those events, perhaps we should ask, "What *is* an event?" Fair enough. Webster defines an "event" as:

1. A happening or occurrence, esp. when important
2. A result; consequence; outcome
3. A particular contest or item in a program (the pole vault, high jump, and other "events")

An "event," then, is simply something that you wish to acknowledge. Some events, of course, are more noteworthy than others. People tend not to observe the day after a birthday with quite the same zeal as the birthday itself. And in the APL environment, the same situations hold. *Any* action in APL can be considered to be an event; assignments, branches, calling functions are all events. ...They just may not be *noteworthy* events. So what makes an event noteworthy? ...The desire to observe it.

There are many places in programming where foreseeing some situation may not be possible, or even if it's possible, may not be practical to measure. Take, for instance, the most common example of checking for an error. Your programs, of course, should do "extensive error checking" if they are to be used in a production environment; anyone will tell you that.

McGurk's Law:

> *"Any improbable event which would create maximum confusion if it* ***did*** *occur,* ***will*** *occur."*
>
> —H.S. Kindler,
> from "Organizing the Technical Conference,"
> Reinhold Publishing Company, 1960

Used with the permission of The Dick Sutphen Studio.

But some things just aren't practical to check. There are an infinite number of things that *could* happen during the execution of your programs — all things which are outside normal operations. What happens if there's an unexpected *DOMAIN ERROR*? What happens if an error occurs that I haven't thought of? What happens if the user presses ATTENTION and interrupts the execution part way through? ...Welcome to Event Handling. Here's where we discuss that particular "item in a program" that Mr. Webster spoke of, and find out how to get it into our own programs.

APL2 provides the means for letting execution simply run its course, and *if* an abnormal situation occurs, for helping you to determine what occurred and why, and then taking corrective action. APL2 also allows you to *simulate* the error conditions that APL2 itself reports, so that you can, for instance, report errors in different degrees of detail for different audiences.

You may have noticed by now that the previous page spoke of *events* as being a broad subject, but that we are now referring mostly to *errors*. Errors are indeed only one type of event that we might want to handle, but errors also happen to be the one type of event that the most people have had the greatest interest in handling. Because of this, most of the event handling in APL2 is aimed toward the handling of errors. So errors aren't the only things that are considered to be "events" (controlling the use of the ATTENTION key was one example of another kind of event) but errors *will* be the subject of most of our following discussions.

Before we get into the in-depth discussions of events, let's take a look at a list of the facilities that are available for handling these events.

Facilities Available for Event Handling

Distinguished Names for Event Handling

Symbol	System variable	System function with Monadic usage	System function with Dyadic usage	Function or Variable Name	Pages
⎕*EA*			×	Execute Alternate	172-176
⎕*EC*		×		Execute Controlled	176-177
⎕*EM*	×			Event Message	179-182
⎕*ES*		×	×	Event Simulation	183-188
⎕*ET*	×			Event Type	177-179
⎕*L*	×			Left Argument	165
⎕*R*	×			Right Argument	165
⎕*SVE*	×			Shared Variable Event	156

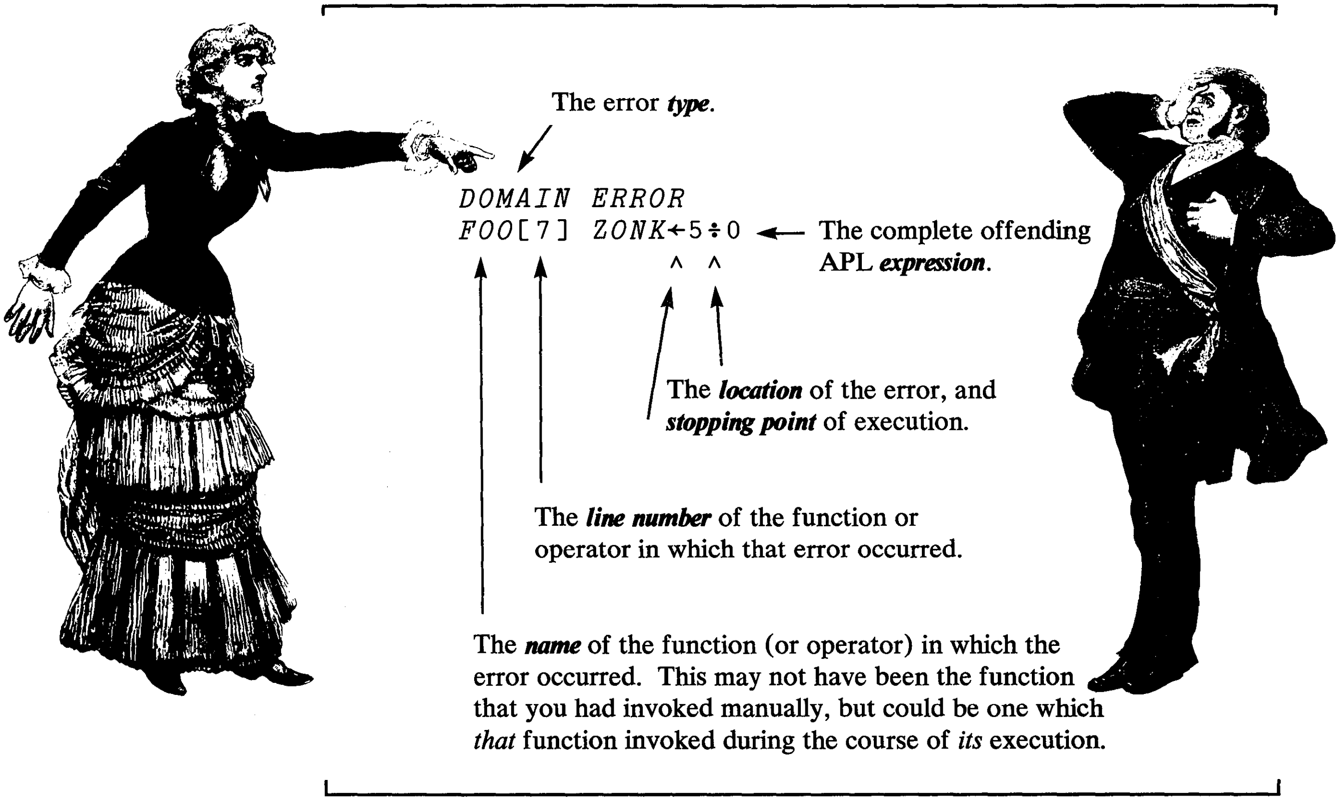
Additional Facilities for Event Handling

Facility	Purpose	Pages
^ ^	Multiple carets displayed at errors	29, 161, 165
→⎕*LC*	Restart (at beginning of halted line)	164
→⍳0	Resume (inside halted statement)	161-166
→	Abort	161-166
)*RESET*	Resets the State Indicator	163
)*SI*	Display State Indicator	161-166 83
)*SINL*	Display State Indicator with Name List	83
)*SIS*	Display State Indicator with Statements	161-166, 83

Note: For formal definitions of all of these Event Handling facilities, please refer to APL2 Programming: Language Reference.

Display of Errors

Entry of a statement which cannot be executed will invoke an *error report* which indicates the nature of the error and displays carets, indicating both where the error occurred and where the execution halted:



The error ***type***.

```
DOMAIN ERROR
FOO[7] ZONK←5÷0
           ^ ^
```

The complete offending APL ***expression***.

The ***location*** of the error, and ***stopping point*** of execution.

The ***line number*** of the function or operator in which that error occurred.

The ***name*** of the function (or operator) in which the error occurred. This may not have been the function that you had invoked manually, but could be one which *that* function invoked during the course of *its* execution.

Used with the permission of The Dick Sutphen Studio.

There will typically be two carets under the line of code. The left caret shows you how far APL got in its right-to-left scan of the line. The right caret shows you the point of the actual error — normally, that will indicate which function APL was evaluating when the error occurred. In this example, division by zero can't be performed, because the zero divisor is outside the "domain" (or defined range) of the division function.

It's possible, of course, that the last thing evaluated was the same point that had the error. In that situation, there will be only one caret (or you may think of it as being as though the two carets are there, but superimposed upon one another).

Any time that an error is reported by APL, you will be given this type of information. Learning to understand these error messages will be a great assistance in writing code quickly and easily. The error messages are part of the interactive process that APL handles so well. Let APL help you to design applications, by learning to interpret and use the error messages.

Whether you use the results of these reports to modify the code or not, *any time that you receive an error message from* APL, *some action is being called for on your part.* An error does not cancel the execution of a statement; it "suspends" the execution. This means that the code that you have entered is kept in a "things to do" list, so that APL can get back to it later and resume execution *upon request from you.* If you take *no* action, this list may accumulate entries, and can create some confusing side-effects. (Refer to "A Mystification to Avoid," on page 83.) Good practice therefore dictates that you *take some action whenever an error is reported.* The choices for action are:

- Correct the error and resume execution where you stopped.
- Cancel the execution of that statement; let APL know that you won't be coming back to it.

Using The State Indicator

This "suspended statements" list that we mentioned is a stack of statements which have not completed execution either because they contain an error, or because they call a function that was stopped for some reason. It is a list showing the "state" of the workspace at any given moment. It is therefore called the "*state indicator.*" You can display this list by typing ")*SI*" (state indicator):

```
      )SI
FOO[7]
*
*
```

The list is displayed in the order of the most recent statements first (at the top of the list). The stars ("*") mark the entries that were entered manually from the keyboard.

You can also optionally display the actual statements that were being executed when the errors occurred. This is done using the ")*SIS*" (state indicator with statements) command:

```
      )SIS
FOO[7]  ZONK←5÷0
            ∧ ∧
* FOO
  ∧
* 3+(Y×X)
     ∧∧
```

The "*FOO*" entry is the name of the most recent function in which we had an error. The stars in the display mean that this statement was a user input (as opposed to a line from a function). Entries will accumulate both from running functions and from direct keyboard entries of statements ("immediate-execution mode").

Clearing Out an Error

There are two ways to discard the entries from the state indicator. You may either selectively cancel one suspension at a time, or you may simply cancel all of the suspensions.

Abort

You can cancel the execution of a statement by entering a right-arrow (→) and pressing ENTER (or CARRIAGE-RETURN, or EXECUTE, or whatever your particular terminal calls it). Doing so will remove the top entry or entries from the state indicator, up to the line with a star on it. This takes you back to the point of the last keyboard input:

```
      →
      )SI
*
```

We had two entries originally; there is now just one left. If we don't want to resume execution of that statement, it can be cleared out by entering another right arrow:

```
      →
      )SI
```

Since there are no more errors pending completion, the state indicator is now "clear."

)RESET

Let's assume for a moment that you had a *lot* of suspensions. If you wish to clear *all* of the entries from the state indicator, you may do so in one swell foop, by entering ")RESET":

```
      )SI
FOO[7]
*
THISNAMEISLONGERTHANIWOULDHAVEPREFERRED[3]
*
ANOTHER_NAME[6]
*
REPORT[9]
*
*
*
      )RESET
      )SI
```

By entering)RESET n, you could also have cleared just the "n" most recent entries from the state indicator, instead of having to clear all of them. Some additional detail regarding this command is available on page 84.

Several Methods for Fixing an Error

Restart

In the case of a defined function or operator, you can edit the function and branch to the fixed line to start that line over:

```
      PROCESS JULY
SYNTAX ERROR
PROCESS[6]  ←3 4⍴DATA ←──── Here's our error
            ∧

      ∇PROCESS[6] M←3 4⍴DATA∇ ←─── This will
                                  repair the
                                  statement.

      ⎕LC ←──── The line counter indicates the line numbers
6               where the halt occurred in a defined function


      →⎕LC ←─── This restarts the last line
1 2 3           (and is equivalent to "→6")
4 5 6           And here's our final result.
```

Resume

All well and good, but suppose that "*DATA*" in the above example was a *shared variable* — or a *long*-running subfunction. We *don't* want to have to restart that line from the beginning! What we'd like to do is to correct the problem and pick up from right where we stopped... in the middle of the line. No problem. That can be done by following the same steps shown above, but substituting "→⍳0" for the "→⎕*LC*". "→⎕*LC*" says to restart the current line from the beginning, and "→⍳0" says to resume the line from the point where it halted.

```
    ∇ FOO
[1]   C←0
[2]   A+C←C+1
    ∇

      FOO
VALUE ERROR
FOO[2]  A+C←C+1
        ∧∧

      A←0

      →⎕LC
2
```

```
    ∇ FOO
[1]   C←0
[2]   A+C←C+1
    ∇

      FOO
VALUE ERROR
FOO[2]  A+C←C+1
        ∧∧

      A←0

      →⍳0
1
```

⎕L and ⎕R: Left and Right Arguments

If the offending line was from immediate execution mode rather than from a defined function, we don't have the option of editing the line and picking up from where we left off. But what we *can* do is to respecify the arguments of the failing function, and then resume (assuming that the error is one where that is appropriate).

Here is a statement with an obvious error. (They won't always be so obvious):

```
      1+(A←2 3)×4 5 6+2
```

When you attempt to execute it, you are immediately given an indication that something went wrong. Here's what would display at your terminal:

```
LENGTH ERROR
      1+(A←2 3)×4 5 6+2
       ∧        ∧
```

The first line tells you what error was detected. The second line gives you the statement in which the error occurred. In this case you just entered the line in immediate-execution mode, but in other cases it could be a line from a defined function. The third line contains two carets to tell you where in the execution of the line the error was detected.

The left caret says how much of the statement has been looked at. In particular, everything to the right of the left caret has been scanned and whatever possible has been evaluated. So, for example, the assignment of the value 2 3 to the name *A* has been done.

The right caret identifies the function in which the error was detected. In this case it was the "×" function.

Since the arguments to "×" were calculated, it is not immediately apparent what they were. Therefore, the system variables ⎕*L* and ⎕*R* are set to the values of the arguments of the failing function. Thus, we can display the values of the left and right arguments that caused the failure and see immediately why a *LENGTH ERROR* was produced.

```
      ⎕L
2 3
      ⎕R
6 7 8
```

These are the left and right arguments of the failing function (intermediate results)

We can *respecify* the left argument of the "×" function, and tell APL2 to pick up right where it left off... in the middle of the line:

```
      ⎕L←1 2 3
      →⍳0
7 15 25
```

Resume at the point where the error occurred

Keep in mind that this does not change the statement; it only changes this execution of it. In particular, if *another error* occurs, the original statement will display. Also notice that specifying new values for ⎕*L* and ⎕*R* replaces the evaluated left and right arguments, but affects no names. In our case, *A* will be unchanged, even though ⎕*L* was specified.

Some Other Ways to Fix Errors

Specifying new values for ⎕*L* and ⎕*R* won't always get things going again. ⎕*L* and ⎕*R* provide a means for viewing and setting the arguments for a function that has failed during execution. If the error that had been reported was a *VALUE ERROR*, *no function* would be currently under execution, so ⎕*L* and ⎕*R* wouldn't be involved at all. To resume after a *VALUE ERROR*, you can assign a value directly to the name being reported, and resume:

```
      X←5
      (Y×X)+3
VALUE ERROR
      (Y×X)+3
       ^^

      Y←10
      →⍳0
53
```

In similar fashion, since a *SYNTAX ERROR* indicates a problem with the statement as opposed to a problem with the data, *SYNTAX ERROR*s can never be fixed by resetting ⎕*L* and ⎕*R*. You'll find scattered other errors which will refuse the assistance of ⎕*L* and ⎕*R*. A *WS FULL* problem, to cite another example, is best handled by freeing up some additional space. ⎕*L* and ⎕*R* are used only for those errors that are related to problems with the *data*.

Remember!

Whenever you have an APL *error message reported*, it is best to either fix the error and *resume* execution by entering a branch to an empty vector (→⍳0), or *clear* the entry by entering an "abort" statement (→).

Common APL Error Reports

Text of Message[14]	Cause of Error; CORRECTIVE ACTION
AXIS ERROR	Axis specified does not exist, or is inappropriate for the context in which it appears.
DEFN ERROR	Misuse of ∇ symbol: 1. Improper request for an edit command while in an editor. 2. Use of other than a function name alone in re-opening a definition.
DOMAIN ERROR	Argument(s) outside the range of valid argument(s) (domain) of the function, or invalid operands for an operator.
ENTRY ERROR	An invalid character has been transmitted or received. BE SURE THAT ALL CHARACTERS ENTERED ARE AMONG THE SET SHOWN ON PAGE 11. This could also be caused by a communications line transmission failure. RE-ENTER. IF CHRONIC, RE-DIAL OR HAVE TERMINAL OR PHONE REPAIRED. IF YOU SUSPECT PHONE LINE PROBLEMS, CONTACT YOUR TELEPHONE LINE SERVICES GROUP.
INDEX ERROR	Index value out of range.
INTERRUPT	Execution was suspended within an APL statement. TO RESUME EXECUTION, ENTER A BRANCH TO THE STATEMENT INTERRUPTED: →⎕*LC* WILL RESTART THE LINE, OR →⍳0 WILL RESUME AT THE POINT WHERE THE HALT OCCURRED.
LENGTH ERROR	Shape(s) not conformable.
⎕-- *ERROR*	The system variable "⎕--" (for example, ⎕*IO*) has been set to an inappropriate value, or has been localized and not been assigned a value.
RANK ERROR	Rank(s) not conformable.

[14] The text of these error messages may be displayed in any of several national languages through the use of the system variable "⎕*NLT*." Refer to pages 143-145.

Common APL Error Reports, continued

Text of Message[14]	**Cause of Error; CORRECTIVE ACTION**
SI WARNING	The state indicator (an internal list of halted functions and operators) has been altered by editing a function or in performing a)*COPY*. "→⍳0" is disabled, but "→⎕*LC*" will restart line.
SYNTAX ERROR	Invalid syntax; for example, function or operator used without appropriate arguments or operands; unmatched parentheses, brackets, or quotes.
SYSTEM ERROR	Fault in internal operation of the system. RELOAD. IF POSSIBLE, SEND A PRINTED RECORD, INCLUDING ALL WORK LEADING TO THE ERROR, TO YOUR APL PROGRAMMING SUPPORT GROUP.
SYSTEM LIMIT	A syntactically correct statement has been entered, but cannot be executed because of an APL implementation restriction. If you are using shared variables, one possible restriction may be an attempt to simultaneously share more variables than your allotted quota permits. REQUEST A LARGER SHARED-VARIABLE QUOTA FROM YOUR APL ADMINISTRATOR. Additionally, the error could arise from attempting to share a variable when the Shared Variable Processor is not in operation. CONTACT YOUR APL SYSTEM SUPPORT REPRESENTATIVES.
VALUE ERROR	Use of a name that does not have a value, or trying to assign the "result" of a function that doesn't return a result. ASSIGN A VALUE TO THE VARIABLE OR DEFINE THE FUNCTION OR OPERATOR.
VALENCE ERROR	Use of function with the wrong number of arguments. SUPPLY THE PROPER ARGUMENT(S) OR ELIDE THE EXTRA ARGUMENT(S).
WS FULL	Workspace is filled (perhaps by temporary values produced in evaluating a multiple-step expression, or by values of shared variables). 1. CLEAR THE STATE INDICATOR 2. ERASE UNNEEDED OBJECTS 3. REVISE CALCULATIONS TO USE LESS SPACE 4. REWRITE APPLICATION TO USE EXTERNAL FILES FOR DATA STORAGE

For further information on APL2 error reports, refer to *APL2 Messages and Codes*.

In our discussions of event handling, we should take a look at the "execute" function. Execute isn't properly an event handling function, but its action is *so* similar to that of "execute alternate" — the *prime* event-handling function — that execute just seems like required reading for this context. So, here goes.

Execute

Any character vector or scalar can be regarded as a representation of an APL2 statement (which may or may not be well-formed). The monadic function denoted by "⍎" takes as its argument a character vector or scalar and evaluates or *executes* the APL2 statement it represents. When applied to a character array that might be construed as a system command or the opening of function definition, an error will necessarily result when evaluation is attempted, because neither of these is a well-formed APL2 statement.

The execute function may appear anywhere in a statement, but it will successfully evaluate only valid (complete) expressions, and its result must be at least syntactically acceptable to its context. Thus, execute applied to a vector that is empty, contains only spaces, or starts with "→" (branch symbol) or ⍝ (comment symbol) produces no explicit result and therefore can be used only on the extreme left. For example:

```
      ⍎''
      Z←⍎''
VALUE ERROR
      Z←⍎''
       ∧∧
```

The domain of ⍎ is any simple character array of rank less than two,[15] and *RANK* and *DOMAIN* errors are reported in the usual way:

```
      C←'3 4'          │
      +/⍎C             │       ⍎3 4
7                      │ DOMAIN ERROR
      ⍎1 3⍴C           │       ⍎3 4
RANK ERROR             │       ∧
      ⍎1 3⍴C           │
      ∧                │
```

An error can also occur in the attempted execution of the APL expression represented by the argument of ⍎; such an indirect error is reported by the error type and followed by the character string and the caret marking the point of difficulty, with the calling expression under that. For example:

[15] There's an additional restriction that only rarely surfaces: the character string must be composed only of valid APL2 characters; refer to the table on page 11 for a list of the complete character set.

```
      ⍎'5÷0'
DOMAIN ERROR
      5÷0
      ^^
      ⍎'5÷0'
      ^

      ⍎')WSID'
VALUE ERROR
      )WSID
       ^
      ⍎')WSID'
      ^
```

Using Execute to Assign a Value to a Supplied Name

An example of the use of Execute is a situation in which the user of the application is supplying a name for a variable, which then needs to have data assigned to it. The problem is to find a way to get data stored into a name which is really just represented as a character string. So that you don't lose sleep over this one, we'll just show you how it's done:

```
      DATA ◄──────────────── Here's some existing data
3 5 7 11 13 17 19 23

      NAME
MYDATA ◄───────────────────── We want to move it into a
                                 variable having this name
      MYDATA
VALUE ERROR ◄──────────────── ...which doesn't currently
      MYDATA                     exist.
      ^

      ⍴NAME
6

      ⍎NAME,'←DATA' ◄──────── This will do it

      MYDATA ◄─────────────── Here's the new variable
3 5 7 11 13 17 19 23
```

Now, if you wish to *extend* it, and catenate more data to what's already there, you can do this:

```
            ⍎NAME,'←',NAME,',29 31 37 41'
```

or this:

```
            ⍎NAME,'←(⍎NAME),29 31 37 41'
```

Store under this name...

The data that previously existed under the same name...

Followed by this new data

```
            MYDATA
      3 5 7 11 13 17 19 23 29 31 37 41
```

⎕*EA*: Execute Alternate

Discussions of execute have often alluded to the idea that ⍎⍞ can be used as "an alternative to ⎕ input offering more program control." Well, maybe, but any of you who have tried to actually *do* this have probably discovered that the problems start when the first user of your application types in an entry that's *not* a "well-formed APL expression":

```
      ⍎⍞
2+
SYNTAX ERROR
      2+
      ^^
      ⍎⍞
      ^
```

A function that is trying to prompt for a character string that represents a vector of floating-point numbers, and then execute it to get the string into numeric form, may well spend most of its time simply ensuring that the execution is going to work; every possibility of an erroneous input must first be checked. Perhaps the classic example of this is using "⌹" to invert a matrix: there are rules governing the acceptability of the matrix for inversion, but checking the matrix will probably take longer than the inversion. A nice approach would be to simply try it, and back off if it fails. Normal error behavior involves a halt to execution if it fails. But sometimes it's undesirable for an application to stop. Some means of getting control when an error occurs is needed. This may easily be done with execute alternate.

Consider the case of ⍎*R*. If *R* can't be executed, an error message will be returned (such as *SYNTAX ERROR*, *LENGTH ERROR*, *WS FULL*, and so forth). Execute alternate, *L* ⎕*EA R*, will return exactly the same result as ⍎*R* if the execution *is* successful; the left argument will be ignored. But if *R* can't be executed, the expression will be treated just as if it was ⍎*L*. In particular, if the left and right arguments are *both* invalid, an error message will be reported that will look just as if the expression had been ⍎*L*:

```
      ⍎'3+'
SYNTAX ERROR
      3+
      ∧∧
      ⍎'3+'
      ∧


      '2+2' ⎕EA '3+'
4


      '2+' ⎕EA '3+'
SYNTAX ERROR
      2+
      ∧∧
      '2+' ⎕EA '3+'
      ∧    ∧
```

A particularly useful application of execute alternate is "'→*OOPS*' ⎕*EA FOO*", in which any problem in the character string *FOO* which would prevent it from being executed will cause a branch to label *OOPS*.

Execute Alternate will switch arguments after any occurrence of an error in the right argument, regardless of the depth of the function calls that may have occurred in the right argument. For example, consider "'→*OOPS*' ⎕*EA* '*FN*'", where *FN* is a function. If *FN* calls another function, *FN*2, which subsequently encounters a *DOMAIN ERROR*, the error will not be reported, but rather, ⎕*EA* will immediately abandon execution of the right argument, and instead will execute the left argument (→*OOPS*).

Be aware of a frequent trap: a common approach is to enter an expression such as "*Z*←'→*OOPS*' ⎕*EA* ⍞" with the idea that an error would cause the function to branch. ...'taint so, McGee. If the input is executable, the expression can be viewed as "*Z*←⍎⍞". But if it's *not* executable, the expression becomes "*Z*←⍎'→*OOPS*'", or "*Z*←→*OOPS*" ...an immediate error. Therefore, although it's longer, a bit slower, and somewhat more cumbersome, what's *really* needed is "'→*OOPS*' ⎕*EA* '*Z*←⍎⍞'".

Please realize that this primitive is *not* meant to be an all-encompassing coverage of generalized error side-tracking all by itself. There are many situations where recovering from an error during execution will not be possible. But for situations in which you can anticipate a specific problem, and have a remedy for it, execute alternate may be just the ticket. One of the first things that you will discover as you start to use ⎕*EA* is that you need to know what the error was that occurred, and precisely where it occurred. Never fear — we'll cover those points in just a bit.

*   *   *

Here's an example of a simple input-checking function which will prompt for numeric data, and will reprompt if the input can't be executed:

```
      ∇ Z←NUM T
[1]    ⍝PROMPTS USER WITH MSG IN RT ARG, EXECUTES INPUT
[2]    START:⍞←T
[3]     Z←⍞
[4]     →(Z∧.=' ')/EXIT
[5]     '→OOPS' ⎕EA 'Z←,⍎Z'
[6]     →0
[7]    OOPS:⎕←'INVALID, PLEASE RETRY...'
[8]     →START
[9]    EXIT:Z←⍳0
      ∇    6/15/1984  14.32.18 (GMT-5)


        R←NUM 'ENTER NUMERIC STRING:  '
ENTER NUMERIC STRING:  1 2 3 4.5.6
INVALID, PLEASE RETRY...
ENTER NUMERIC STRING:  ¯3.7¯
INVALID, PLEASE RETRY...
ENTER NUMERIC STRING:  1 2 3 4. 5.6


        ⍴R
5

        R
1 2 3 4 5.6


        R←NUM 'ENTER NUMERIC STRING:  '
ENTER NUMERIC STRING:      [user just presses ENTER]

        ⍴R
0
```

Note that if the function were "simplified" a bit, it could become difficult for a well-meaning user to exit the function:

```
     ∇ Z←NUM2 T
[1]    ⍝PROMPTS USER WITH MSG IN RT ARG, EXECUTES INPUT
[2]    START:⍞←T
[3]     '→OOPS' ⎕EA 'Z←,⍎⍞'
[4]     →0
[5]    OOPS:⎕←'INVALID, PLEASE RETRY...'
[6]     →START
     ∇    6/15/1984  15.27.21 (GMT-5)

           [The user calls the function,
            but then decides to cancel
            or interrupt the function]


       R←NUM2 'ENTER NUMERIC STRING:  '
ENTER NUMERIC STRING:       [user presses ENTER]
INVALID, PLEASE RETRY...
ENTER NUMERIC STRING:  ■   [the user depresses the INTERRUPT key,
INVALID, PLEASE RETRY...   trying to halt the execution, but to
ENTER NUMERIC STRING:       no avail.]
             |
   (...and on, ad infinitum...)
             |
             ↓
```

...*The Moral:* Although there may be some legitimate times where you want to "trap" a user's input without letting him interrupt the function, be sure that you use this sort of capability with discretion; don't make your functions unresponsive to the user.

Be aware that, using execute alternate, it is possible to write *uninterruptible* functions. Be careful that you don't work yourself into a box.

Also take care to avoid name conflicts in functions that use either execute or execute alternate. A user who is entering lots of repetitive data may wish to set up a variable in the workspace, and enter its name in response to the prompt for input. That's fine, but with this particular function he would suddenly discover "mysterious" operations occurring if the name that he chose was "*T*" or "*Z*."

Another trap

"After reading the section on execute alternate, I was surprised that its usefulness was not demonstrated in the section on ambivalent functions. Your example of the *ROOT* function could very easily have employed ⎕*EA*:

```
     ∇ Z←N ROOT A
[1]    'Z←A*÷2' ⎕EA 'Z←A*÷N'          {Not recommended}
     ∇
```

— A Reader"

We received several such statements. Say what you will about programming style, but we feel that this sample function brings up a potentially dangerous situation. You are anticipating a *VALUE ERROR* if *N* isn't assigned, forcing execution of the left argument. However, *any* error arising from the use of the left argument will have the same effect — even *WS FULL*.

As an example of this, "3 *ROOT* 64 729 4096" correctly finds the cube root of each of the values, yielding "4 9 16." So far, so good. But "2 3 *ROOT* 64 729 4096" *should* produce a *LENGTH ERROR*; instead it returns "8 27 64" ***—the wrong answer—*** and would allow a calling function to continue.

We recommend against using execute alternate to circumvent normal, quick checks like this that have been traditional in the past, and we *especially* recommend against it if it's used as a return to "one-liners." (See our discussion of one-liners on pages 230-233.)

⎕*EC*: Execute Controlled

"Execute Controlled" is a way to really know what happened during execution of an expression.

The character vector right argument is executed as in "execute" (⍎), except you always get a three item result: a return code, a ⎕*ET* value, and an array.

Here's a summary of the types of expressions which can be differentiated, ordered by return code:

0 — expression in error, ⎕*ET* and ⎕*EM* returned

```
          ⎕EC '2+'
 0  2 1  SYNTAX ERROR
                    2+
                    ^^
```

1 — expression with result

```
          ⎕EC '2 3ρι6'
 1  0 0  1 2 3
         4 5 6
```

2 — assignment

```
      ⎕EC 'A←2 3⍴⍳6'
2  0 0  1 2 3
        4 5 6
```

3 — an expression with no result

```
    ⎕FX 'F X' 'Y←2*X'
F
    ⎕EC 'F 20'
3  0 0
```

4 — branch expression, returns branch target

```
      ⎕EC '→9+1'
4  0 0  10
```

5 — branch escape

```
      ⎕EC '→'
5  0 0
```

⎕ET: Event Type

If an error occurs during execution of a function, one of the first things that you need to know is, "What was the error?" This can be determined through the use of the system variable, ⎕ET.

The event type variable, ⎕ET, indicates the type of error or event which most recently occurred. Its value following)CLEAR is 0 0.

⎕ET is a two-element vector, in which the first element indicates the category of event, and the second element indicates a more specific nature of the event.

Event Type Classifications

Class	Value	Meaning
0 n **Defaults**	0 0	No error
	0 1	Unclassified event (⎕*ES* '*??*')
1 n **Resource Errors**	1 1	*INTERRUPT*
	1 2	*SYSTEM ERROR*
	1 3	*WS FULL*
	1 4	*SYSTEM LIMIT*: Symbol table full
	1 5	*SYSTEM LIMIT*: No shares available
	1 6	*SYSTEM LIMIT*: Interface quota exhausted
	1 7	*SYSTEM LIMIT*: Interface capacity exceeded
	1 8	*SYSTEM LIMIT*: Array rank too great
	1 9	*SYSTEM LIMIT*: Array size too large
	1 10	*SYSTEM LIMIT*: Array depth too great
	1 11	*SYSTEM LIMIT*: Length of prompt (via ⍞←) exceeds device input length
	1 12	*SYSTEM LIMIT*: Interface representation
2 n *SYNTAX ERROR*s	2 1	*SYNTAX ERROR*: No array (2×)
	2 2	*SYNTAX ERROR*: Ill-formed line ([( ])
	2 3	*SYNTAX ERROR*: Name class (3←2)
	2 4	*SYNTAX ERROR*: Illegal operation in context ((*A*←*B*)←2)
	2 5	*SYNTAX ERROR*:)*CS* not 0
3 n *VALUE ERROR*s	3 1	*VALUE ERROR*: Name with no value
	3 2	*VALUE ERROR*: Function with no result
4 n **Implicit Argument Errors**	4 1	⎕*PP ERROR*
	4 2	⎕*IO ERROR*
	4 3	⎕*CT ERROR*
	4 4	⎕*FC ERROR*
	4 5	⎕*RL ERROR*
	4 6	(unassigned)
	4 7	⎕*PR ERROR*
5 n **Explicit Argument Errors**	5 1	*VALENCE ERROR*
	5 2	*RANK ERROR*
	5 3	*LENGTH ERROR*
	5 4	*DOMAIN ERROR*
	5 5	*INDEX ERROR*
	5 6	*AXIS ERROR*

Note: The numbers 0 *through* 99 *are reserved for system use; user-defined events should always be given values of* 100 *or higher.*

⎕*ET* is set by the system every time an error occurs. You may set it (indirectly) through the use of the system function ⎕*ES* (event simulation). ⎕*ET* is respecified by the system every time an error occurs, so assigning a value to it directly or localizing it is not meaningful.

⎕*EM*: Event Message

One of the common things that you need to know about any error is, "Where precisely did the error occur?" This can be determined through the use of the system variable, ⎕*EM*.

The event message variable, ⎕*EM*, indicates the text of the error or event which most recently occurred. Its value following)*CLEAR* is 3 0⍴' '.

⎕*EM* shows exactly the same messages that you would see if APL was reporting the errors directly. Let's go back to a previous example that we used:

```
          FOO ←──────────── We type this in,
    DOMAIN ERROR ←───────── and get this....
    FOO[7] ZONK←5÷0
                ∧  ∧

          ⎕EM ←──────────── ⎕EM now contains the text
    DOMAIN ERROR            of the last error, as a
    FOO[7] ZONK←5÷0         simple character matrix.
                ∧  ∧

          ⍴⎕EM
    3 15
```

That character matrix from ⎕*EM* is now explicitly available, so that you can use pieces of the text in your own messages, store the text on a file, or, well — you name it.

```
DOMAIN ERROR
FOO[7] ZONK←5÷0
            ∧  ∧
```

⎕*EM* usually has three rows, but it could have more, if the original error display had more:

```
          ⍎'2+'
    SYNTAX ERROR
          2+
          ∧∧
          ⍎'2+'
          ∧

          ⍴⎕EM
    5 12
```

An Example of the Use of ⎕*EM*

Let's say that you want to report the normal error messages that the system always reports, but you don't want to halt execution. *...Piece of cake.* Here's a part of an APL tutorial which evaluates what the user enters by simply passing it back to APL to execute, and then reports the result and continues:

```
ENTER AN EXPRESSION TO ASSIGN TO "AREA"
THE RESULT OF PI TIMES R SQUARED
(THE AREA OF A CIRCLE):

      AREA←○1)×R*2 ←——————— We typed this in,
SYNTAX ERROR                but we forgot the
      AREA←○1)×R*2          other parenthesis.
      ^
...NOT QUITE RIGHT; THE CORRECT ANSWER IS:

      AREA←(○1)×R*2    OR    AREA←(○1)×R×R
   OR AREA←○R*2        OR    AREA←○R×R
```

Hmmm... how did this lesson display the error that the student created — the student, of course, *could* have typed in *anything*. How can the lesson check for all of the possibilities? Well, it can't — and it doesn't. It simply uses execute alternate to try to execute the input, and if it doesn't work, it reports the error that would have been generated, using ⎕*EM* — and then *continues execution* of the lesson (...very important!).

Here is an example of the code that could be used for that previous example:

```
     ∇ SCORE←QUIZ19;INPUT;AREA;R;ANS
[1]     'ENTER AN EXPRESSION TO ASSIGN TO "AREA"'
[2]     'THE RESULT OF PI TIMES R SQUARED'
[3]     '(THE AREA OF A CIRCLE):'
[4]     ANS←(○1)×R×R←AREA←*7
[5]     INPUT←ASK '       '
[6]     '→ERROR' ⎕EA INPUT
[7]     →(AREA≡ANS)/RIGHT
[8]     →WRONG
[9]    ERROR:⎕EM
[10]   WRONG:
[11]    '...NOT QUITE RIGHT; THE CORRECT ANSWER IS:'
[12]    ''
[13]    '       AREA←(○1)×R*2      OR     AREA←(○1)×R×R'
[14]    '  OR   AREA←○R*2          OR     AREA←○R×R'
[15]    ''
[16]    SCORE←0
[17]    →0
[18]   RIGHT:SCORE←1
[19]    'THAT''S RIGHT!'
       ∇
```

We will talk about this "*ASK*" function on page 196.

Here is the display of ⎕*EM*.

That last example is trying to show a real application in which ⎕*EM* may realistically be used. In its bare-bones form, however, that example may be reduced to a form as simple as this:

```
'⎕EM' ⎕EA INPUT
```

← Execute the input; if an error occurs, report it, but continue processing.

The display of the message in ⎕*EM* is affected by the setting of ⎕*NLT* (national language translation; see pages 143-145):

```
      ⎕NLT←'FRANCAIS'
      XYZ
VALEUR NON DEFINIE
      XYZ
      ∧

      ⎕EM
VALEUR NON DEFINIE
      XYZ
      ∧
      ρ⎕EM
3 18

      ⎕NLT←'DEUTSCH'
      ⎕EM
VALEUR NON DEFINIE ←———————  The value of ⎕EM is based
      XYZ                    upon the setting of ⎕NLT
      ∧                      at the time that the
      ρ⎕EM                   error occurred.
3 18
```

You may notice that ⎕*EM* — and in fact, the APL error messages themselves — don't have as fine a resolution of the problem as ⎕*ET*. As an example of this,

both

3 1	(name with no value)

and

3 2	(function with no explicit result)

will be reported at the terminal and in ⎕*EM* as "*VALUE ERROR*."

Also, because ⎕*EM* is affected by ⎕*NLT*, it is *strongly recommended* that you *do not* try to keep a table of error messages and look up the text from ⎕*EM* when an error occurs. The first time that a user of your application sets ⎕*NLT* to something unexpected, the application would mysteriously stop working. And, no, you can't just localize ⎕*NLT* when you look at ⎕*EM*, because as we saw above, ⎕*EM* shows the message in the language that was active when the event occurred. Besides, table look-up with ⎕*EM* is too much like work... you'll find life *much easier* if you just look at ⎕*ET* to determine the event type (...that's what it's for). Just in case you missed ⎕*ET*, it's back on pages 177-179. Go for it.

⎕*EM* is set by the system every time an error occurs. You may set it (indirectly) through the use of the system function ⎕*ES* (event simulation). ⎕*EM* is respecified by the system every time an error occurs, so assigning a value to it directly or localizing it is not meaningful.

⎕*ES*: Event Simulation

Did you ever want to have the capability of issuing *your own* error messages that acted *just like* the system-generated variety? No? ...Then skip this section.

If you *did* ever consider trying to write such a piece of code, you would find that knowing where to point the error carets and where to return control would be a formidable task. It would be quite a task *without* ⎕*ES*, that is. ⎕*ES* is a system function that allows you to simulate the system's action with respect to the display of error messages.

When an error occurs, the system does several things:

- An error message is issued.
- The offending line is displayed. (It may be a line in a function, or a line from immediate-execution.)
- Error carets are displayed to show where the events occurred.
- ⎕*EM* is assigned the error message.
- ⎕*ET* is assigned the event type.
- ⎕*L* and ⎕*R* are assigned the values of the left and right arguments.
- The execution is halted and suspended (if appropriate), unless it's under the control of ⎕*EA* (described on pages 172-176).

Just as primitive functions can generate errors in your statements when they're unhappy about your arguments, *you* can cause *your* programs to generate errors *in the functions that called your function*, using event simulation. ⎕*ES* allows you to control the actions listed above, while giving you full control to selectively change portions of them. Let's examine the rules for coding the arguments of ⎕*ES*.

Monadic ⎕*ES*:

The right argument for ⎕*ES* can be either a numeric event type (like we used with ⎕*ET*), or a message.

If you supply the *event type*, it must be a two-element numeric vector — again, just like ⎕*ET*. If the values are found in the ⎕*ET* table on page 178, the corresponding message shown there will be reported (that is, a 1 3 will report a *WS FULL*, and so forth). If the values that you supply *aren't* in that table, then *no* message will be displayed. But you'll still get the rest of the error report, replete with error carets and all of the standard embellishments.

In any event (as it were), ⎕*ET* will be set to the value which you supply. This lets you check things as you go along; you can make up your own error codes (in effect, create your own table), but it's suggested that you use a first element of 100 or higher for *your* events, so that you won't run into a conflict if APL2 extends its list someday.

If you supply a *message* as the right argument, ⎕*ET* will always be set to 0 1 (meaning "unclassified event"), and that message will be displayed along with the rest of the regular bells and whistles.

A ⎕*ES* with an argument of 0 0 will reset ⎕*ET* and ⎕*EM* to their default values, but will not simulate an event.

```
      ∇ FOO
[1]     ⎕ES 5 3
[2]     2+2
      ∇


        FOO
LENGTH ERROR
        FOO
        ^
```

Notice that the error is reported as though the *FOO* function was a locked function (or a primitive); that is, *FOO* is *not* suspended and the line that displays is the one that *called* the *FOO* function, and *it* is the one that gets suspended.

Notice also that execution never proceeded past the ⎕*ES* statement.

```
      ⎕ET
5 3

   ∇ GOO
[1]   ⎕ES 'MESSAGE'
   ∇

      GOO
MESSAGE
      GOO
      ^

      ⎕ET
0 1
  ↖ (unclassified event)

      ⎕EM
MESSAGE
      GOO
      ^


      ρ⎕EM
3 9
```

```
      ∇ HOO
[1]     'ANOTHER MESSAGE'
[2]     ⎕ES 100 11
      ∇

        HOO
ANOTHER MESSAGE
        HOO
        ^

        ⎕ET
100 11


        ⎕EM

        HOO
        ^


        ρ⎕EM
3 9
```

But what if you want to supply *both* a message of your own *and* an event type of your own, and you didn't like the two-line approach shown in the *HOO* example? In that case, you can use the dyadic form of ⎕*ES*, discussed below.

Dyadic ⎕*ES*:

In its dyadic form, ⎕*ES* accepts a two element numeric vector representing an event type as its right argument, and your message as its left argument. As in the monadic case, the numeric right argument will set ⎕*ET*, and the message in the left argument will be displayed.

```
      ∇ MOO
[1]     'YOUR MESSAGE HERE' ⎕ES 5 3
      ∇

        MOO
YOUR MESSAGE HERE
        MOO
        ∧

        ⎕ET
5  3 ◄──────────────── This is listed as a LENGTH ERROR
                       in the table on page 178


        ⎕EM
YOUR MESSAGE HERE
        MOO
        ∧


        ρ⎕EM
3  17
```

Displaying "No Event"

You never have to bother branching around ⎕*ES*; you can issue a test as part of the right argument, allowing that argument to be either one of the forms described above, or an empty vector. An empty vector right argument causes ⎕*ES* to be ignored. Execution will proceed right on past ⎕*ES*, with no event reported, and no setting of ⎕*ET*.

As an example, here's a defined function called *PROGRAM* which expects three-element vectors as arguments. On line one it checks to see if the shape of the data is 3 and if not, it selects 5 3 (which is the ⎕*ET* code for *LENGTH ERROR*). Therefore, the argument that ⎕*ES* sees is 5 3 unless the length of the input is 3:

```
      ∇ Z←PROGRAM DATA
[1]     ⎕ES (3≠⍴DATA)/5 3
      ∇
```

Now let's execute the function, giving it only a two-element vector:

```
      PROGRAM 'AB'
LENGTH ERROR
      PROGRAM 'AB'
      ∧
```

As requested, we get a *LENGTH ERROR* in the calling expression as though *PROGRAM* were a primitive function.

Let's look at a few more examples:

```
    ∇ Z←A DIVIDED_BY B
[1]   ⎕ES (0∊B)/'DIVISION BY ZERO NOT ALLOWED'
[2]   Z←A÷B
    ∇

      10 20 30 DIVIDED_BY 1 2 5
10 10 6

      10 20 30 DIVIDED_BY 1 2 5-1
DIVISION BY ZERO NOT ALLOWED
      10 20 30 DIVIDED_BY 1 2 5-1
      ^        ^

      ⎕ET
0 1

    ∇ RESULT←SQUARE ARRAY
[1]   'MUST BE SIMPLE' ⎕ES (1<≡,ARRAY)/101 1
[2]   'MUST BE NUMERIC' ⎕ES (' '=1↑0ρARRAY)/101 2
[3]   RESULT←ARRAY*2
    ∇

      SQUARE 1234
1522756

      SQUARE 1 2 3 (4 5) 6 7
MUST BE SIMPLE
      SQUARE 1 2 3(4 5)6 7
      ^
      ⎕ET
101 1

      SQUARE '65536'
MUST BE NUMERIC
      SQUARE '65536'
      ^
      ⎕ET
101 2
```

And have you ever wished that you could just get in a couple of extra instructions between the time that APL detects an error and the time that it reports that error? (Well, pretend that you have, so that we can discuss it.) The combination of execute alternate (⎕*EA*), event type (⎕*ET*), and event simulation (⎕*ES*) allows you to construct a function which can proceed until it detects an error, and then do whatever additional processing you wish before it reports that error. For example:

```
      ∇ RESULT←A DIVIDED_BY B
[1]     '→ERROR' ⎕EA 'RESULT←A÷B'
[2]     →0
[3]    ERROR:
[4]     ⍎(0∊B)/'⎕←''DIVISION BY ZERO'''
[5]     ⎕ES ⎕ET
      ∇
```

Errors in a Defined Function: Normal Behavior

When an error in a defined function occurs, the name of the function, the line number in brackets, and the line from the function are displayed.

For example, given the function *F* which does addition:

```
      ∇ Z←L F R
[1]     Z←L+R
      ∇
```

...executing it with a character argument will cause a *DOMAIN ERROR* on line one of the function:

```
      2 3 F 'A'
DOMAIN ERROR
F[1]  Z←L+R
       ^ ^
```

You would also find the same three lines in ⎕*EM*.

)*SIS* shows the line in error as well as the expression that caused the defined function to be called:

```
      )SIS
F[1]  Z←L+R
       ^ ^
*   2 3 F 'A'
    ^   ^
```

This is what happens by default, but you can alter this behavior in several ways which we will discuss next.

Errors in a Defined Function: Special Behavior

There are four properties of functions that can affect what happens when an error occurs.

Execution Properties

1. ***Cannot Be Displayed or Altered***

   If a function is marked nondisplayable, then the statement in error will not be displayed. Also, the function may not be edited.

2. ***Cannot Be Suspended***

   If the function is marked nonsuspendable, then after error processing in the function, it is removed from the state indicator and an error is generated in the calling environment of the function.

3. ***Weak Attention Ignored***

   This does not affect error conditions. It means that a weak attention will not stop the execution of this function.

4. ***Errors Changed to*** *DOMAIN ERROR*

   Any error other than a resource error is turned into a *DOMAIN ERROR*. By a "resource error," we mean one that is reporting a physical limitation of the system — such as a *WS FULL*. These environmental errors are always reported in their usual way; they are not mapped to *DOMAIN ERROR*s.

APL2 primitive functions have the first three options enabled; your own programs may have any combination.

Your functions are given these properties when they are defined by using dyadic ⎕*FX* with a four-element left argument, which is set to one for each of the properties desired. Refer to page 129 for additional details.

This will fix a function named *PROGRAM*, which will cause any errors to be reflected back to the caller of the function:

```
      0 1 0 0 ⎕FX 'Z←PROGRAM DATA' 'LINE 1' 'LINE 2'
```

Chapter 5: Adding a Professional Appearance

This chapter shows you some tips on how to make your application packages look better to other users. It's important to consider human factors with any programming endeavor. We'll be discussing *Terminal Input and Output*, so that you can communicate with the user more effectively, *Grade*, so that your data can be sorted in human-readable fashion, and *Formatting*, so that you can make your numeric data look better and convey more meaning.

Used with the permission of The Dick Sutphen Studio.

Description of Features and Facilities

Terminal Input and Output

Evaluated Input

In many significant applications, such as text processing, for example, it is necessary that the user supply information as the execution of the application programs progresses. It is also often convenient, even in the use of an isolated function, to supply information in response to a request, rather than as arguments to the function as part of the original entry. This is illustrated by considering the use of the function *CI*, which determines the growth of a unit amount invested at periodic interest rate *RATE* for number of periods *TIME*:

```
      ∇ Z←RATE CI TIME
[1]     Z←(1+RATE)*TIME
      ∇
```

For example, the value of 1000 dollars at 18 per cent for 7 years, compounded quarterly, might be found by:

```
        1000 × (.18÷4) CI 7×4
3429.699993
```

The casual user of such a function might, however, find it difficult to remember which argument of *CI* is which, how to adjust the rate and period stated in years for the frequency of compounding, and whether the interest rate is to be entered as the actual rate (for example, 0.18) or as a percentage (for example, 18). An exchange of the following form might be more suitable:

```
      INVEST
ENTER CAPITAL AMOUNT IN DOLLARS
⎕:
      1000
ENTER NUMBER OF TIMES COMPOUNDED IN ONE YEAR
⎕:
      4
ENTER ANNUAL INTEREST RATE IN PERCENT
⎕:
      18
ENTER PERIOD IN YEARS
⎕:
      7
VALUE IS 3429.699993
```

It is necessary that each of the entries (1000, 4, 18, and 7) occurring in such an exchange be accepted not as an ordinary entry (which would only evoke the response 1000, and so forth), but as data to be used within the function *INVEST*. Facilities for this are provided in two ways, termed *evaluated input*, and *character input*. A definition of the function *INVEST*, which uses evaluated input, is as follows:

```
     ∇ INVEST;C;F;R;T
[1]    'ENTER CAPITAL AMOUNT IN DOLLARS'
[2]    C←⎕
[3]    'ENTER NUMBER OF TIMES COMPOUNDED IN ONE YEAR'
[4]    F←⎕
[5]    'ENTER ANNUAL INTEREST RATE IN PERCENT'
[6]    R←⎕÷F×100
[7]    'ENTER PERIOD IN YEARS'
[8]    T←F×⎕
[9]    'VALUE IS ',⍕C×R CI T
     ∇
```

This use of the ⎕ for input is also sometimes called simply "quad input." The term "evaluated input" springs from the fact that ⎕, when used for input, evaluates whatever is entered before passing along the result. Therefore, if the user had entered "500×2" in response to that first prompt, the variable *C* (on line 2) would have been given the same value of 1000.

If an error occurs during the entry of an input, the user will be reprompted with another "⎕:" — but *not* with the text preceding it. This inability to react more fully to errors during keyboard entry, and the inability to repeat the character prompt if the user types in an entry which results in an error, make evaluated input generally undesirable for a production application with users who may not be familiar with APL.

Better control over the input, and therefore, better communication and interaction with the user, can be accomplished through the use of character input. The trade-off is that character input returns its data in character form (of course), and therefore must rely upon the execute function (or ⎕*EA*) in instances where a numeric form is required. See the description of ⎕*EA* on pages 172-176 for some considerations on this subject. (But stick around for the rest of the character input discussions first.)

Since both numeric and character inputs are required for any real applications, and since the use of execute on the character inputs can bring up a sizable set of additional programming considerations, both ⎕ and ⍞ input have their place.

Character Input

It is often desired to prompt a user for some text which he would like to be able to enter in much the same fashion as he would enter it on a typewriter. For example, the casual user shouldn't have to be concerned with such programming details as putting quotation marks around his response.

This is easily handled by using quad quote (⍞) for input, which returns a character vector. It's used like this:

```
     ∇ REPORT;TITLE;NAME;X
[1]    'ENTER A TITLE FOR THIS REPORT:'
[2]    TITLE←⍞
[3]    'ENTER YOUR NAME:'
[4]    NAME←⍞
[5]    'TURN TO A NEW PAGE AND PRESS RETURN:'
[6]    X←⍞
[7]    'REPORT ON ',TITLE,', COMPILED BY ',NAME
[8]    SUB_FUNCTION
     ∇
```

The user of this *REPORT* function wouldn't be required to know much about the details of APL to use this; the prompts ask him for whatever is needed, and he just types in his responses. Whatever he enters will be taken as text, with no evaluation whatsoever. Therefore, he doesn't need to enter his name, *DAVIS*, as '*DAVIS*' ...*nor* does he have to be concerned with the "intricacies" of entering *O'RILEY* as '*O''RILEY*'. If he *did* enter '*O''RILEY*', the result of the ⍞ input would be a *ten*-element vector.

Note that referencing ⍞ *always* returns a vector. In particular, if the input is a single character, it is returned as a one-element vector rather than as a scalar. Some of the previous versions of APL returned a scalar if a single character was entered via "⍞", but it seemed that most APL programmers tended to *ravel* that result every time to ensure that they had a vector, so this ended up being a a simplification of the rules for most applications. There's no need to ravel this; it will always be a vector.

Bare Output

Normal output includes a concluding new line signal so that the succeeding display (either input or output) will begin at a standard position on the following line. *Bare output*, denoted by expressions of the form ⍞←*X*, does not include this signal if it is followed either by another bare output or by character input (of the form *X*←⍞). The new line signals that would be supplied by the system in order to break lines that exceed the printing width are *not* supplied with bare output. However, since an expression of the form ⍞←*X* entered directly from the keyboard (rather than being executed as part of a defined function) must necessarily be followed by another keyboard entry, the output it causes *is* concluded with a new line signal. The effect is in this case indistinguishable from normal output, except for the possibility of exceeding the printing width limitation.

We can readily discern two distinct classes of operations for which bare output followed by character input is most commonly used; these are *prompting* and *editing*. You can tell APL which of these two actions you wish to invoke by setting a system variable, ⎕*PR* (prompt replacement). Let's look at some examples of each of these.

Prompting

If the Prompt Replacement variable is set to a *single blank*, ⎕*PR*←' ', then the text of the prompt will be replaced by blanks. This is the default case in a clear workspace. In the above example, character input following a bare output is treated as though the user had spaced over to the position occupied at the conclusion of the bare output, so that the characters received in response will normally be prefixed by a number of space characters. This allows for the possibility that, after the keyboard is unlocked, the user backspaces into the area occupied by the preceding output.

The following function prompts the user with whatever message is supplied as its argument, and returns the response:

```
      ∇ ANSWER←ASK QUESTION
[1]     ⍞←QUESTION
[2]     ANSWER←⍞
      ∇
```

Using such a function, the expression:

```
        RESPONSE←ASK 'ENTER CAPITAL: '
```

would have the following effect:

```
        displayed by the system
________________
ENTER CAPITAL: 1000
               ____
                    entered by the user
```

The value of *RESPONSE* is as many blank characters as there are characters in *QUESTION*, followed by the characters entered by the user. In this case, *RESPONSE* will have fifteen blanks followed by the four characters "1000".

It is worth noting here that we're depending upon the global value of ⎕*PR* being a blank for this function to work properly. In actual practice, it may be set to some other value in whatever workspace this function gets used. If its value is other than a blank, this function could start to fail in unusual situations; its operation may then be unpredictable. For this reason, it's always smart to localize ⎕*PR* to the prompting function. In that way, you can guarantee a proper setting of ⎕*PR*, and therefore, can guarantee the operation of the function. With ⎕*PR* localized, the function would look like this:

```
      ∇ ANSWER←ASK QUESTION;⎕PR
[1]     ⎕PR←' '
[2]     ⍞←QUESTION
[3]     ANSWER←⍞
      ∇
```

It is strongly recommended that prompting for user input be done with a *prompting subfunction*, such as we have looked at here. This gives you several advantages: First, if you ever transfer your functions to a system whose behavior with respect to quad quote differs from this system, a change to that single prompting function is sufficient to modify all of the prompting for your application. Second, it gives you the ability to easily introduce additional function in the future. And finally, the use of a prompting subfunction allows you to modify the appearance of all of the prompts by changing one function. You can easily start with a very simple function, like the one above, and add structure as it's needed. Try it; you'll quickly find that an *ASK* function is worth its weight in gold (however much *that* is).

There are some other common cases that you'll undoubtedly want to handle with this prompting function. Let's get some background in the editing case first, and then we'll talk about prompting some more.

If the prompt replacement variable is set to an *empty vector*, ⎕PR←'' , then the text of the prompt will not be replaced at all. In this case, character input following a bare output will cause *both* the text of the output *and* the user's response to be returned. The characters received in response will normally be prefixed by the text of the original output. Normally in this case, you will be expecting the user to backspace into the area occupied by the preceding output and to type over the original text.

The following function presents the user with whatever text is supplied as its argument, and returns both that text and the response:

```
      ∇ RESULT←EDIT STRING;⎕PR
[1]     ⎕PR←''
[2]     ⍞←STRING
[3]     RESULT←⍞
      ∇
```

Using such a function, the expression:

```
      NEW_TEXT←EDIT 'THIS IS A LINE OF TEXT'
```

would have the following effect:

displayed by the system

```
THIS IS A LINE OF TEXT
               TO EDIT
```

the user backspaces and types on top of the output.

The value of *NEW_TEXT* is now "*THIS IS A LINE TO EDIT*".

Here are some comparisons of various forms of bare output followed by character input:

```
⎕PR←' '
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input                     JON           3
Result                    JON          16

⎕PR←' '
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input         XYZ  User backspaces
Result        XYZ                      13
```

```
⎕PR←''
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input                     JON           3
Result       ENTER NAME:  JON          16

⎕PR←''
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input         XYZ  User backspaces
Result       EXYZR NAME:               13
```

```
⎕PR←'='
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input                     JON           3
Result       =============JON          16

⎕PR←'='
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input         XYZ  User backspaces
Result       =XYZ=========             13

⎕PR←'='
             ----+----|----+----|  Length
Output       ENTER NAME:               13
Input         XYZ
Result       =XYZ                       4
```

User backspaced
and pressed the
"ERASE EOF" key
or "ATTENTION."

If you are using bare output followed by quad quote input for *prompting* (as opposed to editing), you probably will want to end up with *only* the text that the user typed in. The prompt itself and any extraneous blanks or fill characters need to get discarded somewhere along the way. If you have ⎕*PR* set to an empty vector, the prompt will be returned, and that could be messy to remove — you shouldn't just use (ρ,*STRING*)↓⍞ because the user may have backspaced into the prompting area, and dropping off those characters would cause you to lose some of his response. And if you set ⎕*PR* to a single blank, the prompt will get replaced by blanks — but compressing out all of the blanks isn't a good method either, because his response may have contained one or more blanks. What you really want to do is to get back just the text that the user keyed in, with no extraneous characters. Fortunately, there's an easy way to do this.

In that last group of three examples, we assigned an arbitrary character (in this case an "equals" sign) to ⎕*PR*. This gives us the capability of determining that the user entered a space character — in other words, a space on the input is distinct from the replacement character.

Be careful, though: If you use this last technique, make sure that the character that you choose is one that the user *can't* enter himself. It's generally not good enough to simply choose one that the user *probably* won't enter, since surprises may greet the user who enters something just a little different. An example of a value that *can't* match the user's input would be a carriage-return.

If you set ⎕*PR*←⎕*TC*[2] (which is a carriage-return character in origin 1), then each position of the prompt will be replaced with a carriage-return. Quad quote returns everything up to but not including the first carriage-return that the user enters (by definition), so it can't contain a carriage-return. This replacement process then substitutes carriage-return characters for all of the unchanged positions of the prompt. Therefore, using a carriage-return character for the prompt replacement character guarantees that it can't ever get confused with what the user enters.

If you then wish to return *only* the user's response, *without* the prompt *or* leading blanks (and, of course, without the carriage-return characters), you may simply use the "without" function (dyadic "~"), like this:

```
      ∇ ANSWER←ASK2 QUESTION;⎕PR;⎕IO
[1]     ⎕IO←1              ⍝ORIGIN 1
[2]     ⎕PR←⎕TC[2]         ⍝REPLACE PROMPT WITH CARRIAGE-
[3]   ⍝                     RETURN CHARACTERS
[4]     ⍞←QUESTION         ⍝DISPLAY THE PROMPT;
[5]   ⍝                     RETURN USER'S RESPONSE
[6]     ANSWER←⍞~⎕PR       ⍝WITHOUT THE PROMPT
      ∇
```

Using such a function, the expression:

```
        RESPONSE←ASK2 'ENTER YOUR NAME:   '
```

would have the following effect:

```
                 displayed by system
ENTER YOUR NAME:   JON
                         user's input
```

The value of *RESPONSE* is now "*JON*", and contains only three characters.

"Ask Each"

For places where you don't necessarily need to check and verify the contents of the response from the user between each prompt, here's a quick way to issue multiple prompts:

```
      NAME←ASK2¨'FIRST NAME:    ' 'LAST NAME:    '
FIRST NAME:   JOHN
LAST NAME:    DOE

      NAME
 JOHN DOE
      ρNAME
2
      ρ¨NAME
 4  3
```

NAME now contains a two-item nested vector, where the first item contains a four-element vector, "*JOHN*", and the second item contains a three-element vector, "*DOE*".

Because the APL function can't examine the results until everything within the scope of the "each" operator has been processed, and because the user can't even exit from the prompts without interrupting the function, this construction isn't being shown with the idea that you'll necessarily find a place for it in your own applications. But it is a construction that's often surprising at first appearance, and may spur some creative ideas for your applications.

Editing the Contents of a Function or Variable

Remember the discussion of the transfer form function, ⎕*TF*, back on pages 132-135? ...Here's a little creative application of its abilities.

The following function takes as its argument the *name* of a function or variable, and gives you an easy way to edit the lines of that function or the values stored in that variable:

```
    ∇ MODIFY IT;⎕PR
[1]   ⎕PR←''
[2]   ⍞←2 ⎕TF IT ←——— display the transfer form
[3]   2 ⎕TF ⍞ ←——————— and reconstruct the object
    ∇
```

Assume that we have a variable in the workspace that looks like this:

```
          M
105    20     8
 40  1000  NONE
  1     0     ?
 15   N/A    12

          ⍴M
4  3
```

The function is used like this:

```
      MODIFY 'M'
M←4 3⍴105 20 8 40 1000 'NONE' 1 0 '?' 15 'N/A' 12
```

The cursor stops at the end of the line (way over here) and the user is free to back up and type over the "*M←*..." line; whatever he sees on his screen when he presses ENTER will be re-assigned to the variable *M*.

While this approach is certainly only reasonable for small arrays, it does makes the editing of those arrays very easy.

We could *even* use the *MODIFY* function to modify the *MODIFY* function(!):

```
      MODIFY 'MODIFY'
⎕FX 'MODIFY IT;⎕PR' '⎕PR←''''' '⍞←2 ⎕TF IT' '2 ⎕TF ⍞',
```

The cursor stops at the end of the line again, and the user can type over the present lines in the function. When he presses enter, the system prints "*MODIFY*"... the name of the new function. Of course, if the *name* was part of what was modified, an entirely *new function* will be created.

As with the previous example, this approach is certainly only reasonable for very small functions — but it's elegant for those, and it might be the most compact function editor you'll ever see!

Grade

An easy-to-use sorting capability is available through the use of the grade function. Grade can directly sort character arrays. By specifying the desired collating sequence as the left argument, you may order the rows of the array according to your application requirements:

```
      M←4 4⍴'NOW IS  THE TIME'
      ALF←' ABCDEFGHIJKLMNOPQRSTUVWXUZ0123456789'

      ALF⍋M
2 1 3 4

      M[ALF⍋M;]
IS
NOW
THE
TIME
```

The left argument of grade can be whatever alphabet is best suited to your own application. Here is a sample text matrix sorted four different ways:[16]

[16] The examples are from "Sorting — A New/Old Problem," by Howard J. Smith, from the APL79 Conference Proceedings. Copyright 1979 by the Association for Computing Machinery, Inc., Reprinted by permission. This paper also provides some additional discussion of sorting, and the use of dyadic grade.

```
         ' abc...xyz'          |           ' abc...ABC...'
                               |
   ama                         |     ama
   phosphate                   |     phosphate
   pH                          |     pH
   Philodendron                |     Ama
   Ama                         |     AMA
   AMA                         |     Philodendron
-------------------------------+------------------------------
                               |      2 27ρ abc...xyz
         ' aAbBcCdD...'        |            ABC...XYZ
   ama                         |     ama
   Ama                         |     Ama
   AMA                         |     AMA
   phosphate                   |     pH
   pH                          |     Philodendron
   Philodendron                |     phosphate
```

Note that any of the sorting methods can take care of the obvious cases, such as putting "*ama*" first, but the cases rapidly become more complex when we introduce different fonts (both underscored and nonunderscored characters, or caps and lowercase) within the same matrix. In the fourth case, a matrix left argument was used, looking like this:

```
      ALF←2 27ρ' abc...xyz ABC...XYZ'

      ALF
 abcdefghijklmnopqrstuvwxyz
 ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

This case says that our primary interest in sorting the text should be spelling. Both fonts have the same weighting, because the alphabetics in each row are aligned. But if several words have identical spelling ("*ama*", "*Ama*", and "*AMA*"), then they should be sorted according to the row order of the fonts in the left argument. Thus, "*ama*", "*Ama*", and "*AMA*" are grouped together in the final list.

If a character in the right argument doesn't appear at all in the left argument, it's treated in a fashion analogous to the action of *A* ι *B* — the unknown characters will be pushed to the end of the list, in "first-come-first-served" order:

```
      M←5 3⍴'○○○XXXAAA⎕⎕⎕∩∩∩'

      M['⎕○∩'⍋M;]
⎕⎕⎕
○○○
∩∩∩
XXX
AAA
```

Format

The symbol "⍕" denotes three *format* functions which convert numeric arrays to character arrays. These three functions are:

- Monadic format
- Format by Specification (numeric left argument)
- Format by Example, also known as Picture Format (character left argument)

There are several significant uses of these three functions in addition to the obvious one for composing tabular output. For example, the use of format is complementary to the use of execute in treating bulk input and output, and in the management of combined alphabetic and numeric data.

Monadic Format

The *monadic* format function produces a character array which will display identically to the display normally produced by its argument, but makes this character array explicitly available. For example:

```
      M←2=?4 4⍴2

      C←⍕M

      M
0 1 0 1
0 0 1 1
1 0 1 1
0 0 1 1

      ⍴M
4 4

      ⍴⍕2 5
3

      C≡⍕C
1
      ⍕'ABCD'
ABCD
```

```
      C
0 1 0 1
0 0 1 1
1 0 1 1
0 0 1 1

      ⍴C
4 7
```

```
      C[;1 3 5 7]
0101
0011
1011
0011
```

The format function applied to a simple character array yields the array unchanged, as illustrated by the last two examples above. For a numeric array, the shape of the result is the same as the shape of the argument except for the required expansion along the last coordinate, each number going, in general, to several characters. The format of a scalar number is always a vector.

The format of a nested array yields a simple (nonnested) array. Therefore,

```
      ⍕10 20 'NONE'
 10 20 NONE
      ' '=⍕10 20 'NONE'
1 0 0 1 0 0 1 0 0 0 0 1
```

The printing normally produced by &APL. systems may vary slightly from system to system, but in the case of a simple (nonnested) vector or scalar argument, the result produced by the monadic format will have no initial spaces and no final column of all spaces.

Format by Specification

Dyadic format with a numeric left argument is sometimes called Format by Specification. This *dyadic* format function uses variations in the left argument to provide progressively more detailed control over the result. Thus, for *F*⍕*A*, the argument *F* may be a single number, a pair of numbers, or a vector of length 2×¯1↑1,⍴*A*.

In general, a pair of numbers is used to control the result: the first determines the total width of a number field, and the second sets the precision. For decimal form the precision is specified as the number of digits to the right of the decimal point, and for scaled form it is specified as the number of digits in the multiplier. The form to be used is determined by the sign of the precision indicator, negative numbers indicating scaled form. Thus:

```
      A← 3 2 ⍴ 12.34 ¯34.567 0 'NONE' ¯.26 ¯123.45

      ⍴⎕←A
 12.34  ¯34.567
 0         NONE
¯0.26 ¯123.45
3 2

      ⍴⎕←9 2⍕A
    12.34   ¯34.57
      .00     NONE
     ¯.26  ¯123.45
3 18

      ⍴⎕←9 ¯2⍕A
   1.2E1    ¯3.5E1
   0.0E0      NONE
  ¯2.6E¯1   ¯1.2E2
3 18

      ⍴⎕←12 3⍕A
      12.340     ¯34.567
        .000        NONE
       ¯.260    ¯123.450
3 24

      ⍴⎕←6 0⍕A
    12   ¯35
     0  NONE
     0  ¯123
3 12

      ⍴⎕←7 ¯1⍕A
   1E1    ¯3E1
   0E0    NONE
  ¯3E¯1   ¯1E2
3 14
```

If the width indicator of the control pair is zero, a field width is chosen such that at least one space will be left between adjacent numbers. If only a single control number is used, it is treated like a number pair with a width indicator of zero:

```
      ρ⎕←0 2⍕A
 12.34   ¯34.57
   .00     NONE
  ¯.26  ¯123.45
3 14

      ρ⎕←2⍕A
 12.34   ¯34.57
   .00     NONE
  ¯.26  ¯123.45
3 14
```

```
      ρ⎕←0 ¯2⍕A
 1.2E1   ¯3.5E1
 0.0E0     NONE
¯2.6E¯1  ¯1.2E2
3 15

      ρ⎕←¯2⍕A
 1.2E1   ¯3.5E1
 0.0E0     NONE
¯2.6E¯1  ¯1.2E2
3 15
```

Each column of an array can be individually composed by a left argument that has a control pair for each:

```
      ρ⎕←0 2 0 2⍕A
 12.34    ¯34.57
   .00      NONE
  ¯.26   ¯123.45
3 14

      ρ⎕←6 2 12 ¯3⍕A
 12.34       ¯3.46E1
   .00          NONE
  ¯.26       ¯1.23E2
3 18
```

```
      ρ⎕←8 3 0 2⍕A
 12.340   ¯34.57
   .000     NONE
  ¯.260  ¯123.45
3 16

      ρ⎕←8 0 0 ¯2⍕A
      12 ¯3.5E1
       0   NONE
       0 ¯1.2E2
3 15
```

```
      6 2 8 3 3 0 4 0 5 0 12 4⍕,A
 12.34 ¯34.567  0NONE     0    ¯123.4500
```

The format function applied to an array of *rank greater than two* applies to each of the planes defined by the last two axes. For example:

```
      L←2=?2 2 5⍴2
      L
1 1 0 0 1
1 1 1 0 1

1 0 0 1 0
0 0 0 0 0

      ⍴L
2 2 5
```

```
      4 1⍕L
 1.0 1.0  .0  .0 1.0
 1.0 1.0 1.0  .0 1.0

 1.0  .0  .0 1.0  .0
  .0  .0  .0  .0  .0

      ⍴4 1⍕L
2 2 20
```

Tabular displays incorporating row and column headings, or other information between columns or rows, are easily configured using the format function together with catenation. For example:

```
      ROWHEADS←'' 'JAN' 'APR' 'JUL' 'OCT'
      YEARS←79+⍳5
      TABLE←.001×¯4E5+?4 5⍴8E5
      ROWHEADS,(2⌽9 0⍕YEARS),[1] 9 2⍕TABLE
        80       81       82       83       84
JAN  ¯294.77   ¯33.08  ¯224.83   143.09   347.76
APR    15.53  ¯372.34    23.76  ¯393.84   ¯66.01
JUL    71.18   276.93  ¯326.43   ¯67.20   328.26
OCT  ¯190.04   188.87   106.11   392.83  ¯202.37
```

There are obvious restrictions on the left argument of format, because the width of a field must be large enough to hold the requested form; and if the specified width is inadequate, the result will be a *DOMAIN ERROR* (unless ⎕FC[4]≠'0' ...see the description of "Overflow Control" in the reference table on page 216). However, the width need not provide open spaces between adjacent numbers. For example, boolean arrays can be tightly packed:

```
      1 0⍕2=?4 4⍴2
1001
0000
1101
0111
```

Picture Format

One of the most common requirements of business data processing is formatting data for reports. This has sometimes been a difficult task, with the output often lacking the decorators that were desired for a truly readable report. How many of you, for example, have taken the trouble to insert commas into large numbers for readability? "`454217329`" becomes much more understandable as "`454,217,329`", but as nice as that might be on the final report, formatting it that way used to be a formidable task. With picture format (also known as "format-by-example"), such tasks become trivial.

The picture format primitive shares the same symbol with the other formatting primitives: `⍕`. But as well as using a *numeric* left argument, (`5 0⍕`*M*), format may also use a *character* left argument (`' 55,555.00'⍕`*M*). When the left argument is a character string, the function is picture format.

Picture format provides an *easy* method for you to:

- Print numeric output with controlled commas: `16,777,215`
- Use any "negative" indicator that you wish, in case the APL "`¯`" symbol isn't available with the printer or typeball that you want to print your report on; use "`-`", "*CREDIT*", or whatever you want
- Optionally suppress fields that represent values of `0`, so that they print as blank fields
- Print values with leading or trailing zeros
- Float a decorator, such as a dollar sign, in against your data
- Print numeric values in European notation, with a comma separating the integer and decimal portions of a number: `12,34`
- Display negative numbers within parentheses, as on accounting reports: `(12.50)`

...or, well, you name it.

Picture format can often make short work of what had previously been complex formatting jobs.

There are, of course, very specific rules for the format of the left argument... but we'll come back to that. For now, let's just say that the left argument (or "pattern") shows APL a sample "picture" of what we want the results to look like.

We should point out that the *right argument* for picture format must be a simple all-numeric array; it may not contain character data, and it may not be nested.

Using picture format, we can do a lot of formatting with a minimal amount of programming. Let's say that we want to build a report function that will dress up the output for us. Here's a sample function that could do the formatting for us. Again, don't be concerned right now with just what the rules are for coding the left argument for picture format... we'll get to that in a bit. For right now, simply notice how compact the actual formatting is when we use picture format:

```
    ∇ Z←REPORT M
[1]  ⍝FORMAT ALL THE DATA, ADD DOLLAR SIGNS AND
[2]  ⍝   'CR' NOTES, AND ADD TOTAL LINE:
[3]   Z←'   $35,555.19_CR'⍕M,[1]+⌿M

[4]  ⍝ADD COLUMN HEADINGS:
[5]   Z←'       ITEM 1          ITEM 2    ',[1] Z

[6]  ⍝ADD ROW MARKINGS:
[7]   Z←(5 7⍴'GROUP  DEPT A:DEPT B:DEPT C:TOTAL: '),Z

[8]  ⍝INSERT BLANK LINES BETWEEN HEADING AND BODY,
[9]  ⍝   AND BETWEEN BODY AND TOTAL:
[10]  Z← 1 0 1 1 1 0 1 ⍀Z
    ∇
```

Here's the data that we're working with:

```
      ⍴DATA
3 2

      DATA
12345.67     1
   34.15 ¯1234.56
  227.5      0
```

Running the "*REPORT*" function, we get this finished report:

```
      REPORT DATA
GROUP         ITEM 1          ITEM 2

DEPT A:   $12,345.67           $1.00
DEPT B:       $34.15       $1,234.56 CR
DEPT C:      $227.50

TOTAL:    $12,607.32       $1,233.56 CR
```

* * *

One of the problems that you may have experienced in the past is specifying the proper numeric left argument for the format primitive in such a way that the output properly lines up with your column headings. With picture format, the length of the left argument is the length of the result, so this becomes a good deal simpler:

```
    ∇ FMT V
[1]  ⍝PROBLEM:  WHEN USING FORMAT (⍕),
[2]  ⍝   IT'S DIFFICULT TO ALIGN COLUMNS:
[3]   □←'   USERS  CONNECT  COMPUTE  WORKSPACES'
[4]   □← 8 0 9 0 9 1 12 0 ⍕V
[5]   □←''

[6]  ⍝SOLUTION:  PICTURE FORMAT ALLOWS HEADINGS
[7]  ⍝   AND FORMAT CONTROL TO ALIGN:
[8]   □←'   USERS  CONNECT  COMPUTE  WORKSPACES'
[9]   □←'  55,555   55,555    555.0     555,555'⍕V
[10]  □←''
    ∇

      FMT V
   USERS  CONNECT  COMPUTE  WORKSPACES
    3513    11173    107.3       33658

   USERS  CONNECT  COMPUTE  WORKSPACES
   3,513   11,173    107.3      33,658
```

As we mentioned, the length of the result from picture format is the same as the length of its left argument (except for the case where picture format contains just one field, which will then apply to each column of data). Similarly, the positions of such things as commas and decimal points will match the position of these items in the output.

Fields within the left argument are typically separated by blanks (although we'll see a way to let other characters separate the fields, too). The number of fields in the left argument must match the last dimension of the data being formatted, although if there is only a single field, that's acceptable too... it will be used for every column of data. So then, what's a field? A field is a sequence of characters bounded by blanks (or the end of the pattern) containing at least one digit. If it doesn't have any digits, it's a decoration. That's allowed, too; this example has two fields:

```
      '| DATA 55 55 |'⍕2 2⍴10 20 30 40
| DATA 10 20 |
| DATA 30 40 |
```

The vertical bars and the word "*DATA*" aren't fields, because they don't contain any digits; they therefore become simple decorators.

Here's an application of the previous example:

```
      ∇ Z←PART V
   [1]  Z←'COST OF PART-NUMBER 5555:    $ 5.50/BOX'⍕V
      ∇

        PART 2117 4.25
   COST OF PART-NUMBER 2117:    $ 4.25/BOX
```

Consider how you would have to do this *without* the assistance of picture format:

```
      ∇ Z←PART1 V
   [1]  Z←'COST OF PART-NUMBER ',(⍕V[1]),':    $ ',
        (4 2 ⍕V[2]),'/BOX'
      ∇

        PART1 2117 4.25
   COST OF PART-NUMBER 2117:    $ 4.25/BOX
```

Digits in the pattern serve a dual purpose: they show where digits may appear in the result, but further, specific digits in the left argument have specific meanings regarding the formatting that is to take place. They are called "distinguished digits," and a table of them follows shortly.

Nonnumeric characters in the left argument can be:

- ***Simple*** decorators (like the example just shown)
- ***Controlled*** decorators (such as a comma, which appears or is suppressed according to established conventions)
- ***Floating*** decorators (such as a dollar sign, which can be made to nestle in against the left side of the data). The action of these floating decorators is also controlled by selecting which of the "distinguished digits" you use.

Further control is provided through the use of a system variable, ⎕*FC* (format control). This variable acts as another (implicit) argument for picture format. A table explaining its operation is also following.

This page has been intentionally left blank so that the following table is on facing pages.

Distinguished Digits for Picture Format

(In general, use 5's for the pattern except where special handling is desired, as noted in the table.)

1

Float the decorator in against the number *only* if the value is negative. (See notes with digits 1 and 3.)

```
      ' -551.50'⍕ ¯1 10 ¯100
   -1.00   10.00 -100.00

      ' 551.50-'⍕ ¯1 10 ¯100
   1.00-  10.00  100.00-

      ' (55,551)'⍕ ¯10000 ¯1 10 ¯100
 (10,000)      (1)      10     (100)
```

Note that it's up to you to provide a "negative" indicator that's appropriate to your own application. Picture format provides the capability of using any sign that you wish ("¯", "-", "*CR*", or whatever else you may want to choose). If this is *not* done (that is, if the pattern doesn't include "1"s or "2"s), a *DOMAIN ERROR* will result (see ⎕*FC*[4] in the "Description of ⎕*FC*" to override this).

2

Float the decorator in against the number *only* if the value is *non*negative. (See note with digit 3.)

```
      ' +552.50'⍕ ¯1 10 ¯100
    1.00  +10.00  100.00
```

3

Float the decorator in against the number for *all* values (positive or negative). (See note below.)

```
      ' $553.50'⍕ 1 10 100
   $1.00  $10.00 $100.00

      ' $553.10-'⍕ ¯1 10 ¯100
   $1.00-  $10.00  $100.00-

      ' $553.10CR'⍕ ¯1 10 ¯100
   $1.00CR  $10.00   $100.00CR
```

If only one of distinguished digits 1, 2, or 3 appears within a given pattern, its effect applies to both right and left floating decorations. If more than one appears, each one affects its own side.

4

Counteracts the action of a 1, 2, or 3, preventing it from affecting the other side of the field, which is then treated as a simple decorator.

```
      ' -551.40CR'⍕ ¯1 10 ¯100
   -1.00CR   10.00CR -100.00CR
```

5	Perform normal formatting, observing normal APL rules of removing leading and trailing zeros, except that a value of zero will display as all blank. Be careful, though; it's up to you to include an appropriate sign character if you expect any negative values, and a "1" or "2" to control it. See the note with digit 3.
	`  '  555.55'⍕1.1 10.01 100 0 100.10`
	`1.1    10.01  100             100.1`

6	The decorator to the right also marks the end of this field; treat it as though there's a blank between the fields, but also print the decorator.
	`'05/55/55'⍕32580 03/25/80`
	`'06/06/05'⍕ 3 25 80 03/25/80`

7	The next nonnumeric character to the right is the symbol to be used for exponential notation ("*E*-format").
	`   '   ¯1.70E¯01'⍕ ¯.001 100 10000 ¯10000000000`
	`¯1.00E¯03    1.00E 02    1.00E 04   ¯1.00E 10`
	`   '   ¯1.70*¯10'⍕ ¯.001 100 10000 ¯10000000000`
	`¯1.00* ¯3    1.00*  2    1.00*  4   ¯1.00* 10`

8	"Check-protection": fill empty portions of the field with whatever character is in ⎕*FC*[3] (in origin 1). The default character is *.
	`     ' 8555.50'⍕ 1 10 100`
	`***1.00 **10.00 *100.00`
	`     ' 5855.50'⍕ 1 10 100`
	` **1.00  *10.00  100.00`

9	Pad with leading or trailing zeros out to this point (before or after the decimal point, respectively), unless the value is zero (then use all blanks). (Compare with 0.)
	`  '  555.59'⍕ 1.1 10.01 0 100`
	`1.10   10.01          100.00`
	`  '  555.50'⍕ 1.1 10.01 0 100`
	`1.10   10.01     .00  100.00`

0	Pad with leading or trailing zeros out to this point (before or after the decimal point, respectively). (Compare with 9.)
	`    '  055.50'⍕ 1 10 100`
	`001.00  010.00  100.00`

Description of ⎕*FC* — Format Control

Default setting: ⎕*FC*←'.,*0_¯'

⎕*FC*[1]	.	Decimal point: the character that's to be substituted for the period where a decimal point is required in the result. This also affects numeric-left-argument format.
		`      ⎕FC[1]←'.'`
		`      '55.55'⍕12.34`
		`12.34`
		`      5 2⍕12.34`
		`12.34`
		`      ⎕FC[1]←','`
		`      '55.55'⍕12.34`
		`12,34`
		`      5 2⍕12.34`
		`12,34`

⎕*FC*[2]	,	Comma: the character to be substituted for the comma where a controlled comma is required in the result.
		`      ⎕FC[2]←','`
		`      '5,555'⍕1234`
		`1,234`
		`      ⎕FC[2]←' '`
		`      '5,555'⍕1234`
		`1 234`

⎕*FC*[3]	*	Check Protection character: the character to be printed in response to the "8"'s in the pattern.
		`      ⎕FC[3]←'*'`
		`      ' 855'⍕1 10 100`
		` **1 *10 100`
		`      ⎕FC[3]←'/'`
		`      ' 855'⍕1 10 100`
		` //1 /10 100`

⎕*FC*[4]	0	Overflow control: If the default character appears here, a value which is too large to fit into a field specified will cause a *DOMAIN ERROR*. If any other character appears here, the error will not occur and instead the offending field will be filled with the character specified. This also affects numeric-left-argument format.
		`      ⎕FC[4]←'0'`
		`      ' 00'⍕1 10 100`
		`DOMAIN ERROR`
		`      ' 00'⍕ 1 10 100`
		`      ^     ^`
		`      ⎕FC[4]←'?'`
		`      ' 00'⍕1 10 100`
		`01 10 ??`
		`      3 0⍕10000`
		`???`

⎕*FC*[5]	_	"Print-as-blank" character: any place that this character appears in the pattern, it will print as a blank; it functions normally in the analysis of the pattern, but is replaced by a blank in the result. In ⎕*FC*, it may not be a blank, period, comma, or a digit.
		`      ' $_35,555'⍕12345 250 5000`
		` $ 12,345     $ 250   $ 5,000`
		`      ' 15,555_CR'⍕¯12345 250 ¯5000`
		` 12,345 CR     250      5,000 CR`

⎕*FC*[6]	¯	Negative number indicator. This character is used as the negative sign when you use format by specification. It has no effect when you use picture format.
		`      3 0⍕-⍳12`
		`¯1 ¯2 ¯3 ¯4 ¯5 ¯6 ¯7 ¯8 ¯9¯10¯11¯12`
		`      ⎕FC[6]←'-'`
		`      3 0⍕-⍳12`
		`-1 -2 -3 -4 -5 -6 -7 -8 -9-10-11-12`

While ⎕*FC* currently contains six characters, it is recommended that you don't consider its length to be fixed. Future extensions could add additional elements.

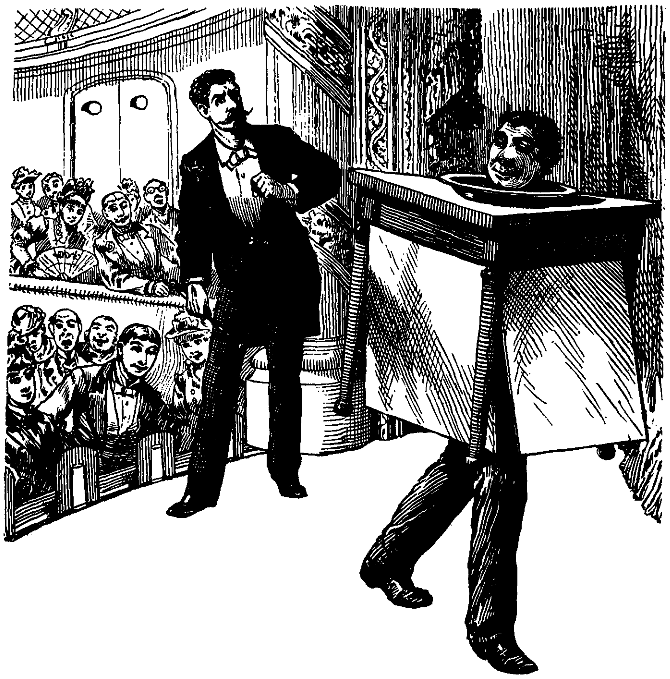
The only valid current configuration for ⎕*FC* is six character elements. Any setting other than this will cause a ⎕*FC ERROR* to be evoked when any use of dyadic format is attempted.

Note that the first two elements show what characters are to be printed where the decimal point and controlled comma are required in the result. The pattern is always coded using U.S. conventions; ⎕*FC* can be changed to allow display of British or other standards. This requirement for the pattern was done to allow an easy transfer of programs between countries... a change to all the patterns in the workspace isn't needed for such a move — only a simple change to ⎕*FC*.

Chapter 6: Our Own Biased Views of Programming

Congratulations! You've read the rest of the manual [let's assume]; now you know what many of the pieces do. But how do you put them together into a usable application? Where do you even *begin*??

Fair enough. You begin here. This chapter describes some of the pitfalls that are common to APL programming — in fact to programming in almost *any* language — and some ways in which you can avoid the problems. These thoughts are very subjective, and really only represent *opinions*. They are *not "standards"*. And by *no* means are they meant to be a statement of "*the* proper way of doing things"; they are simply some "tricks of the trade" which have proven to be helpful to us. Maybe they can help you, too.

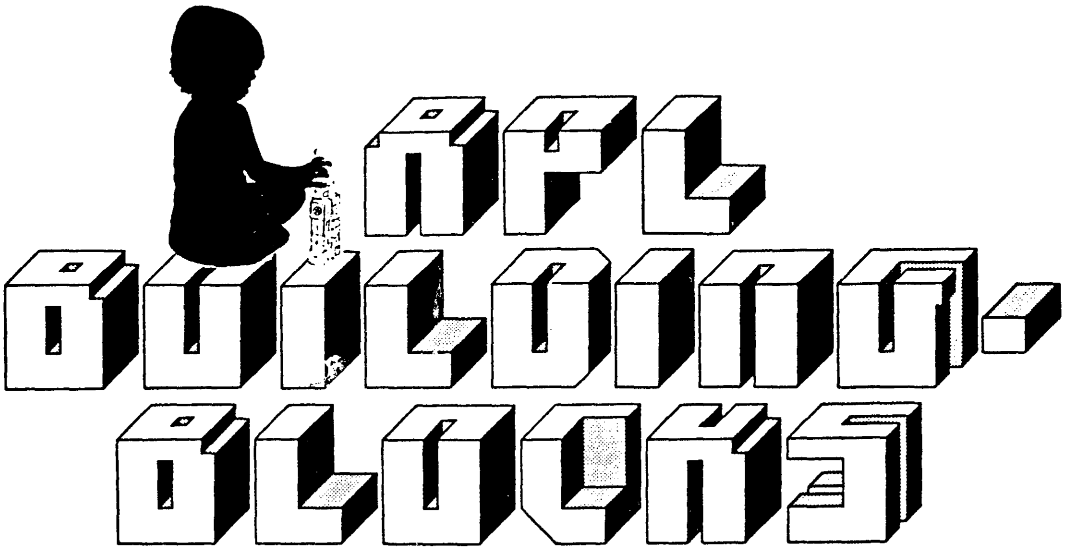


Used with the permission of Hart Publishing Company, Inc.

Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke
Profiles of the Future,
Harper & Row, 1962

Some Thoughts on Programming Style



What would it be worth to you to find a way to write functions more quickly and easily, make trouble-shooting considerably faster, aid documentation, *and* reduce *WS FULL* problems, all with one technique? ...Read on, Macduff....

A "building-block," or "modular," approach to writing APL functions is a writing style in which the main function calls sub-functions to perform the details of the work, rather than all of the code being in the main function. Developing an APL writing style like this can result in some rather substantial benefits for you.

Answering lots of phone calls from our customers as I do, I see quite a bit of programming written by people with a wide variety of backgrounds and approaches. One distressing sight that I often see is a workspace containing an extremely lengthy function with no subroutines. Now, I realize that most of the users of most APL systems aren't programmers, but it's sad to see APL functions like this because substantial savings could have been realized both in original writing time and in subsequent troubleshooting if the author had used a building-block approach in writing his application.

Probably each of you have seen examples like the ones here; probably many of you also have some similar ones of your own. (I have lots of them, but I'm fixing them little by little.) First, take the case of this portion of a billing function that I once worked with:

```
[65] PRTOT1:Y← 0 60 60 ⊤TT[4;0]
[66]  Z← 0 60 60 ⊤(TT[4;1]×3.33)÷1000
[67]  CON←(4 0 ⍕Y[0]),':',2 0 ⍕Y[1]
[68]  →(0=∨/S←CON[¯6↑⍳7]=' ')/L20
[69]  CON[S/¯6↑⍳7]←'0'
[70] L20:CPU←(3 0 ⍕Z[0]),':',(2 0⍕Z[1]),':',2 0⍕Z[2]
[71]  →(0=∨/S←CPU[¯8↑⍳9]=' ')/L21
[72]  CPU[S/¯8↑⍳9]←'0'
[73] L21:OUTPUT←TEMP[CHG],' ',CON,' ',CPU,⍕TT[4;REST]
[74]  PRINT OUTPUT
```

Now don't get me wrong; this function does work. ...It's just a little obscure. It's an example of opacity in coding (perhaps the real meaning of "coding"). However, a reader of the function doesn't normally need to know minute detail of any portion of the code until he identifies the portion that needs repairs. This can easily be achieved by simply enclosing pieces of the function in subfunctions, whose contents are unimportant until we need to see their own particular piece of the operation:

```
[21] PRTOT1:CON←FORMAT RATES[0]×TT[4;0]
[22]  CPU←FORMAT RATES[1]×TT[4;1]
[23]  PRINT TEMP[CHG],' ',CON,' ',CPU,⍕TT[4;REST]
```

Notice that this function is notably shorter than the previous one. The original one was 118 lines long (ugh!). How long *should* a function be? Well, Abraham Lincoln observed that a person's legs are the right length when they just touch the ground. In the same spirit, there's no particular rule that says how long a function can be. But if you properly observe building-block techniques, you'll probably find that your average function length is fairly short.

And what's that? I checked my own workspaces, and here's the tally: distributed through 12,366 unlocked functions were 145,090 lines of code, giving an overall average of 11.7 lines per function (counting comment lines). The average width, by the way (excluding comment lines) was 19.2 characters.

Keeping the average to under twenty lines ensures that you will be able to display and edit the function much more easily on a 3270 display terminal, and you'll save a lot of printing time on a typewriter-type terminal. Also as a rule of thumb, any one single function *should always* fit on an 8.5×11-inch sheet of paper when it's printed. Of course, Lincoln *also* observed that "important principles may and must be flexible," but if an APL function is longer than a page, you're probably trying to accomplish too much with one function. See the discussion of abstracts, "Where's This Function Going?," on pages 228-229, for more on this.

Save Yourself Writing Time

This benefit comes in several flavors. First, a short function is very easy to display and edit. Imagine having to repeatedly display that 118 line function during its development. Second, there is a finite amount of material that you can concentrate on at one time. Keep a given function focused on a specific goal. In that way, *you* can also concentrate on that same goal without being drowned in extraneous details. And third, once you identify the specific purpose of the function, you may not even have to write it at all! What's that you say?? Very simple....

Become a Friend of the Public Library

Take advantage of the voluminous resources of the APL Public Libraries that are available on many APL systems. If you're trying to accomplish some very standard operation, why reinvent the wheel? While perhaps no one else has ever written a package just like the one that you're working on, certainly *many* of the building blocks will be the same. You may have designed your own very unique house, but chances are you didn't have to design the bricks, lumber, nails, and wiring components. Why do that with your APL code?

The classic analogy to making your own nails is writing a lengthy *REPORT* function which spends several lines, labels, and branches just formatting the current time and date to print on the report. What a waste of time for an author to spend his good time figuring that stuff out, when his real goal is to produce a specific report. Does he assume that he's the first one who has had to timestamp a report? Building blocks like this are available in abundance in the Public Libraries on many APL systems, free for the taking.

Simplify Your Troubleshooting

Debugging any program can become a tedious task; anything that helps to lessen that task is a welcome treat. A building-block approach immediately reduces the task. In the aforementioned billing function, let's suppose that the connect and CPU times were printing improperly. In the first example, you may have had to print over seventy lines of code to find the code that was causing the problem, and even then it would have required making the same change to several sections of code. Using building blocks (and assuming that you have used obvious names for the blocks), you or anyone else should be able to quickly locate the culprit. With the subfunctions in the second example, changing the *FORMAT* function (which is perhaps five lines long) could solve the problem; you wouldn't have to be concerned with the main function at all. And changing that one function would fix the problem wherever the function is called.

You'll find that a modular approach to functions will immediately put at your disposal the considerable resources of (again) the Public Library for debugging aids. Some Public Libraries contain applications which will analyze the interaction of functions without being very concerned with what's inside those blocks.

Simplify The Job of Documenting Your Application

Documentation is usually considered to be most programmer's anathema. There are several forms of documentation. One is in the form of a formal manual. Another is comments within the code. Another is the code itself. Compare the two examples that we have discussed so far. Certainly the second function would be the easier one for a reader of its user's guide to correlate to the text. The number of comments required in the code is drastically reduced if appropriately named subfunctions are employed, since the code itself becomes much more self-documenting.

Prevent *WS FULL* Problems with Building Blocks

The first example in this discussion of building blocks shows some identical lines of code being repeated. Everything that gets entered in the function uses space that can't be used for something else. Using a subfunction allows one copy of the code to be used several places in the workspace without taking several times its storage space. And, by properly localizing variables you don't have to take any special care to expunge variables that you're through using; they'll go away on their own, thereby freeing up the space that they were consuming.

Considering the rather substantial benefits that can be realized by using a building-block approach, it's really a shame that more people aren't taking advantage of it. Don't cheat yourself; give it a try. Start by making a conscious effort to use the technique. Chances are, you'll soon have altered your writing style to the point where you'll wonder why you didn't always write that way.

Some Considerations in Writing Building Blocks

A building block should represent a single isolated operation. It should be something that can be commonly used, usually without any modification, by many applications. For example, a *REPORT* function that reads the contents of one of your datasets and breaks your data down into subtotals is *not* a building block. The function that opens the file *could* be a building block, if sufficient thought is given to it so that it can also be used elsewhere. The same could be said of the function that reads each block of data from the file, or of the function that checks the return code from a file processor. There's no reason to burden the main calling function with specialized code to do this sort of thing when you're going to have to do the same thing with *many* applications. Write it one time, and be done with it.

So, how can a function be generalized? Well, to be considered a building block, a function almost certainly has to use arguments and an explicit result. Refer back to pages 93-95 if you're hazy on how to use these properly. If a function doesn't have arguments, where is it getting its data? The use of global variables makes for very obscure applications. And if the function doesn't have an explicit result, it can't pass data along to another function for continued processing.

An example of a real ground level building-block function (a real *nail*) is a function that we'll call *DMB*. This function is designed to "delete multiple blanks," by dropping all leading and trailing blanks, and reducing multiple contiguous blanks within a character string to single blanks.

```
      ∇ Z←DMB V            ⍝ DELETE MULTIPLE BLANKS
[1]    ⍝...REMOVES LEAD, TRAIL, REDUNDANT BLANKS
[2]     Z←1↓¯1↓(~'  '⍷Z)/Z←,' ',V,' '
      ∇

        L←'    THIS    IS   IT  '

        ⍴L
21

        DMB L
THIS IS IT

        ⍴DMB L
10
```

This function is so generalized that it can be considered to be almost a primitive; it just happens to be written in APL. This function is very useful when you wish to pull a line out of a character matrix and use it elsewhere; *DMB* lops off all of the unneeded blanks. It is also quite helpful when prompting for input with ⍞. Using *DMB* means that you don't have to be concerned if you get back a few extra blanks along with the response (as may happen with bare output, for example).

Another tiny building block would be a function called *THRU*. (For example, `3 THRU 7` yields `3 4 5 6 7`.) It's obviously not related to any particular application; it's just a useful tool for saving yourself a little bit of effort.

Most of the building blocks that you put together may well be more complex, and therefore more specialized, than these examples. But remember that the more generalized you can make the function, the higher the probability is that you will be able to use it again in another application.

You will undoubtedly find that some things that you use building blocks for early in your APL career may later on be discarded in favor of just entering their definitions directly. For example, where you used to use a function for right-justifying a matrix of text...

```
      M←RIGHT_JUSTIFY M
```

...you might replace it later with its definition:

```
      M←(-+/∧\⌽' '=M)⌽M
```

"What gives? I thought you just tried to talk me into using building blocks?" True, true. But building blocks may *also* serve the purpose of providing education. If you haven't had to left- or right-justify a matrix of text before, having a small stand-alone function to experiment with is certainly much more convenient than trying to observe that one line of code within a larger application. If that expression is something that you use several places within one workspace (as would typically be the case with *DMB*), then it should remain as a separate building block. If it's used only once in the workspace, but you don't know how to code it from memory, then a building block would also make sense. But if it's only going to be used once in the workspace, if you know how to code it, and (as with the above example) if the actual working code is no longer than its name — it may be just as easy to code it directly. These short expressions that you may find yourself using repeatedly are referred to as "APL idioms."

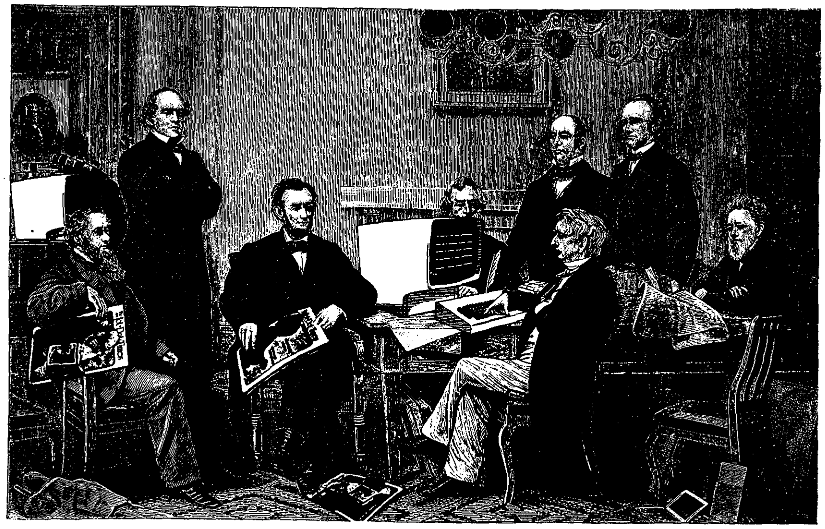
* * *

Occasionally we see an application that has a set of defined functions that look like this:

```
      ∇ Z←A PLUS B
[1]     Z←A+B
      ∇
```

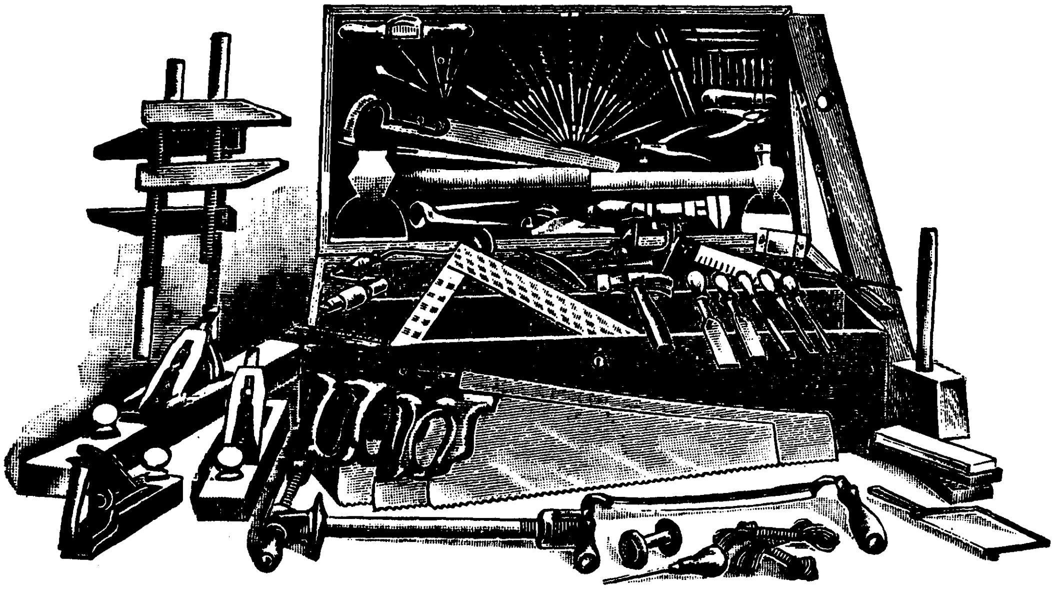
If the authors of these functions are using them as textbook examples of how defined functions work, that's fine, but if they're actually trying to run these functions in an application, they have possibly missed the point of modular approaches... they aren't making the overall function any easier to read or maintain, and they aren't saving space by reducing multiple definitions. Don't break things down this far.

Precisely where the divisions should occur, we can't say. We can give some examples, but there are no rules. A big factor in how far you break them down is your degree of familiarity with APL. The discussion entitled "Putting It All Together" on pages 238-243 gives some examples of a good approach to one project.



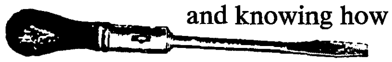
Used with the permission of The Dick Sutphen Studio.

How to Build a Toolbox

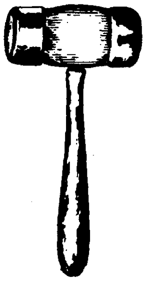


Used with the permission of Hart Publishing Company, Inc.

Have you ever seen a really devoted home handyman with an extensively-equipped woodwork shop? These guys seem to be able to whip up a beautifully-made walnut coffee table in the time that it would take me to gouge out my own specialty (an ash tray), using my trusty (rusty) combination screwdriver/chisel. So how do they do it? Obviously, part of it is careful training and skill. But an equally important element is having a well-equipped toolbox, knowing where all of these tools are, and knowing how to use each of them the right way.



The same idea holds true with APL programming, or with nearly any undertaking. If you want to build big complex packages quickly, you need to have a well-equipped toolbox. In APL terms, that means having access to workspaces that contain some basic building blocks. These blocks can then be used repeatedly as elements of many larger applications.



When you first get into APL programming, things are very slow; you're often left with the feeling that in order to do anything, you have to concern yourself with all of the details of the universe. But as time goes by, you may realize that many of the components of a new application that you're working on are really identical to the same blocks in your last application. And so, those don't need to be written. The longer you work with it, the more building blocks you have at your disposal; hence, the faster, easier, and more pleasurable the work becomes... more pleasurable because you can concentrate your efforts on the truly creative aspects of each new job, rather than being buried by the drudgery of low-level details.

In the discussion of "Building Blocks" (on pages 221-226), we recommended that you gather subroutines from the Public Library rather than writing your own. Where else can building blocks come from? Well, many of them you will undoubtedly write yourself. Others will come from friends, or from examples in textbooks. And that brings us to our next topic.

If you're not already taking advantage of the abundant resources of the APL Public Libraries, you're probably doing things the hard way. The Public Library represents an impressive storehouse of knowledge on a wide variety of subjects.

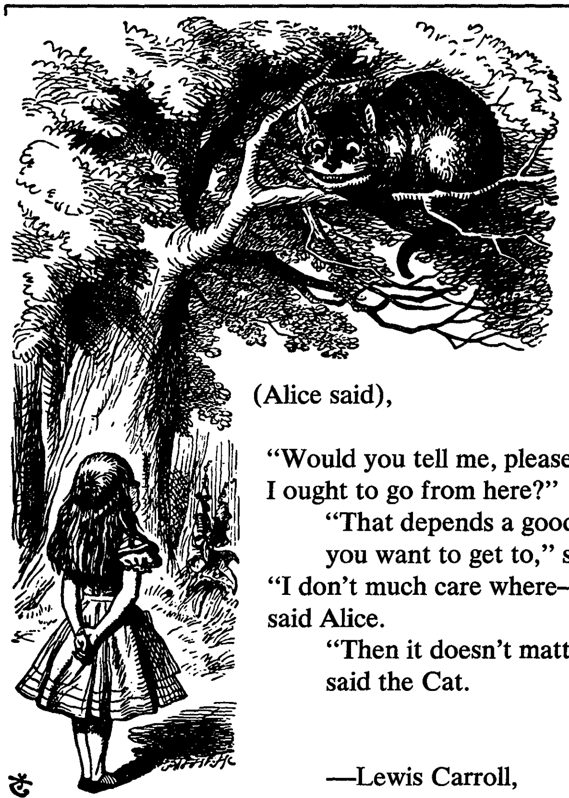
Whether you need complete packages (like file access systems), or building blocks (like time and date functions), try the library.

When you're writing an application, your goal is the overall operation of that package. You shouldn't have to spend the time concerning yourself with minute details that have certainly been solved before by others. By using Public Library functions, not only are you saving yourself from re-inventing the wheel, but you are using building blocks that are often better than the ones that you would have been able to build yourself. Keep in mind that the author of a workspace containing building blocks had *those blocks* as his goals, and therefore probably had the time to consider (and solve) many problems that you might not have thought of.

Where's This Function Going?

In the discussion of creating "building block" applications (on pages 221-226), we made it sound like all you had to do was put things in small packages and you'd be all set. Well, there's one obvious problem that arises. How do you keep track of all of those subroutines?

But hold on; there's a simple solution (did you *really* think we were going to say there isn't any solution?).



(Alice said),

"Would you tell me, please, which way
I ought to go from here?"
"That depends a good deal on where
you want to get to," said the Cat.
"I don't much care where—,"
said Alice.
"Then it doesn't matter which way you go,"
said the Cat.

—Lewis Carroll,
"Alice's Adventures in Wonderland"

Used with the permission of MacMillan Publishers, Ltd.

It may seem obvious that you should decide what each APL function is supposed to do *before* you write it, but this fundamental seems to get overlooked rather often. We've all seen examples of functions that just seem to ramble about aimlessly, without any discernible purpose in life. Roget's Thesaurus shows "purpose" as a suitable synonym for "function" (of course, realize that it also shows "gathering," and *that* one may be more suitable for some of the functions that we've seen). Do each of your own APL functions have a single specific purpose?[17]

When you begin to write a function, the first thing that you should do is decide what *one* particular operation you wish to perform, and document that with a comment line in the beginning of the function *before* you write any code. This line is called an "abstract" (and hopefully prevents the rest of the function from becoming the abstract portion). The abstract line shouldn't ever exceed 60-80 characters. You may find that you can't state the function's purpose in just one line. *Stop!* Don't just write a longer one! That's probably a good indication that you're trying to do too much with that function. Narrow its scope a bit and get it down to one operation before you start to write *any* code.

After you are into the writing of the function, this comment line will give you a good reference point to be sure that you aren't wandering from your purpose. You'll be surprised how much time you can save in writing the function by taking a minute to consider the scope of the function before you start, and clear your mind of any side purposes. Leave them for the next function.

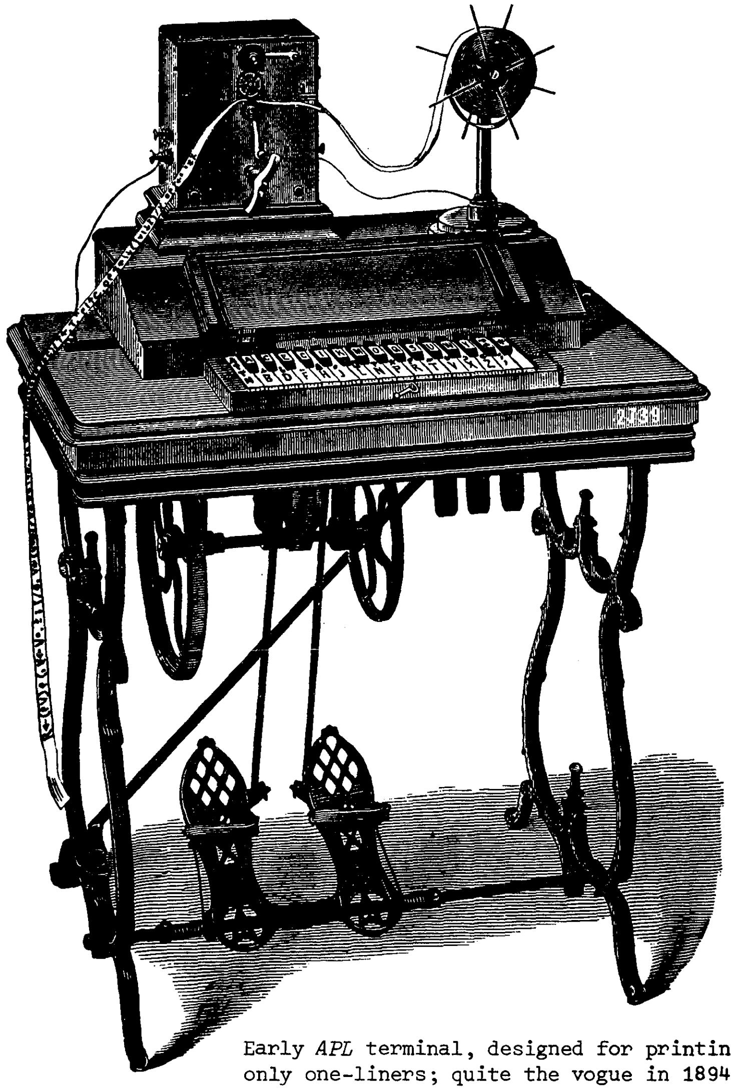
The next advantage that comes from this comment line is that the person who has to maintain the application will be able to understand the purpose of the function by reading just one line, even though the rest of it may be completely obscure to him. And remember, that person who has to go back and modify this strange beast two years from now *may* be *you.*

If the particular algorithm that's being used to meet the stated purpose isn't clear, or seems undesirable, the person charged with the maintenance has the option of simply replacing the module with a functionally equivalent one which is more obvious or runs faster.

Let's assume that you're given the job of maintaining an APL application which you've never seen before. How do you learn what it's composed of, and what each of the building blocks do? If each function contains a one-line abstract, it's rather easy to write a function that simply displays each of the function names along with its corresponding description. In this way, the workspace could be made to explain itself.

[17] Refer also to the discussion of "Building Blocks" (on pages 221-226) for a discussion of creating a connected series of "black-boxes."

The One-Liner Syndrome



Early *APL* terminal, designed for printing only one-liners; quite the vogue in 1894.

Used with the permission of The Dick Sutphen Studio.

In APL, a line of code isn't restricted to having just one or a small number of primitives; it can be of *any* arbitrary length. Indeed, if one so desired, a championship chess program could be written in one line. However, it is appropriate to mention at this point that **just because something *can* be done doesn't mean that it *should* be done.**

One of the biggest criticism that APL receives is that it is often obtuse, opaque, and just plain hard to follow. Unfortunately, that's often true. But the fault isn't with APL, it's with the "clever" [*sic*] programmer who shows how much he can pack into one line.

The One-Liner Syndrome is an affliction that seems to hit every APLer at some time in their APL career... I'm not sure why. I certainly wasn't bypassed; everything that I wrote used to be packed into very few lines, all of which were hundred-character-long twisted barbed-wire tangles of circles, slashes, and stars, looking like the wrath of Zeus bolting from the heavens, or perhaps like that crack in the living room wall. The one thing that they *didn't* look like was readable code.

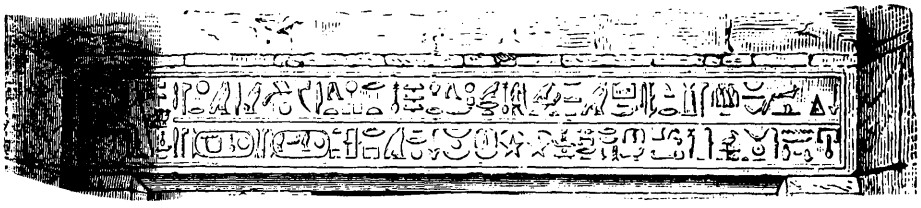
Over the years I slowly came to my senses, after noticing all of the obstacles that I was throwing into my own path. The most obvious problem is that such functions can't be understood without extensive study. This should be reason enough to stop using them, since the author most often has to maintain his own code. But there are other arguments against one-liners.

One-liners can't be meaningfully traced, using the "*T*∆-name" facility.[18] Trace shows only the last (leftmost) result of each line, and when the entire function is on one line, that's not too meaningful.

Since a one-liner normally has to hold *many* temporary results in intermediate storage while it's parsing the rest of the line, one-liners are much more susceptible to *WS FULL*s than other (more reasonable) functions. Many is the time that I couldn't get past a *WS FULL*, until I finally bit the bullet and broke the offending line into several shorter lines.

This brings up the next problem: one-liners are just plain hard to edit. Those one-liners frequently exceed the page or screen width. True, that doesn't necessarily mean that you *can't* edit them, but it does generally make it more cumbersome. Why put yourself through that?

First Known Example of One-Liners



Used with the permission of The Dick Sutphen Studio.

Everything that we've discussed so far has had to do with *programmer* efficiency, not *program* efficiency. One of the arguments that I have heard in *favor* of one-liners is that such a function will run faster. I always wonder if the proponents of this school of thought have actually tried any timing tests. I urge you to do so. One-liners aren't by nature faster *or* slower, but in actual practice, they are normally very much slower. One of the reasons for this is what I shall call "fake catenation" (a contrived phrase, but I think it communicates).

"Fake catenation" is an artificial construction that is employed as a trick simply to glue two lines together into one line. It should be avoided like the plague. For example, let's assume that we wish to assign a value of 0 to both *A* and *B*. One method would be to have two lines, *A*←0 and *B*←0. There is nothing wrong, though, with *A*←*B*←0. The computer doesn't have to do any extra work (and neither does the programmer when he reads the code six months later). However, let's now assume that we wish to assign *A*←0 and *B*←1. My preferred method would be two lines, stated just as they are in this statement of the problem. But a construction that I see all too often is *B*←1+*A*←0. Notice that this time APL has to do an extra addition that it wouldn't otherwise have to do; the addition isn't part of any of the productive work that the function is trying to accomplish. Of course,

[18] "*T*∆-name" is "trace control," causing the final value of each line in a defined function or operator to be displayed at the terminal. Stop and Trace are both described in *APL2 Programming: Language Reference.*

the time to do an addition is extremely small, but then so is the time to go to a new line.

A larger problem associated with the above example is in readability. Sure, that last example isn't *too* bad, but let's suppose now that you were going to set *A*←19 and *B*←457. If that's stated in one line (and I often see examples like this), you have your choice of *B*←438+*A*←19 or *A*←¯438+*B*←457. Now, try to read *that* and know what the values of the two variables are. To do so you have to mentally perform those same meaningless calculations that shouldn't be there in the first place. In an effort to prevent this problem, many programmers turn to the next (and more devastating) trick.

Using that last example, where we want *A*←19 and *B*←457, many programmers just automatically enter either "*A*←19,0/*B*←457" or "*A*←19,0ρ*B*←457". In each case, the value is assigned to *B*, then that value is reshaped in a null vector, catenated onto another value (19), and assigned to *A*. This is the real essence of fake catenation. One would expect that (other than the computer doing some extra work) this would be pretty much the same as entering those two statements on two lines; in particular, the resultant values in *A* and *B* should still end up being the same. ...'taint so, McGee.

Several times I have gotten calls from people who claim to have mysterious APL errors that just seemed to spring up out of the woodwork, when "*Not Much of Anything*" had been changed in the code. Well, picture the following two cases:

```
      ⎕CR¨ 'FN1' 'FN2'
FN1 M;C;H;L       FN2 M;C;H;L
C←0               ⍝HERE'S TROUBLE...
H←1↑ρM            C←0,0ρH←1↑ρM ◄──── set up variables
LOOP:C←C+1        LOOP:C←C+1 ◄────── increment counter
→(C>H)/0          →(C>H)/0 ◄──────── check for the end
L←M[C;]           L←M[C;] ◄───────── get the next line
'THIS IS ',L      'THIS IS ',L ◄──── print the line
→LOOP             →LOOP ◄─────────── go back to loop
```

Well, the differences seem innocuous enough. And except for a slight case of fake catenation in *FN*2, the two functions appear to be identical. ...Oh, yes, there is one other difference: *FN*2 doesn't work....

```
          FN2 MATRIX
    LENGTH ERROR
    FN2[5] 'THIS IS ',L
            ^          ^
```

This problem is due entirely to the fake catenation, but that may not be at all apparent to the person who has to fix it at this point.

In *FN*1, with *C* assigned on a separate line, *C* took on a *scalar* (dimensionless) value of 0. In *FN*2, ",0ρ" caused the value that was put into *C* to be a *one-element vector*. Throughout the operation of these functions, *C* retains its given dimension. On the line containing *L*←*M*[*C*;], *C* is used as a subscript for a matrix. The shape of the result of any subscripting is the same as the catenation of the shapes of each of the subscripts [that is, (⍴sub1),⍴sub2]. No column positions are indicated, so all columns are selected. In *FN*1, the scalar subscript causes *L* to be a vector, while in *FN*2, the vector subscript causes *L* to be a 1×n matrix.

Now, the statement that's trying to print the words "*THIS IS* " in front of the line has no problem when *L* is a vector. But when *FN*2 attempts to catenate an eight-element vector ("*THIS IS* ") with a 1×n-matrix, a *LENGTH ERROR* results. At this point, there isn't likely to be any suspicion directed toward *C* until much time has been wasted.

This is the real problem with fake catenation: unless great care is exercised, it often produces treacherous (and elusive) side effects.

Remember also that each of those extra symbols that are used for the fake catenation takes up storage space as well as execution time. We recently had an interesting discussion with an APLer who was tracing the execution of a workspace and carefully measuring the results. Since the workspace was filled with functions containing artificially-long lines, he decided to take the time to rid the workspace of the fake catenation once and for all. Well, at the conclusion of his efforts he measured it again: he had saved twenty percent on storage space and *fifty* percent on execution time! Now, we can't guarantee that *your* functions will run twice as fast if you remove the instances of fake catenation, but everything helps... and they certainly won't be any slower.

In general, one-liners are a waste of time... yours *and* the computer's. Abolishing one-liners from your library can save you needless errors and debugging time, and can ease your lot in life in understanding the code. It can also save you *WS FULL* problems and let your code run faster, saving you money and your own time.

Much more productivity can be realized if each line of an APL function contains *one thought.*

→LOOPING versus USING ARRAYS

Part of the real power of APL is its ability to deal directly with arrays, without having to be concerned with lots of nuts-and-bolts details; the APL designers have done an admirable job of hiding all of that stuff beneath the surface. If you have a thousand numbers stored in an array called *TABLE*, and you choose to add two to every element, *TABLE*+2 or 2+*TABLE* is all that's needed. Show a table of numbers to a youngster, and tell him that it is called "*TABLE*," then ask him to write down the arithmetic for adding two to each number.

It seems sad, then, with APL working in such straightforward ways, that so many people insist on writing loops into all of their code. I recently saw this example, not once, but many times in one function:

```
ARRAY[1;8;]←MATRIX[1;]
ARRAY[2;8;]←MATRIX[2;]
ARRAY[3;8;]←MATRIX[3;]
ARRAY[4;8;]←MATRIX[4;]
ARRAY[5;8;]←MATRIX[5;]
ARRAY[6;8;]←MATRIX[6;]
```

Bad Example Number One

In this example, *ARRAY* is a three-dimensional array, although the only row that we are changing is row 8. It is being fed some data from *MATRIX*, which has the same number of columns as *ARRAY* has. And, the number of planes in *ARRAY* is the same as the number of rows in *MATRIX*; in other words, their first dimensions match.

This isn't meant to be a reflection upon the person who wrote the function. He was simply using the background that he had acquired in using other languages, and perhaps didn't realize how much effort he could have saved by going to simpler approaches. Now, unfortunately, the "simpler" approaches that some people resort to is to remove that in-line code, and replace it with a loop, like this:

```
 C←0
 H←1↑ρMATRIX
LP:C←C+1
 →(C>H)/0
 ARRAY[C;8;]←MATRIX[C;]
 →LP
```

Bad Example Number Two

Bad Example Number Two does have some advantages over Bad Example Number One. For instance, since it checks the height of *ARRAY*, notice that it will work regardless of how many planes there are in *ARRAY*. Well, that's progress; at least the author wouldn't have to add more lines of code if the size of the array were increased some day.

However, all that's needed to accomplish that same task is one concise expression:

```
ARRAY[;8;]←MATRIX
```

[I figured that I'd better set this one off from the text the same as the other two examples, or everybody would go skipping right past it.]

Yep, that last example does just the same thing as the first two examples... but it does it *much* more quickly, and with many fewer keystrokes for the author. And consider the job that APL has to go through to execute any of these functions.

One of the frequent criticisms of APL is that it is "interpretive"; that is, each symbol that you key in has to be separately resolved by APL before it can do any useful work for you. An alternate approach is a "compiled" program, in which the compilation step sets up internal pointers to the data and operations and allows that program to run very fast. Taking advantage of arrays, however, gives you the best of both worlds. Some of the computer time that you use is spent resolving the interpretation of the names and symbols, and some of it is spent actually manipulating your data. Obviously, we would like the first time to be zero and the second time to be 100%. *If* the number of names and symbols that you use is small, the interpretation is very quick, and the bulk of the execution time is expended toward your end product. The internal coding for each of the APL primitive functions has been written in System 370 Assembler Code by some very knowledgeable system programmers, and most of the code has been optimized for lots of special cases. Doing things in big steps with arrays instead of little steps with loops allows most of the data movement to be handled by these optimized modules. In some cases, you'd be hard pressed to beat the speed even if you were writing your own programs in Assembler Language instead of APL, because the APL modules have years of optimization behind them.

As an example of how significant the gains in using arrays can be, consider the following examples:

```
      SCALAR←2
      VECTOR←1000⍴2
```

We constructed several expressions and timed each of them carefully, running each expression 100 times, and then took the average. "A←SCALAR+SCALAR" took 0.2 milliseconds, while "A←VECTOR+VECTOR" took 1.0 millisecond.[19] That says that we can add a thousand numbers to a thousand other numbers in just five times as long as it takes to add two numbers together (2+2). Almost all of the time for the SCALAR+SCALAR case is used for interpretation.

Projecting that out a bit, that says that if we added those two scalars together a thousand times (to equal the work done by VECTOR+VECTOR), it would take 200 milliseconds... two hundred times as long as the array operation. Would you care for an easy 200× speed-up for your functions? But wait, even that isn't the whole story... If you are adding two numbers together a thousand times, it's unlikely that you actually have 1000 identical lines of SCALAR+SCALAR in your function; you're probably using that old devil loop again. And, of course, each of the symbols that comprise the loop have to be interpreted 1000 times. Trying this, we came up with an average time of 892.3 milliseconds. That's 892 times longer than the VECTOR+VECTOR case, and the result is the same.

Where else can you get an 892× performance improvement by *removing* code?

This brief dissertation isn't meant to teach anyone how to use arrays; it's simply meant to interest you in pursuing them further. For more information on the use of arrays, look in *APL2 Programming: Language Reference*, under "Arrays" or "Scalar Functions."

* * *

[19] These numbers are for comparison only; your actual mileage may differ.

Wil 3-Chr Nms Run Fst?

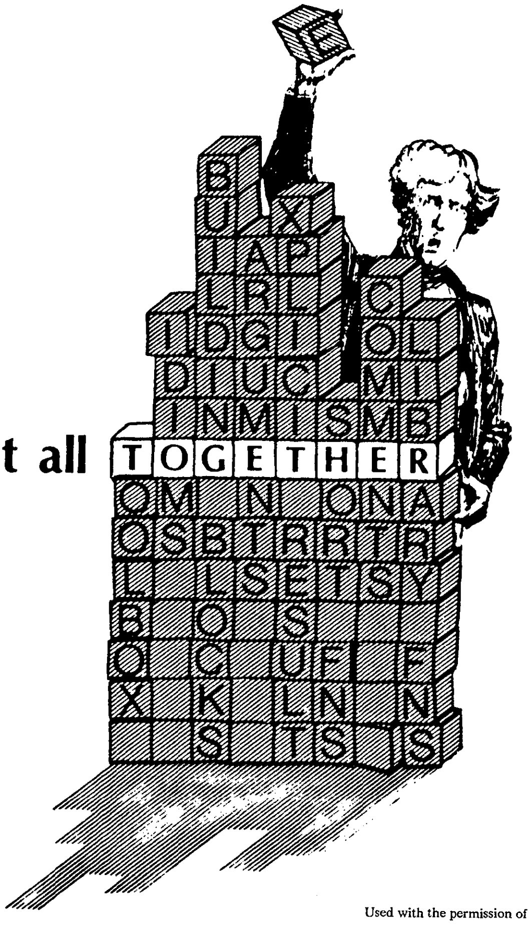
Used with the permission of Hart Publishing Company, Inc.

Perhaps you've heard the scam: "Hey, buddy, wanna make your APL functions run *real fast?* ...jes' keep all of the names to three characters or less...."

Well, first of all, it *is* true that most APL systems process short names better than they process long names — APL simply ends up doing more work to manipulate long names. *But...* that extra amount of work that it has to do is *trivial.* Probably you won't even be able to measure the difference, whereas we just finished discussing other changes in programming techniques that can result in improvements that may make the code run *hundreds* of times faster. Don't even consider the three-character limit as being significant; on most early APL systems the limit was three, but that limit is purely implementation dependent, and isn't even being used anymore.

If there is ever any chance that a longer name will be more readable or more descriptive, vote in favor of the more meaningful name.

Putting it all TOGETHER



Used with the permission of Hart Publishing Company, Inc.

Okay, I *would* like to write modular code.... But *how*?

* * *

Your manager just walked into your office and asked you if there is any chance of whipping up a part number inventory report by tomorrow morning. You say "no problem," and he quickly leaves before your brain has a chance to interpret the words your mouth has just volunteered. Oh oh.

Where Do You Go From Here?

Obviously, you go for a cup of coffee; that's an important step in programming. Of course, you ask your office mate to join you, knowing that he created the part number inventory file that you will be using. That free coffee that you get him quickly yields some free information for you. Not bad. As you return to your desk, still scribbling record formats on your coffee cup, you begin to think about the last five "quick and dirty" functions you have written that never died, but rather continue to haunt you with daily updates and fixes. You suddenly make up your mind that this program is going to be the eagle of functions, not another albatross. It's going to be complete, dependable, logical, efficient, idiot proof, and also maintainable. Sounds like you'll be burning the midnight oil? Maybe not; read on....

Should You Make a Flowchart?

...or a "HIPO"?
...or maybe a Nassi-Schneiderman chart?
(...whatever *that* is)

Well, you could do that if you had the time, but when you finish it, you only have something that is human readable (at best), not machine readable. Perhaps there is a way to do the problem analysis at the terminal. First, get a fresh, clean APL flowchart pad, via:

```
      )CLEAR
CLEAR WS
```

Probably it would be good to ensure the that you haven't used up your workspace quota:

```
      )SAVE INVENTORY
   7/13/1984  11.55.18 (GMT-5)
```

Now, in the most basic terms (probably those of your manager), what are you going to accomplish?

```
      ∇INVENTORY
[1]   ⍝REPORTS CURRENT PART NBR QTY FROM FILE
```

Now what logically needs to be done first? Is there any input required? Remembering that your manager doesn't have time to scan through lengthy reports, you decide to allow for part number selection:

```
[2]   ⍝PROMPT FOR PART NUMBERS:
[3]   PN←GETPN
```

Hmmm... let's see, what happens if he types in something invalid? Well, you can put some code in the *GETPN* function to check for this and print an appropriate error message. But then how do you get it to exit the main calling function (*INVENTORY*)? Or what if he just presses return without typing in any part number? In either case, having the *GETPN* function return a null vector is an easy (and fairly common) way of signalling that some other action is to be taken. So let's use a null response from *GETPN* to cause an exit from *INVENTORY*:

```
[4]   →(0=⍴PN)/0
```

What do you need to do next? Remember, don't go into any detail at this point. You just want a general overview.

```
[5]   ⍝SEARCH INV FILE, RETURN PN'S AND QTYS:
[6]   PNQ←SEARCH PN
```

It might help to sort the part numbers on your report.

```
[7]   ⍝SORT BY PART NUMBER:
[8]   PNQ←SORT PNQ
```

You can't think of anything else you need to do besides producing the report? Ok, then wrap up this function.

```
[9]    ⍝FORMAT AND DISPLAY:
[10]   DISPLAY PNQ
[11]   ∇
```

You probably recognize by now that I am suggesting a *top-down* approach to APL programming. By breaking the problem down into logical steps, you will find yourself writing modular code without even thinking about it. And by freely supplying comments, your functions will be even more easily maintained. If you are concerned about ⎕*WA* (the amount of workspace available), you can strip the comments (and, optionally, lock the functions) and save them in a production workspace, leaving your original, commented version as the maintenance copy. There may even be functions for doing this available in the APL Public Library on your system. Check around.

Now that you have the main function written for your inventory report, you can define the first subfunction that you called: *GETPN*.

```
       ∇PN←GETPN
[1]    ⍝PROMPTS USER FOR PART NUMBERS
[2]    ⍝PN ↔ NUMERIC VECTOR OF PART NUMBERS
           |
    (...more detail will be shown in
       the next subfunction, SEARCH)
           ↓
[7]    ∇
```

Without going into detail on this particular function, I would like to offer a couple of general comments on prompting for input.

Keep Your Prompts Short

This will be greatly appreciated by your users, especially if your function does repetitive prompting. You may wish to use a longer initial prompt, followed by a short form thereafter. I sometimes expect that when I meet programmers who use prompts like "*ENTER YOUR NAME:* ", they will greet me with "*TELL ME YOUR NAME*", and "*TELL ME HOW YOU ARE*". What's wrong with simple *questions* like "*NAME?*", "*RANK?*", or "*SERIAL NUMBER?*". However, watch out for ambiguous short prompts; your intentions may be misunderstood.

Do Lots of Auditing

You will thank yourself every night that you *don't* get a frantic 3am call from a user who mistyped an "*O*" for a "0." Here's a good place to use "toolbox" functions. You can write, beg, or borrow functions (or APL idioms) for auditing numeric or character data, testing dimensions of input, and for auditing special inputs like dates and times.

Now, let's get back to the part number inventory report. Forgot where you left off? No problem, just look back at your "APL flowchart":

```
       ∇INVENTORY[⎕]∇
     ∇ INVENTORY
[1]    ⍝REPORTS CURRENT PART NBR QTY FROM FILE
[2]    ⍝PROMPT FOR PART NUMBERS:
[3]     PN←GETPN
[4]     →(0=⍴PN)/0
[5]    ⍝SEARCH INV FILE, RETURN PN'S AND QTYS:
[6]     PNQ←SEARCH PN
[7]    ⍝SORT BY PART NUMBER:
[8]     PNQ←SORT PNQ
[9]    ⍝FORMAT AND DISPLAY:
[10]    DISPLAY PNQ
     ∇    7/13/1984  12.22.13 (GMT-5)
```

Ok, you've got some part numbers (via *GETPN*). Now search the inventory file.

```
       ∇Z←SEARCH PN
[1]    ⍝SEARCHES FILE TO GET TOTAL PN QTYS
```

Always be sure to describe the arguments and the explicit result of your function.

```
[2]    ⍝PN ↔ NUMERIC VECTOR OF PART NUMBERS
[3]    ⍝Z ↔ TWO-COLUMN MATRIX OF PN'S AND QTYS
```

First you have to open the file. You could open it for either direct access or sequential reading. For this project (assuming that you need to go all the way through the file) a sequential read is reasonable. If you don't already have a toolbox function for opening a dataset, try to generalize this function by using arguments, so that you can use it again on another project.

```
[4]    OPEN 'PNINV'
```

Next, the plan is to read a block of records at a time, each containing a part number and serial number (according to the notes on your coffee cup). You will increment the appropriate part number quantity for each serial number. Before you start, you want to set the *PN* quantities to 0.

```
[5]    Z←PN,[1.5] 0
[6]    READLOOP:
[7]    →(0=1↑⍴BLOCK←READ)/END
[8]    ⍝UPDATE QTYS BASED ON PN'S IN BLOCK[;1]:
```

At this point you have two options. If you are not sure of the APL expression to do this, don't let that slow you down now. Simply call a subfunction to do the job. (You can code it later.)

```
[9]    Z←BLOCK[;1] UPDATE Z
```

...Or, if the content of the *UPDATE* function is just one brief line and you know what the process is to be, you can just enter it here directly:

```
[10]   [9]
[9]    Z[;2]←Z[;2]++/PN∘.=BLOCK[;1]
```

...and now you can go back and get the next record:

```
[10]   →READLOOP
```

Well, you've done what you said you were going to do at the beginning of this *SEARCH* function, so close it out:

```
[11]  END:
[12]  [⎕]∇
     ∇ Z←SEARCH PN
[1]   ⍝SEARCHES FILE TO GET TOTAL PN QTYS
[2]   ⍝PN ↔ NUMERIC VECTOR OF PART NUMBERS
[3]   ⍝Z ↔ TWO-COLUMN MATRIX OF PN'S AND QTYS
[4]    OPEN 'PNINV'
[5]    Z←PN,[1.5] 0
[6]   READLOOP:
[7]    →(0=1↑⍴BLOCK←READ)/END
[8]   ⍝UPDATE QTYS BASED ON PN'S IN BLOCK[;1]:
[9]    Z[;2]←Z[;2]++/PN∘.=BLOCK[;1]
[10]   →READLOOP
[11]  END:
     ∇    7/13/1984  12.40.37 (GMT-5)
```

Looking back at your main function *INVENTORY*, you can see that your next job is to sort the part numbers. At this point you realize that this particular sort is really quite trivial, and a function call probably isn't necessary. You could easily go back and edit your "flowchart":

```
        ∇INVENTORY[⎕8]
[8]     PNQ←SORT PNQ
[8]     PNQ←PNQ[⍋PNQ[;1];]
[9]     ∇
```

However, if this had been a more complicated sort (perhaps a file sort) you could have checked the APL Public Library... there may be all sorts of sorts around.

You've reached the end of your main function. You can now write the *DISPLAY* function (with appropriate bells and whistles to impress your manager). Then go back and take care of the next lower level of subfunctions (that is, *OPEN*, *READ*, and so forth) in a similar manner. Also, you will need to localize some variables in the functions you have written. And *do* take the time to be neat about localization of variables. Don't leave variables around as global objects unnecessarily. That could cause some confusions later (as described in the "Mystification to Avoid" discussion on page 83).

A side benefit of this *top-down* programming approach is that you can start testing your functions at any point along the way by writing dummy subfunctions. For instance, suppose that the input file (*PNINV*) hadn't yet been created by the time you were ready to test your functions. You could quickly write a null *OPEN* function (just a header line) and a *READ* function that simply returned a matrix of sample records.

At this point, you (hopefully) feel that you've done a good job, so don't forget to sign your work:

```
        ∇INVENTORY[.1]
[0.1] ⍝VAN DER MEULEN, INVENTORY CONTROL DEPT
[0.2] ∇
        ∇GETPN[.1]
[0.1] ⍝VAN DER MEULEN, INVENTORY CONTROL DEPT
[0.2] ∇
        ∇SEARCH[.1]
[0.1] ⍝VAN DER MEULEN, INVENTORY CONTROL DEPT
[0.2] ∇

         )SAVE
      7/13/1984  12.45.17 (GMT-5) INVENTORY
```

Here, let's even include some "human factors" stuff, to make it easier for someone else to get started with it:

```
      ∇LX
[1]   'REPORTS CURRENT PART NUMBER QUANTITIES'
[2]   '...TYPE  INVENTORY  TO START'
[3]   ''∇

      ⎕LX←'LX'

      )SAVE
   7/13/1984  12.46.52 (GMT-5) INVENTORY
```

Now it will "come up running"; let's try it:

```
      )LOAD INVENTORY
SAVED   7/13/1984  12.46.52 (GMT-5)
REPORTS CURRENT PART NUMBER QUANTITIES
...TYPE  INVENTORY  TO START
```

When you look back at your functions after finishing the project, you see a lot of short, concise modules, logically linked together under a main function. As you look closer at individual functions, you're almost embarrassed by their simplicity. Congratulate yourself on a job well done.

Appendix. Reference Section

Oh-Oh! I just remembered that I left my Migration Guide back in the office!

Used with the permission of The Dick Sutphen Studio.

List of Major Extensions, New Features and Differences

APL2 offers a lot of power that wasn't previously available to APL users. For reasons of conversion and compatibility, it may be important to identify the new features. Here is a list of the items which have been added, extended, or changed with respect to previous versions of APL. Some of these changes are discussed in this manual; all of them are documented in detail in *APL2 Programming: Language Reference*. Further discussions of differences between various versions of APL can be found in *APL2 Migration Guide*.

Extended Items

1. **Miscellaneous:**

   a. “¯” and “_” are alphanumeric characters
   b. Selective specification
   c. Parentheses are accepted in function expressions
   d. Any statement may end with a comment
   e. Overstrike combinations exist for national characters and lowercase characters
   f. Error reports normally show two error carets
   g. Some system variables are session variables
   h. Some empty arrays may display on multiple lines
   i. The active workspace is saved into the user's library after a *SYSTEM ERROR*
   j. Halted, partially-executed statements may be resumed
   k. The four properties of a locked function may be set independently
   l. System commands may have comments

2. **Monadic Primitive Functions:**

   a. ⍕ formats columns independently
   b. / accepts integer left argument
   c. ⍒ accepts arrays of any nonscalar rank
   d. ⍋ accepts arrays of any nonscalar rank

3. **Dyadic Primitive Functions:**

   a. ○ accepts `¯12 ¯11 ¯10 ¯9 ¯8 8 9 10 11 12` left argument
   b. ⍕ accepts a character left argument

4. **Monadic Primitive Operators:**

   a. / accepts a defined or derived function operand
   b. \ accepts a defined or derived function operand

5. **Dyadic Primitive Operators:**

   a. . accepts a defined or derived function operand

6. **Monadic System Functions:**

   a. ⎕*EX* may expunge a suspended, pendent or active function
   b. ⎕*FX* accepts vector of vectors and operator definitions
   c. ⎕*FX* may fix a suspended, pendent or active function
   d. ⎕*NC* class 4 means operator
   e. ⎕*NL* accepts 4

7. **System Variables:**

   a. ⎕*PP* may be up to 18

8. **Dyadic System Functions:**

   a. ⎕*NL* accepts 4 in its right argument

9. **System Commands:**

   a.)*COPY* accepts indirect lists in parentheses
   b.)*COPY* will attempt to continue after a *WS FULL*
   c.)*PCOPY* accepts indirect lists in parentheses
   d.)*PCOPY* will attempt to continue after a *WS FULL*
   e.)*ERASE* accepts indirect lists in parentheses
   f.)*FNS* accepts an optional second argument
   g.)*LIB* lists workspace names across the page
   h.)*SYMBOLS* may change the size of the symbol table at any time
   i.)*VARS* accepts an optional second argument
   j. Comments may be entered on system command lines

10. ∇ **Editor (Default):**

    a. The editor will edit defined operators
    b. [→] command to abort
    c. [⎕n] accepts a vector argument
    d. [∆n] accepts a vector argument
    e. [⎕n] accepts an interval argument with "-"
    f. [∆n] accepts an interval argument with "-"
    g. Previous function kept following a name change
    h. Recursive editing

New Items

1. **Miscellaneous:**

    a. National language translation
    b. Defined operators
    c. Vector notation
    d. Characters ⌻ ⍸ ⍷ ⎕ ⍂ ≡ ∵
    e. Session parameters

2. **Data Types:**

    a. Complex numbers
    b. Mixed arrays
    c. Nested arrays

3. **Monadic Primitive Functions:**

    a. ≡ — Depth
    b. ⊂ — Enclose
    c. ↑ — First
    d. ⊃ — Disclose
    e. ⊃[*A*] — Disclose with Axis
    f. ⊂[*A*] — Enclose with Axis
    g. ,[*A*] — Ravel with Axis

4. **Dyadic Primitive Functions:**

    a. ⍷ — Find
    b. ⍒ — Grade Down
    c. ⍋ — Grade Up
    d. ≡ — Match
    e. ⊃ — Pick
    f. ~ — Without
    g. ↓[*A*] — Drop with Axis
    h. ↑[*A*] — Take with Axis

5. **Monadic Primitive Operators:**

   a. [*A*] — Axis specification for scalar functions
   b. ¨ — Each
   c. / — N-wise Reduce

6. **System Variables:**

   a. ⎕*EM* — Event Message
   b. ⎕*ET* — Event Type ⎕*ET*
   c. ⎕*FC* — Format Control
   d. ⎕*L* — Left Argument
   e. ⎕*NLT* — National Language Translation
   f. ⎕*PR* — Prompt Replacement
   g. ⎕*R* — Right Argument
   h. ⎕*SVE* — Shared Variable Event
   i. ⎕*TZ* — Time Zone

7. **Monadic System Functions:** ⎕*EC* ⎕*NA*

   a. ⎕*AF* — Atomic Function
   b. ⎕*EC* — Execute Controlled
   c. ⎕*ES* — Event Simulation
   d. ⎕*NA* — Name Association
   e. ⎕*SVS* — Shared Variable State

8. **Dyadic System Functions:**

   a. ⎕*AT* — Attributes
   b. ⎕*EA* — Execute Alternate
   c. ⎕*ES* — Event Simulation
   d. ⎕*FX* — Fix
   e. ⎕*NA* — Name Association
   f. ⎕*TF* — Transfer Form

9. **System Commands:**

   a.)*EDITOR* specifies ∇ system editor
   b.)*HOST* executes host system commands
   c.)*IN* retrieves objects from transfer file
   d.)*MCOPY* copies objects from VS APL workspaces
   e.)*MORE* provides additional error information
   f.)*NMS* lists names of objects
   g.)*OPS* list names of operators
   h.)*OUT* saves objects onto transfer file
   i.)*PBS* sets the printable backspace character
   j.)*RESET* clears the state indicator
   k.)*SIS* lists the state indicator with statements
   l.)*TIME* displays the date and time

10. ∇ **Editor (Extended):**

   a. Full screen processing

11. **Messages:**

a. *AXIS ERROR*
b. *SYSTEM LIMIT*
c. *VALENCE ERROR*

Items Which May Return Different Results

There are several features or operations in APL2 that can produce different results from those in previous versions of APL. This list does not include extensions (operations which produced errors in previous versions, but do not produce errors in APL2).

1. The Atomic Vector (⎕*AV*) is different. In particular, the alphabets are not contiguous.

2. "¯" and "_" are alphanumeric characters.

3. ⎕*NC* name class 4 means operator.

4. ⎕*NC* name class ¯1 means invalid name.

5. The system function name class (⎕*NC*) applies to distinguished names (system variables and system functions).

6. The result of ⎕*EX*, ⎕*NC*, ⎕*SVO*, or ⎕*SVR* applied to a vector is a scalar.

7. The backspace character, the new line character (carriage return), and the line feed character are not permitted in a character constant or in function definition.

8. Lowercase letters, new APL2 characters ⌼ ⍷ ⍸ ⌷ ⍂ ≡ ∵, national use characters ¢ | ! $ ¬ ¦ ` # @ " ~ { } \, and special characters & and % are permitted in character constants and comments.

9. The result of the system function canonical representation (⎕*CR*) separates local names in the function header with blanks.

10. The result of ⎕*CR* may contain some lines which are entirely blank.

11. Numeric constants in the canonical representation of a function retain the same precision with which they were entered.

12. The system function fix (⎕*FX*) will accept blanks as the separator between local names in a function header.

13. Suspended, pendent or active defined functions may be expunged (with ⎕*EX*) or fixed (with ⎕*FX*).

14. Referencing ⍞ always produces a vector.

15. Referencing ⍞ after setting ⍞ with a prompt returns the composite of the prompt and the keyboard entry.

16. ⎕*CT* is an implicit argument of the function residue (|).

17. ⎕*CT* is an implicit argument of the function encode (⊤).

18. Negative integers are not in the domain of the dyadic binomial (!) function.

19. An odd root of a negative number (like ¯8 * ÷ 3) is a complex number.

20. The result of ¯4○R is the *negative* square root if the argument R is negative.

21. The monadic format (⍕) or default display of a (simple) numeric matrix has its columns formatted independently; therefore, it does not contain a leading column of blanks.

22. The result of dyadic format L⍕R, where L is a single nonzero integer, and R is less than 1, does not leave a blank for the units digit.

23. The display of a multidimensional array is folded at ⎕PW, if necessary, plane by plane, rather than line by line.

24. The display of an empty array having rank greater than one may use zero lines, or may extend to multiple lines.

25. The execution of the dyadic system function ⎕SVO is not necessarily atomic. If multiple shares are offered simultaneously, it is possible to exhaust the shared variable quota before all shares are fulfilled. In such a case, after a SYSTEM LIMIT error, some shares may be fulfilled while others are not.

26. If the left argument of the dyadic system function ⎕SVO is a one-element vector, then it does not extend (although a scalar left argument will still extend in the normal fashion).

27. The *dyadic* system function shared variable query (⎕SVQ) is not supported.

28. The edit command [⎕n] will display only line n of the function being edited. The command [⎕n-] will display from line n to the end of the function.

29. Changing the name of a function with the system editor creates a new function *without* affecting the original function.

30. Settings of stop control (S∆) and trace control (T∆) are not relocated as a result of line insertion or deletion by the system editor.

31. Statements entered in immediate execution that are interrupted by an error are placed in the SI stack, and may be resumed by entering →⍳0.

Withdrawal of Obsolete Facilities

I-Beams Have Been Removed

The ancient I-beam functions left over from the early APL\360 days, are no longer supported. In their place, you should be using these system variables:

I-beam	Approximate replacement	Purpose
⌶19	¯1↑⎕AI	Keyboard Unlock time
⌶20	3↓⎕TS	Time of day
⌶21	⎕AI[2]	CPU time (compute) time used during this session
⌶22	⎕WA	Amount of workspace available
⌶23	⎕UL	User load
⌶24	⎕AI[3]	Session start time
⌶25	3↑⎕TS	Current date
⌶26	1↑⎕LC	Current line number being executed
⌶27	⎕LC	Vector of line numbers in the State Indicator
⌶28	⎕TT	Terminal type
⌶29	1↑⎕AI	User number

By "approximate replacement," we mean that the recommended expression yields roughly the same information, although it's typically in different units. The newer facilities are in generally much more "user-friendly" units than the I-beams were. For example, ⎕TS returns the current time and date as year-month-day-hour-minute-second-millisecond... ⌶20 gave the time in 60-ths of a second since the last midnight previous to your sign-on. It would therefore make sense to rewrite the expression in which they appear rather than to convert the quads to old units, and then back to "friendly" units.

Heterogeneous Output Has Been Removed

Heterogeneous output (sometimes called mixed output), was the old practice of printing both numeric and character data on the same line by separating them with semicolons:

```
      N←127
      'HEIGHT IS ';N;' UNITS'
HEIGHT IS 127 UNITS
```

A better approach is to format the numeric data into character data, like this:

```
      'HEIGHT IS ',(⍕N),' UNITS'
HEIGHT IS 127 UNITS
```

or, to simply catenate the character and numeric data together, like this:

```
      'HEIGHT IS',N,'UNITS'
HEIGHT IS 127 UNITS
```

or, finally, to simply display the character and numeric data side-by-side, without catenation, like this:

```
      'HEIGHT IS' N 'UNITS'
HEIGHT IS 127 UNITS
```

This last method is particularly effective where the data to be displayed may not be conformable for catenation, and you wish to have APL handle the display formatting:

```
      M←3 4⍴⍳12
      'THE RESULT IS' M
THE RESULT IS   1  2  3  4
                5  6  7  8
                9 10 11 12
```

The withdrawal of heterogeneous output was not an arbitrary change. The old construction was offered back in APL\360 days, when there was no easy way to format the numerics into character data, but this use of the semicolon *never was* a proper APL construction. In particular, its "result" cannot be assigned to a variable or passed to a function as an argument. It was a convenience whose necessity has been outlived.

Index

An Introduction to APL2
SH20-9229-1

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ______________________

Previous TNL ___________________

Previous TNL ___________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SH20-9229-1

Reader's Comment Form

d and tape

Please do not staple

Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

and tape

Please do not staple

Fold and tape

IBM®

An Introduction to APL2 (File No. S370-40) Printed in U.S.A. SH20-9229-1

An Introduction to APL2
SH20-9229-1

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ______________________

Previous TNL ___________________

Previous TNL ___________________

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

SH20-9229-1

Reader's Comment Form

Fold and tape Please do not staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

ld and tape Please do not staple Fold and tape

IBM®

An Introduction to APL2 (File No. S370-40) Printed in U.S.A. SH20-9229-1

IBM®

An Introduction to APL2

File Number S370-40

SH20-9229-01

Printed in U.S.A.