CONTROL DATA
CORPORATION

CONTROL DATA®
CYBER 70 SERIES
6000 SERIES
COMPUTER SYSTEMS

APL*CYBER
REFERENCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Original printing
(6-73)	
B	Reflect current implementation; support KRONOS 2.1 operating system. This printing obsoletes
(7-73)	revision A.

Publication No.
19980400

Printed in the United States of America

Address comments concerning this manual to:

Control Data Corporation
Technical Publications Department
4201 North Lexington Avenue
Arden Hills, Minnesota 55112

or use Comment Sheet in the back of this manual.

PREFACE

This is the reference manual for APL*CYBER, version 1.0. APL*CYBER runs on all models of CONTROL DATA® CYBER 70 Series and 6000 Series computers, and is currently implemented for use with the KRONOS operating system. No knowledge of any operating system or hardware characteristics is required other than that included in the appendixes of this manual.

CONTENTS

APPENDICES

INTRODUCTION 1

APL - THE LANGUAGE

The Language APL and its acronym are derived from the mathematical language propounded by K. E. Iverson in a book entitled "A Programming Language" (John Wiley and Sons, Inc. 1962).

The Language is essentially a large set of primitive, i.e., predefined, functions for manipulating and performing computations on data. The notation used is very compact. A single APL character conveys the primitive function desired, and function expressions consist of an infix notation associating the arguments with the function being called. Primitive functions have one or two arguments. One argument appears to the right of the APL character conveying the desired function. If a second argument is required it appears to the left of this character. Arguments can themselves be function expressions. Evaluation of the expression proceeds from right to left.

Unlike functions in other programming languages, most primitive functions in APL are defined for general arguments. While scalar (single valued) arguments are possible as a special case, in general the arguments are array data structures and the functions operate in a predefined manner on these structures as a whole.

THE APL*CYBER SYSTEM

The implementation of APL on CYBER computers is known as the APL*CYBER system.

The principal component of the system is a conversationally interactive interpreter designed for time sharing terminal operation. Upon gaining access to the system, APL expressions keyed on a terminal are evaluated and results, if requested, are displayed immediately.

In addition to operating the system as a sophisticated desk calculator, the following features endow it with the capabilities of a complete programming system.

- A procedure exists for a user to define his own APL functions in terms of APL expressions using previously defined or existing functions.
- A user library facility exists whereby such functions and previously input or processed data can be stored for subsequent use of for interchange with other users.
- Extensive diagnostics, debugging aids and editing facilities exist to make the APL programmer extremely productive.
- Methods exist whereby a variety of terminal types can gain access to the APL*CYBER system and exchange data and programs.
- Batch users may also employ the system in batch mode.

SPECIAL NOTATION

The following notation is not part of the APL language but rather is used in describing that language.

[ ] indicates the contents are optionally included.

{ } select one.

... repeat as required.

< > indicates a descriptive term rather than a literal APL construct.

↔ indicates identity, i.e., that the expression on the left has the same value as the expression on the right. If used in the context of a constraint, the expressions must have the same value for the constraint to be satisfied.

NOTE ON EXAMPLES

Where examples are shown in this manual, a clear workspace (see ')CLEAR') is understood to exist prior to input of the first line, unless otherwise stated or implied by the example itself.

DATA 2

ARRAYS

All data in APL is handled in the form of arrays. An ARRAY is a finite, ordered set of data elements, arranged in a multi-dimensional structure of mutually orthogonal coordinate axes.

The number of dimensions, or coordinate axes, is called the RANK of the array. This number is necessarily a non-negative integer. The value of a particular dimension is the number of element positions within the array along the corresponding coordinate axis, and is termed the LENGTH of the coordinate axis. This number is also necessarily a non-negative integer. The set of coordinate lengths is called the SHAPE of the array.

For example, a 3 by 4 matrix is a rank 2 array. Its dimensions, or coordinate lengths, are 3 and 4, and its shape is 3 4.

All elements in the array must be of the same DATA TYPE (see below). The VALUE of an array is determined by its data type, its shape, and the values of all the elements in the array.

DATA TYPES

Two data types are defined in APL*CYBER: numeric and character. The value of a numeric data type is a single number (e. g. , the number 5. 3). The value of a character data element is a single character (e. g. , the character "A", or "+", or "1").

ARRAY STRUCTURES

The most primitive array is an array of rank zero. This structure is the classical SCALAR of tensor analysis, and is analogous to a point in geometry. A scalar has no dimensions and exactly one element.

Example:

```
      4                    ( a numeric scalar)
```

The most common multi-element array is the one-dimensional array. This structure, called a VECTOR, is analogous to a line in geometry. As a line may have arbitrary length, so a vector may have an arbitrary number of elements.

Example:
```
7 1.3 5
```
(a three-element numeric vector)

A MATRIX is much like a vector, except that the structure has two orthogonal coordinate axes instead of one, as in a plane versus a line. A matrix can be thought of as a set of row vectors arranged along a vertical coordinate axis, or as a set of column vectors arranged along a horizontal coordinate axis. The first coordinate axis, by convention, is usually considered the "vertical" axis, and the second the "horizontal" axis.

Example:
```
 1  7 12
16  2  3
```
(a 2 by 3 numeric matrix)

Higher rank arrays are built in a similar fashion. The language does not define any limit to the rank of an array. However, the APL*CYBER implementation will not allow the user to create arrays of rank greater than 127. Any attempt to exceed this limit will result in an error message (usually a RANK ERROR).

ELEMENT POSITION

The position of an element within an array is determined by a set of ordinals, one for each coordinate axis, called the COORDINATES of the element. The value of a particular coordinate is the ordinal for the position of the element on the corresponding coordinate axis.

The ordinal values used to designate a specific element depend on the setting of ORIGIN (q.v.). In ORIGIN 1 the first position is denoted by 1, the second by 2, etc.

VISUAL CONVENTIONS

In vectors, the last (and only) coordinate axis is considered to be horizontal, with the left-most position being considered the first. Thus, if X is the vector `7 5 10`, `X[1]` (see INDEXING) is the element 7.

In matrices, the last coordinate axis is considered to be horizontal, and the second-to-last (namely, the first) is considered to be vertical. The first column is the left-most, and the first row is the top row.

For example, suppose that X is the 3 by 4 array below:

```
 1  7 12  5
16  2 10  9
11  4  3 15
```

Then, X[2;3] (see INDEXING) is the element which is in position 2 on the first coordinate axis (namely, row 2) and position 3 on the second coordinate axis (namely, column 3). The value of this element in the above example is 10.

For arrays of rank 3 or greater, visual conventions do not exist, because today's displays are two-dimensional (however, see DISPLAYING DATA).

ABSOLUTE ELEMENT ORDER

For most operations, the elements of an array do not form a well-ordered set. That is, given any two elements in an array, it is not generally possible to state which comes before the other. However, two important exceptions should be noted.

The RAVEL function (q. v.) creates a vector of the elements in an array. The first element of the resulting vector is the element in the first position on all coordinate axes of the argument. The elements following this are chosen along the last coordinate axis of the argument. When these are exhausted, they are followed by the elements along the last coordinate axis in the next position of the next-to-last coordinate, and so on until all coordinate axes have been accounted for. This ordering is called ROW MAJOR order.

Example:

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

The RESHAPE function (see RESHAPE) creates an array of the specified shape from the elements in the order given. This operation may be thought of as an inverse operation to RAVEL.

Example:

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

SHAPE DETERMINATION

The shape of an array is represented by a vector, each element of which is the value of the corresponding dimension of the array. The shape of a 3 by 4 matrix is 3 4 . The shape of a six-element vector is 6. In general, the shape of an array may be found by the monadic Rho function (see SHAPE):

```
      ⍴1 3 7 6 12 4
6
      X←2 3⍴1 3 7 6 12 4
      ⍴X
2 3
```

NUMBER OF ELEMENTS

The number of elements in an array may be found from the shape of the ravel of the array:

```
      ⍴,X
6
```

An array which has one or more zero length dimensions contains no elements and is said to be EMPTY:

```
      X←2 0 3⍴1 3 7 6 12 4
      ⍴X
2 0 3
      ⍴,X
0
```

Since a scalar has no dimensions, its shape is empty. Note the difference between a scalar, which has one element and no dimensions, and an empty array, which has no elements and one or more dimensions:

```
      X←5
      ⍴,X
1
SHAPE←⍴X


      ⍴,SHAPE
0
```

RANK DETERMINATION

Since the rank of an array is the number of dimensions, it is also the number of elements in the shape of the array:

```
RANK←⍴,⍴X
```

Further, since ⍴X is always a vector, the ravel is unnecessary, and the rank may be found from:

```
RANK←⍴⍴X
```

Example:

```
      ⍴⍴1 3 7 6 12 4
1

      X←2 3⍴1 3 7 6 12 4
      ⍴⍴X
2

      ⍴⍴5
```

ARRAY CREATION AND VARIABLES 3

Arrays are created by the APL interpreter by evaluating APL expressions. An APL expression is a syntactic construct of APL language elements which together totally detail the construction of an array.

Evaluation of an APL expression involves one of three processes within the interpreter, singly or in combination depending on the complexity of the APL expression.

1. APL language elements exist from which literal expressions may be formed. These are interpreted directly and result in arrays having the value as stated in the expression.
2. An expression may state a function to be called with designated arguments. The interpreter executes the function which in turn produces an array as its result.
3. An expression may reference a currently defined variable. Such reference results in the interpreter making available an array having the value of the one being referenced.

LITERAL EXPRESSIONS

Literal expressions allow explicitly valued scalars and vectors to be directly expressed.

LITERAL CHARACTER EXPRESSIONS

A character scalar is expressed by placing the desired character in quote marks, thus:

```
'A'
```

A character vector is expressed by placing zero, two, or more characters within quote marks.

`'AB'`	a 2-element character vector
`'ABCDE'`	a 5-element character vector
`''`	a 0-element character vector

To indicate that a character appearing within quote marks is the quote character itself, two consecutive quote marks are used to represent the single character.

`'DON''T'`	a 5-element character vector DON'T
`''''`	a character scalar '

LITERAL NUMERIC EXPRESSIONS

A numeric scalar is expressed by formulating a numeral from the 13 APL characters 0123456789.¯E

- Unsigned integer and decimal numerals are formed in the usual manner.
- A negative value is indicated with the negative symbol character '¯' (read as 'negative' or 'neg').
- The character E is used to convey base 10 exponentiation and can be read 'times 10 to the'.

```
¯6
3.14159625
4.325E17
2.59376E¯3      exponent must be an integer
10E¯5
.475            Note: embedded spaces
473                   are not allowed.
```

Numbers having any number of digits may be expressed, but the system will retain values of only 14 (in some cases 15) significant digits. Proper scaling will always take place.

Numbers conveying a magnitude exceeding the representation capability of CYBER computers will result in a DOMAIN ERROR. (See appendix C.)

A numeric vector is expressed by formulating a list of two or more numbers each separated by one or more space characters.

```
2.37  5493  ¯2.86E47
```

1-element vectors, 0-element numeric vectors and arrays of rank 2 or greater cannot be conveyed in a literal expression. Such structures can only be expressed by a call of a suitable function with appropriate arguments, or by referencing an existing array having such a shape.

VARIABLE DEFINITION: SPECIFICATION

The process of variable definition is called specification. The APL language syntax is:

<identifier> ← <APL expression>

In this process, a variable is created whose name is the identifier given, and whose value is the value of the array created by the APL expression.

Examples:

```
COUNT←1
```

The variable COUNT now has the value of the numeric scalar 1.

```
TEXT←'THIS IS IMPORTANT'
```

The variable TEXT now has the value of the character vector: *'THIS IS IMPORTANT'*

RULES FOR FORMING IDENTIFIERS

- Names may be from one to 112 characters in length.
- The first character must be an alphabetic character (A to Z, a to z, Δ or ∆).
- The remaining characters (if any) may be any alphabetic character or digit, or the underscore character (_).

REFERENCING VARIABLES

Whenever the identifier of a variable appears in an APL expression, it refers to that variable. On detecting the presence of a variable identifier, the APL interpreter makes available an array having the value of the variable being referenced.

If the variable has not been defined, a reference to it results in a VALUE ERROR.

```
A←2.3 ¯57.3 4E3
XΔ17a←A
```

In the first line above, A is specified as the variable identifier for the vector 2.3 ¯57.3 4E3. The appearance of A in the second line refers to the variable stated above. The reference makes available a vector 2.3 ¯57.3 4E3 which is then associated with a data identifier XΔ17a. Two variables now exist having the same value, one identified by A, the other by XΔ17a. Subsequent occurrences of A or XΔ17a in APL expressions refer to the corresponding variables.

RESPECIFICATION

If a new value is given to the variable A by means of a subsequent specification, for example:

```
A←'NEW A'
```

the previous value of A is no longer referenceable, and hence no longer exists. Note that there are no restrictions on the type or shape of the value newly specified to A. It need bear no relation as to type or shape of the previous value of A. A new specification for A in no way alters the specification for X△17a. It still is associated with vector 2.3 ¯57.3 4E3.

DISPLAYING DATA 4

SYNTAX

The APL language provides a facility for displaying data. The language syntax for conveying this process is:

□←<APL EXPRESSION>

The character □ is called QUAD. If the left-most operation indicated in an APL source line is other than a specification, display of the evaluated APL expression is implicit, and the construct □← need not be present in this case.

```
      □←2+2
4
```

```
      2+2
4
```

DATA OBJECT DISPLAYS

All data displays consist of a tabular arrangement of character representations of the elements of the array. For character data, each data element, being a character, is displayed as that character (or by the mnemonic for that character where it cannot be formed on the terminal being used; see appendix B).

Note that character arrays are displayed without enclosing quote marks:

```
      'A'
A
      'ABC'
ABC
```

Note that single quote marks are displayed as such:

```
      ''''
'
      'DON''T'
DON'T
```

For numeric data, each data element is represented by a suitable format of characters which together convey the value of the numeric element.

All displays begin at the left margin and element representations are displayed left to right in element order. Scalars are displayed in the same manner as a one-element vector.

Each rank 1 subarray display occupies at least one display line. If the number of characters required to display a complete rank 1 subarray exceeds WIDTH (see SYSTEM COMMANDS), its display will continue on subsequent lines with an appropriate indication of continuation (usually an indentation of 6 character positions). Each data element representation will be complete on one line.

Rank n-1 subarrays of rank n arrays are displayed in structure order.

Between subarrays of rank 2 and higher a blank line is displayed.

NUMERIC ELEMENT FORMATTING

The amount of significance used in formatting numeric arrays is controlled by an environmental parameter known as DIGITS. The normal setting for this parameter in APL*CYBER is 10. Numeric elements are formatted into one of two possible forms, decimal or exponential, depending on the value to be represented and on the setting of DIGITS. A rounded representation of the element value in the form of DIGITS digits is obtained, the left-most being non-zero unless the value is zero. Any value whose magnitude when rounded as above is less than 10 and not less than 0.001 will always be expressed in decimal form regardless of the setting of DIGITS.

Numeric Format Rules

- No more than DIGITS digits may be printed, unless they are leading zeros.
- No more than three leading zeros may be printed.

Decimal Form

[¯]<integral part>[.<fraction part>]

- Magnitude scaling is indicated by insertion of a decimal point after the appropriate digit position.
- If the magnitude of the element value is less than 1, the integral part is represented by a single zero.
- Trailing zeros in the fraction part are suppressed.
- If the fraction part is entirely zero, the decimal point is suppressed.
- Negative values are indicated with a leading negative symbol character ¯.

Examples:

```
      )DIGITS 4
WAS 10
      1.2348
1.235
      ¯42.927
¯42.93
      .123
0.123
```

```
      7.0004
7
      .0012365
0.001237
      ¯.0012365
¯0.001237
      .00099997
0.001
```

Exponential Form

In all cases where decimal form is unsuitable, exponential form is used.

<coefficient> E <exponent>

- The coefficient is formed from the DIGITS digits stated above for decimal form.
- A decimal point is inserted to the right of the left-most digit. Coefficients thus always have a magnitude less than 10 and greater than or equal to one.
- Suppression of trailing zeros and the decimal point, and use of the negative symbol are the same as for decimal form.
- The exponent is an integer with appropriate value to indicate proper scaling of the coefficent as formatted, with a leading negative sign if the exponent is negative.

Examples:

```
      )DIGITS 4
WAS 10
      12348
1.235E4
      ¯429273.8
¯4.293E5
```

```
      9999.5
1E4
```

NUMERIC DATA OBJECT FORMATTING

All numeric data objects are formatted as if they were matrices. A vector is formatted as a matrix with one rank 1 subarray. A scalar is formatted identically to a one- element vector. An array B of rank greater than two is treated as a restructured matrix B1 formed as follows:

```
B1←((×/¯1↓ρB),¯1↑ρB)ρB
```

Elements within each column of the above matrix are formatted uniformly as follows:

- The same element representation form (decimal or exponential) is used. Unless one or more elements must be formatted in exponential form, either by the criteria stated in numeric element formatting or as a consequence of the following formatting rules, decimal form will be used.
- Decimal points are aligned (i.e., occur in the same character position) for all element representations. This may entail appending one or more spaces to the left and one or more zeros (and decimal point) to the right of the fraction part as appropriate. If this causes a violation of the Numeric Format Rules stated above, exponential format is used for the column.

DISPLAYING NUMERIC DATA OBJECTS

Recall that all data objects consist of line displays of the rank 1 subsets in subset order, with element representations appearing left to right in element order beginning at the left margin.

Since all elements within each column of the numeric matrix are uniformly formatted and aligned, all such element representations will appear in vertically aligned and uniformly formatted columns, appearing left to right in matrix column order, with two blanks between adjacent columns.

Where displays are continued on indented lines, these should be visualized as additional columns that conceptually belong increasingly to the right of the display. See example on opposite page.

Example:

```
      )DIGITS 5
WAS 10
      )WIDTH 60
WAS 72
      X←.275396  14.3E3  692738  12345  678
      X
0.275396  14300  692738  12345  678
      )WIDTH 30
WAS 60
      X
0.275396  14300  692738  12345
      678                          (display continuation indented)

      Y←3 2ρ42 1.7E9 ¯173.52 6.8345E¯10 .9 0
      )WIDTH 60
WAS 30
      Y
    42.00  1.7000E9
  ¯173.52  6.8345E¯10
     0.90  0.0000E0
      )DIGITS 4
WAS 5
      Y
   4.200E1   1.700E9
  ¯1.735E2   6.835E¯10
   9.000E¯1  0.000E0
```

COMPOSITE DATA OBJECT DISPLAYS

Several evaluated expressions can be displayed in sequence in one composite display by arranging the expressions in desired display sequence and separating them with semicolons:

<expression>;<expression>;...;<expression>

Each APL expression is evaluated starting with the right-most and proceeding to the left-most.

If the display syntax □← occurs within the expression, the expression evaluated at that point is displayed immediately.

After the left-most expression is evaluated, a composite display is output for all those expressions set up for display in reverse order to that in which evaluated; i.e., in the left to right order in which the expressions appear on the line.

For consecutive displays of scalars or vectors, output is displayed contiguously on the same output line. Displays of expressions of higher rank are displayed in a vertical format. Continuation lines are indicated in the same manner as for a single display.

Both numeric and character expressions may be formatted in the same composite display. This feature provides the main use of composite displays. With this feature, result displays can be annotated with character descriptions in the style of an edited report.

Examples:

```
      QUANTITY←3
      UNIT_PRICE←1.50
      'COST OF ';QUANTITY;' UNITS IS ';QUANTITY×UNIT_PRICE
COST OF 3 UNITS IS 4.5

      5+□←⍳3;'ZXC';B←2 1 9;□←'XYZ';2 3⍴⍳6
XYZ                                    (first QUAD)
1  2  3                                (second QUAD)
6  7  8ZXC2  1  9XYZ
  1  2  3                              (composite display)
  4  5  6
```

PRIMITIVE FUNCTIONS 5

The basis of the APL language is a large set of predefined functions. Because they are part of the language, they are termed primitive functions.

NOTATION

The notation used in describing the syntax of APL constructs is as follows:

- The right argument of a function is indicated by the meta-identifier "B". It is understood that any valid APL expression may be used in place of this meta-identifier.
- The left argument of a function (if one exists) is indicated by the meta-identifier "A", as for "B" above.
- If the function produces a result, that fact is indicated by the meta-construct "*R*← ". It is understood that no actual specification of the result need take place.
- The function itself and any associated APL characters required are indicated by the symbols in question. These symbols must be used as shown.
- Function Indices (see INDEXED FUNCTIONS) are indicated by the meta-identifier "K" enclosed in square brackets following the function to be indexed. Any valid APL expression may be substituted for "K". If "K" is elided, the square brackets must also be elided.
- If a syntax involves a general primitive function, this function is represented by the meta-symbol "f". Any valid APL primitive function may be substituted for "f", subject to the restrictions specified in the case in question.
- If a second general primitive function is used in the syntax, it is represented by the meta-symbol "g", as for "f".
- Other syntactic constructs are indicated by a description of the construct enclosed in angular brackets (e.g., <index list>). It is understood that any syntactic construct following the rules specified in the case in question may be substituted for the meta-construct above.

Exceptions to this notation are indicated where they occur.

Example:

```
R←φ[X]B
```

In this example, the function "φ", modified by the function index "[K]", with right argument "B" and left argument "A", produces a result "R". Following this form, here is a possible usage of the above function:

```
      3φ[1]1 2 3 4
4 1 2 3
```

Since the result was not specified after completion, it was displayed.

SYNTAX

Primitive functions are of two types: monadic (i.e., having one argument), and dyadic (i.e., having two arguments). The syntax for calling each type is:

monadic:

<special APL character><argument expression>

dyadic:

<argument expression><special APL character><argument expression>

Most of the special APL characters used in designating monadic APL primitive functions are also used in designating some dyadic APL primitive function. In most cases, but not all, there is some similarity between the function procedure invoked in each case. The actual function called in each instance is, however, quite distinct.

DOMAIN AND RANGE

The class of arguments and the class of results of a given function are called its domain and range, respectively.

The domain for character arguments and the range for a character result is the APL character set.

The largest numeric class currently defined for APL*CYBER is the set of real numbers for which an exact or approximate representation exists on CYBER computers. Complex and other non-real number classes are not currently defined for any APL primitive functions.

Certain numeric arguments and results of function are confined to a subclass of the defined real numbers, namely the integers. Ordinals (see below) are members of this class.

Other numeric arguments and results of functions are confined to a subclass of the integers consisting of the integers 0 and 1. This subclass is known as the logical or Boolean class. (See Boolean numbers.)

Each of the foregoing classes is clearly a subclass of each class preceding it, and any function defined on a class clearly applies to any of its subclasses.

Any argument supplied to a function which is not in its domain of definition or for which the result is not in the defined range of definition results in a DOMAIN ERROR message.

ORDINALS

Ordinal numbers are the numbers used to state position or ranking in an ordered set. The names of these positions are first, second, third, etc.

It is customary to assign values to represent these positions identical to those used to represent the positive integers:

First	1
Second	2
Third	3

It is sometimes more convenient to assign the values as follows:

First	0
Second	1
Third	2

Once the value for first has been decided upon, second is assigned the next higher integer value, and so on.

The two schemes indicated are classified according to the value assigned for first, and are known respectively as ORIGIN 1 and ORIGIN 0.

The scheme to be followed can be designated by using the system command)ORIGIN (see ORIGIN command).

Various APL functions are defined which use ordinal arguments. Some others produce ordinal results.

The domain of definition of such functions for such arguments is the positive integers for ORIGIN 1 and the positive integers and zero for ORIGIN 0.

BOOLEAN NUMBERS

Boolean numbers are truth values and are usually defined for logical systems of two values as true and false. It is customary by convention, to represent the Boolean 'number' (i.e., truth value) true by the number 1 and false by 0.

This convention has been followed in the implementation of APL. The domain of definition of functions defined for such arguments and the range of those functions yielding such results are the numbers 1 and 0.

Such functions must be given arguments whose elements consist of the appropriate number of ones and zeros.

It should be understood that the meaning of a 1 or 0 is that of the truth value - true or false - when it is the argument of a Boolean function, regardless of the fact that it may be the result of some prior numeric computation.

CONFORMABILITY

As stated in the introduction, a key feature of APL is the fact that the primitive functions are defined for general arguments; i.e., the arguments are arrays, usually of more than one element, and the functions operate in a predefined manner on the array structure as a whole.

For most primitive functions there is some constraint placed on the generality of the argument(s). Any rule which limits the generality of shape of an acceptable argument of a function is called a conformability rule. Conformability rules are classified as either singular or dual.

- Singular Conformability: A conformability rule for a monadic function or one which pertains to a specific argument of a dyadic function independent of any shape for the other argument is said to be singular.
- Dual Conformability: A conformability rule for a dyadic function which states a relationship between the shapes of the two arguments is said to be dual. Certain dual conformability rules also implicity convey a singular conformability requirement for one of the arguments.

Conformability rules are stated as part of the description of each primitive function where one or more apply.

Violation of a conformability rule results in a RANK ERROR or LENGTH ERROR as appropriate unless overriding rules are applicable.

OVERRIDING CONFORMABILITY RULES

Conformability rules are subject to the following overriding rules, whereby a conformability rule may be relaxed or somewhat altered.

The following rules have precedence in the order listed.

1. The following singular conformability rules are inviolate:
   - A scalar cannot be indexed.
   - The left argument of DYADIC IOTA must be a vector.
2. A scalar is treated as a one-element vector where singular conformability requires a vector argument. This process is known as scalar extension.
3. A one-element vector is treated as a scalar where singular conformability requires a scalar argument.
4. Where a dual conformability rule exists, a scalar or one-element vector argument is treated for function execution as a restructured array having the minimum rank and number of elements required to meet all conformability requirements. This is another form of scalar extension. The restructured shape will not result in an empty data object unless that is specifically required.

   Exceptions:

   - The left argument of TAKE, DROP, EXPAND and TRANSPOSE, and the right argument of COMPRESS.

   This rule, when applied to INDEXED SPECIFICATION (q.v.), relates to the implied shape of the index list taken as a whole, and not to individual elements which make up the list. Note that this may relult in an indexed expression with bad form if multiple specification to the same indexed element is implied.

ORIGIN

In an ordered set, specific members are designated by an integer called an ordinal specifying the order position in the set. The ordinal of each member is one greater than the ordinal of its predecessor. The ORIGIN parameter is the value designated to the ordinal of the first member of the set. APL*CYBER allows the ORIGIN to be set to either 0 or 1.

The normal setting for ORIGIN in APL*CYBER is 1. To change the setting of ORIGIN, see the system command)ORIGIN.

The first element of the result returned by monadic IOTA (q.v.) is ORIGIN. Thus the setting of ORIGIN may be found from ⍳1:

```
      ⍳1
1
      )ORIGIN 0
WAS 1
      ⍳1
0
```

Since the ORIGIN designates the value of the ordinal of the first member of any set, any function that uses ordinals as an argument or returns ordinals as a result is said to be origin dependent.

Currently there are six primitive functions defined in APL that return ordinals as a result. These are:

1. monadic iota ⍳*B*
2. dyadic iota *A*⍳*B*
3. monadic query ?*B*
4. dyadic query *A*?*B*
5. grade up ⍋*B*
6. grade down ⍒*B*

The primitive function dyadic transpose requires the left argument to be a vector of ordinals.

dyadic transpose *A*⍉*B*

All forms of indexing employ ordinals as indices.

1. expression indexing A[B]
2. indexed specification A[B]←
3. indexed primitive functions f[K]B and Af[K]B

SUBARRAY OPERATIONS - INDEXED FUNCTIONS

Nearly all primitive functions in APL are defined for array arguments. In most cases, the basic operation is defined in terms of arrays of a specific structure, and extended to arrays of other structure by performing the operation in parallel on all basic subarrays of the array given.

All scalar functions are defined in terms of scalars. For higher rank arrays, the operation is carried out using corresponding scalar subarrays of the argument(s) (see SCALAR FUNCTIONS).

Many non-scalar functions are defined in terms of vectors (catenation, reduction, compression, etc.). If the array given is of lower rank, it is extended, if possible, in a manner appropriate to the function in question. If the array is of higher rank, the operation is carried out using vector subarrays of the argument(s).

In this case, however, the choice of the elements which constitute each subarray is non-trivial. For a rank N array, there are N possible coordinate axes along which the vector can be chosen.

In order to resolve this question, a Function Index is used. This takes the form of an index expression, enclosed in square brackets, following the function in question:

```
      R←f[K]B
or    R←Af[K]B
```

The index expression must evaluate to a one-element vector ordinal, designating the coordinate axis along which the vector subarray is to be chosen. From this it is apparent that for an index K, and an array of rank N, the domain of K is:

```
K∈⍳N
```

In most cases, if an index is not specified for the function, it defaults to the last coordinate axis, namely:

```
K0←(N-1)+⍳1
```

The functions which may be so indexed are:

/	Compress
\	Expand
f/	Reduction
⌽	Reverse, Rotate
dyadic ,	Catenate

For these functions, an alternate form exists in which the index defaults to the first coordinate axis, rather than the last:

```
K0←⍳1
```

The symbol for these functions is formed by overstriking the normal function symbol with a minus sign (-).

```
/   ⟶  ⌿
\   ⟶  ⍀
f/  ⟶  f⌿
⌽   ⟶  ⊖        (note disappearance of vertical bar)
⍪   ⟶  ⍮
```

Currently, these alternate forms may not be explicitly indexed.

RELATIVE FUZZ: USE IN RELATIONALS

In the comparison of any two numeric data elements the following three relational cases are always mutually exclusive:

```
A>B
A=B        A and B scalars
A<B
```

To consider A to be equal to B only when the internal representations of the argument are identical would be undesirable for the following reasons:

- Numbers in CYBER series computers can only be represented with 14 significant digits of accuracy (15 digits for integers with a magnitude less than 2*48).
- The deviation between the represented value and the exact value is proportional to the magnitude of the represented value.
- If successive operations are applied to such data elements, the inherent error in such represented values will propagate to the result such that the relative deviation from the theoretical result could be several times the initial relative deviation.
- Alternatively, the data initially supplied may be significant to much less than 14 digits even though internally represented as such.

For these reasons, it is usually desirable for numeric relational operations to be treated as follows:

- Consider A equal to B if A lies anywhere in the inclusive range B ± |B x factor.
- If A is smaller than the lower limit of this range, consider A to be less than B. Otherwise, consider A to be greater than B.

This is exactly how numeric relational operations are performed in APL. The factor used is called FUZZ. The range B x FUZZ is termed the relative FUZZ. Note that the range of the relative FUZZ is proportional to the magnitude of B. Thus, the relative FUZZ for a B of zero is zero.

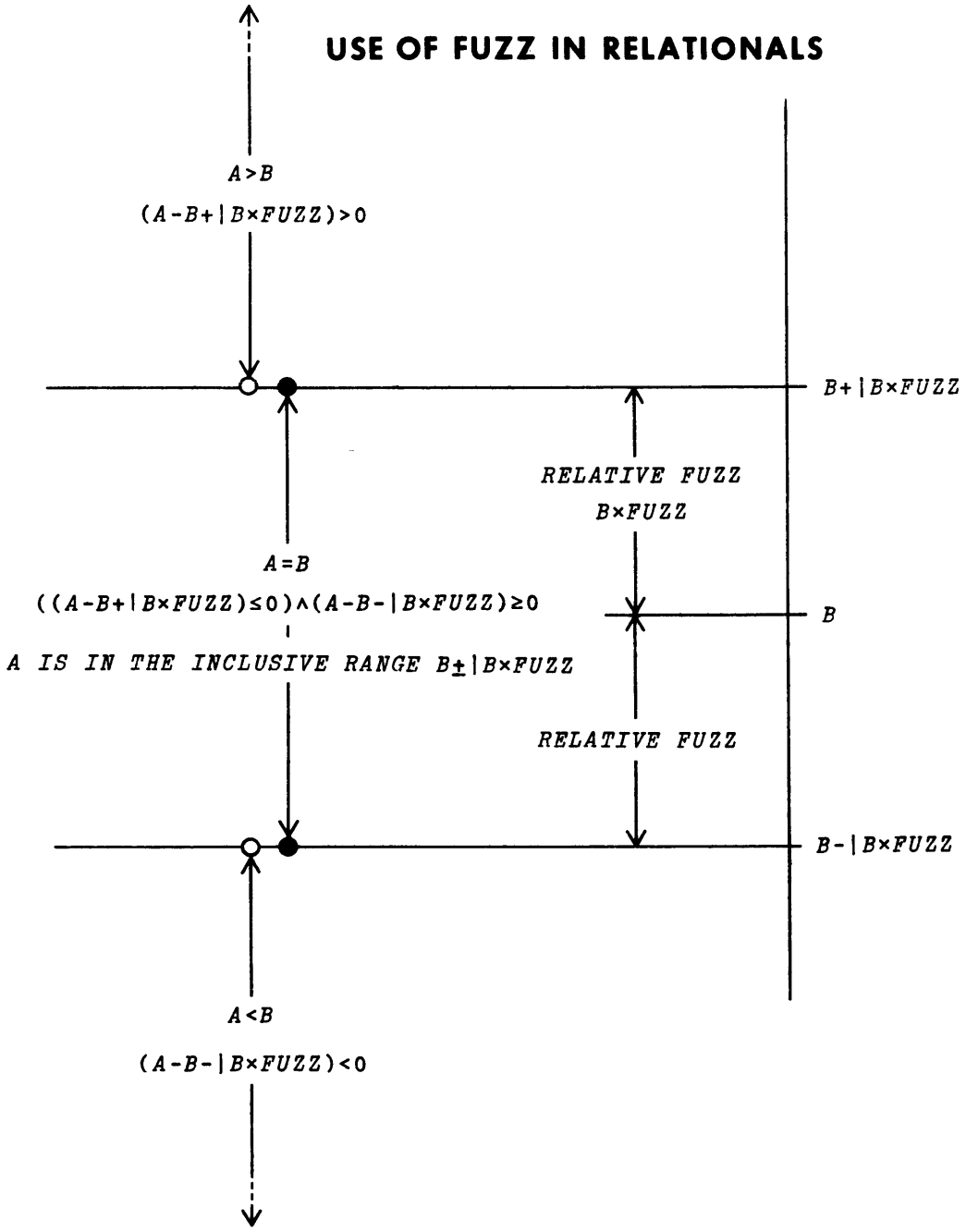
The following primitive functions also perform comparisons between data elements in the same manner as the relationals:

$A \in B$
$A \iota B$
} with numeric arguments

However, ⍋ and ⍒ do not use FUZZ.

The normal setting for FUZZ in APL*CYBER is ≈ 2*⁻43 (≈1.137E⁻13).

USE OF FUZZ IN RELATIONALS



NOTE: $A \geq B \leftrightarrow (A > B) \vee A = B$

$A \leq B \leftrightarrow (A < B) \vee A = B$

ABSOLUTE FUZZ

The following primitive functions use FUZZ itself (ABSOLUTE FUZZ) in determining their results.

FLOOR

Conceptually, FLOOR is a monadic function which returns the largest integer less than or equal to its argument.

In fact, FLOOR adds the value of FUZZ to the argument and then takes the conceptual FLOOR of that.

The conceptual FLOOR is the behaviour of FLOOR with FUZZ set to zero. Let $\underset{\sim}{\lfloor}$ represent the conceptual FLOOR. Then:

$$(\lfloor B) \leftrightarrow \underset{\sim}{\lfloor} B+FUZZ$$

CEILING

In a similar manner, ceiling operates as follows:

$$(\lceil B) \leftrightarrow \underset{\sim}{\lceil} B-FUZZ$$

INTEGER DOMAIN

Many APL primitive functions require integer arguments (Boolean and ordinal domains are subsets of the integer domain).

The test for acceptability as integer is:

$$((\lceil B)-\lfloor B)=0$$

If the above relationship is true, B is accepted as the integer $\lfloor B$. If the accepted integer is a member of the required domain no domain error report is issued.

Regardless of the setting of FUZZ all result values defined to be in integer domain will be represented exactly if their magnitude is less than 2*48.

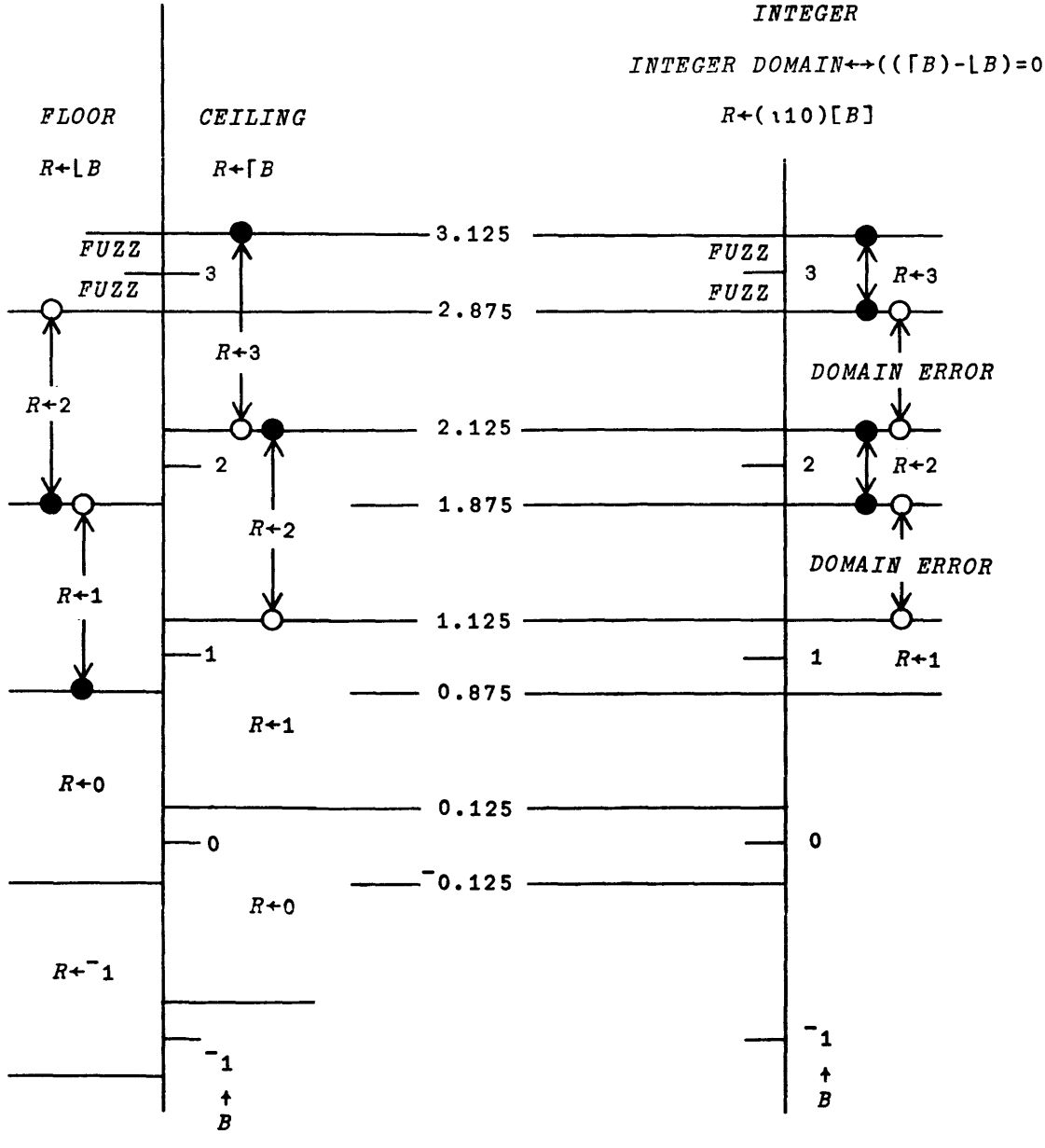
GENERAL NOTES

Note that for functions employing ABSOLUTE FUZZ, the fuzzing is of uniform width for all argument values and is based solely on the setting of FUZZ.

Also note that for such functions no acceptable setting of FUZZ has any effect on arguments greater than or equal to 2*48.

USE OF ABSOLUTE FUZZ

FOR THIS DESCRIPTION FUZZ=0.125



SEED

The functions ROLL and DEAL (q. v.) generate pseudo-random integers. Each element so produced is generated from an environmental parameter known as SEED. The algorithm used is such that a given combination of SEED and range (supplied by the argument(s)) produces a unique, predictable result element. However, the process of producing the element alters the value of SEED, so that the distribution of many elements produced sequentially is pseudo-random and flat.

Likewise, successive uses of these functions produce results which, while in fact completely determined, appear random and independent. Thus, "random" test sets may be reproduced by setting SEED to the same value prior to each test. To set this parameter, see ")SEED".

SELECTION PRIMITIVE FUNCTIONS 6

A SELECTION FUNCTION is one in which the result consists solely of elements supplied from the argument(s), and fill elements.

For certain selection operations, specifically TAKE and EXPAND, fill elements are required to create an array of the required shape from the argument given. For numeric arrays the fill element is zero, and for character arrays the fill element is the space (blank) character.

All selection functions are capable of operating on arrays of any data type, and produce a result of the same data type.

For dyadic selection functions other than CATENATE (q.v.), one argument (usually the right) is used to supply the array from which elements are to be selected, and the other to control the particular selection being performed. Unless otherwise specified, the domain of these control arguments is integer.

In general, restrictions on data type mentioned above or in the definition of the individual selection functions do not apply if the argument in question is empty.

INDEXING

syntax: $R\leftarrow B[\langle INDEX\ LIST\rangle]$

The index list is of the form:

$I_1;I_2;\ \ldots\ ;I_N$

Each index in the index list is separated from the next by a semicolon. Thus, there are N-1 semicolons in an index list of N indices.

conformability: $(\rho\rho B)=N$ (number of indices)

$(\rho\rho B)\geq 1$ (this restriction may not be circumvented by scalar extension)

result shape: $(\rho R)\leftarrow\rightarrow(\rho I_1),(\rho I_2),\ \ldots\ ,\rho I_N$

definition: The result is formed by selecting the subarray indicated by coordinates given in the index list. The positions selected along the jth coordinate axis are given in the index I_j. The portion of the shape imposed on the result by that coordinate selection is the shape of I_j.

If an index position selected does not exist in the array B, an INDEX ERROR results.

Indices may be elided. In this case, the index defaults to:

$I_J\leftarrow\iota(\rho B)[J]$

That is, all index positions along the jth coordinate are selected once, in position order.

Since the indices are ordinals, the result is ORIGIN dependent (see ORIGIN).

examples:

```
      X←4 3 7 5 8
      X[3]
7
      X[5 2]
8 3
      X[2 3⍴1 3 2 4 2 5]
4 7 3
5 3 8
      X[6]
INDEX ERROR
        ∨
      X[6]
      X[⍳0]
(blank)                          (result is empty)
      X←2 3⍴4 3 7 5 8 1
      X
4 3 7
5 8 1
      X[1]
RANK ERROR
      ∨
      X[1]
      X[1;1]
4
      X[2;1 2]
5 8
      X[2 1;3 2 3 1]
1 8 1 5
7 3 7 4
      X[;2]                      (first index elided)
3 8
      X[;,2]
3
8
      )ORIGIN 0
WAS 1
      X[1;1]
8
      X[0;0]
4
```

INDEXED SPECIFICATION

syntax: $R\leftarrow X[<INDEX\ LIST>]\leftarrow B$

The underlined portion of the syntax represents the indexed specification proper, while the remainder of the syntax is required for consistency with the definition of other primitive functions.

domain: B must be of the same data type as X.

conformability: $(\rho\rho X)=N$ where N is the number of items in the index list.

$(\rho\rho X)\geq 1$

$(\rho B)\leftrightarrow(\rho I_1).(\rho I_2), \ \ldots\ ,\rho I_N$

result shape: $(\rho R)\leftrightarrow\rho B$

definition: Rules pertaining to the index list are the same as for Indexing.

X must be an existing defined variable.

Indexed specification selectively replaces the elements of the array X indicated by the indices, with the elements of B corresponding to the positions in the implied index array. The operation has bad form and is not defined if multiple elements of B are specified to the same position in X.

When the specification is complete, the result (available as an argument to the next function) is the array B, not the array X or the indexed array X.

As with Indexing, the operation is ORIGIN dependent.

examples:

```
      X←1 2 3
      X[2]←5
      X
1 5 3
      X[1 2]←5 4
      X
5 4 3
      X[1 2]←1
      X                                  (scalar extension of B occurs)
1 1 3
      Y←X[1 2]←1
      Y                                  (result is B, not X)
1
      X←2 3⍴1 2 3 4 5 6
      X
1 2 3
4 5 6
      X[;2]←9 8
      X
1 9 3
4 8 6
      X[2]←9 8
RANK ERROR
      ∨
      X[2]←9 8
      X[;4]←9 8
INDEX ERROR
        ∨
      X[;4]←9 8
      )ORIGIN 0
WAS 1
      X
1 9 3
4 8 6
      X[;2]←4 5
      X
1 9 4
4 8 5
```

MONADIC RHO: SHAPE

syntax: *R←ρB*

result shape: The result is a vector with N elements, where N is the number of dimensions in the array B.

definition: The jth element of R is the length of the jth coordinate of B (see ARRAYS).

note 1: Although Shape is not a selection function, it is included here because it is integral to the discussion of selection functions.

note 2: The rank of an array is found by applying the Shape function twice.

```
RANKB←ρρB
```

For examples, see the section on ARRAYS.

MONADIC COMMA: RAVEL

syntax: *R*←,*B*

result shape: The result is a vector of N elements, where N is the number of elements in B.

definition: The result consists of the elements of B, selected from it in row major order. For further discussion, see ARRAYS.

examples:

```
      X
5
      ,X
5
      Y
1 4 7
      ;Y
1 4 7
      Z
1 3 2
7 8 4
      ,Z
1 3 2 7 8 4
      W
1 3
7 8

2 5
9 4
      ,W
1 3 7 8 2 5 9 4
```

DYADIC RHO: RESHAPE

syntax: $R\leftarrow A\rho B$

domain: $A\geq 0$ and integer

conformability: $(\rho A)\leq 127$ (APL*CYBER limitation) $(\rho\rho A)=1$

$0=\times/A$ if $0=\times/\rho B$

result shape: $(\rho R)\leftrightarrow A$

definition: If B is a vector, and the number of elements in the array indicated by dimensions A is exactly the number of elements of B, then the result is an array of shape A such that:

$(,R)\leftrightarrow B$

If the result requires N elements, and there are more than N elements in B, only the first N are used.

If there are insufficient elements in B to fill the array indicated by A, the elements are chosen cyclically from B until the array R is filled. This process is known as Cyclic Replication.

If B is not a vector, then:

$R\leftrightarrow A\rho,B$

identity: $(,B)\leftrightarrow(\times/\rho B)\rho B$

```
      X←2 3 8 1 4 7
      6ρX
2 3 8 1 4 7
      2 3ρX
2 3 8
1 4 7
      2 3 1ρX
2
3
8

1
4
7
      1ρX
2
```

```
      7ρX
2 3 8 1 4 7 2
      2 4ρX
2 3 8 1
4 7 2 3
      0ρX
(blank)                 (result is empty)
      (ι0)ρX
2                       (result is a scalar)
      2ρι0
DOMAIN ERROR
      v
      2ρι0              (A must be empty if
                              B is empty)
```

DYADIC COMMA: CATENATE

syntax: `R←A,[K]B`

`R←A⍪B` (reverse indexed)

domain: A and B must be of the same data type. K follows the rules for Function Indices (see INDEXED FUNCTIONS).

Three cases exist:

- `(⍴⍴A)=⍴⍴B`
- `(⍴⍴A)=1+⍴⍴B`

  In this case, B is treated as B1 obtained from:

  `RB1←(K≠⍳⍴⍴A)\⍴B` (See Expand)

  `RB1[K]←1`

  `B1←RB1⍴B`

- `(1+⍴⍴A)=⍴⍴B`

  This case is the mirror image of the above case. A is treated as A1 obtained from:

  `RA1←(K≠⍳⍴⍴B)\⍴A`

  `RA1[K]←1`

  `A1←RA1⍴A1`

In the discussion below, the first case only is considered. Behavior of the other two extend from the first via the above rules.

conformability: `((K≠⍳⍴⍴A)/⍴A)←→(K≠⍳⍴⍴B)/⍴B`

`(⍴⍴A)≥1`

`(⍴⍴B)≥1`

result shape: `(⍴R)←(¯1↑⍴A),(¯1↑⍴A)+¯1↑⍴B`

definition: If A and B are vectors of length M and N respectively, then the result R contains M+N elements, the first M of which are the elements of A, and last N are the elements of B.

If A and B are arrays of rank 2, vector subarrays are selected along the Kth coordinate axis, and catenated as above to form vectors along the Kth coordinate of the result.

Since K is an index, the result, if K is not elided, is ORIGIN dependent.

If ⍪ is used, the default coordinate is the first, rather than the last.

examples:

```
      2 3,4 5
 2 3 4 5
      X←2 3⍴1 2 3 4 5 6
      Y←2 3⍴7 8 9 10 11 12
      X
 1 2 3
 4 5 6
      Y
  7  8  9
 10 11 12
      X,Y
 1 2 3  7  8  9
 4 5 6 10 11 12
      X,[1]Y[1;]                    (first coordinate used)
 1 2 3
 4 5 6
 7 8 9
      X,[1]Y
 1  2  3
 4  5  6
 7  8  9
10 11 12
      X⍪Y
 1  2  3
 4  5  6
 7  8  9
10 11 12


      X,[0]Y
INDEX ERROR
        ∨
      X,[0]Y
      )ORIGIN 0
WAS 1
      X,[0]Y
 1  2  3
 4  5  6
 7  8  9
10 11 12
      X⍪Y
 1  2  3
 4  5  6
 7  8  9
10 11 12
      X,[1]Y
1 2 3  7  8  9
4 5 6 10 11 12
      X,Y
1 2 3  7  8  9
4 5 6 10 11 12
```

```
      )ORIGIN 1
WAS 0
      X,12 13
1 2 3 12
4 5 6 13
      X,[1]12 13
LENGTH ERROR
      ∨
      X,[1]12 13
      X,[1]12 13 14
 1  2  3
 4  5  6
12 13 14
      X,5
1 2 3 5                                (scalar 5 extended)
4 5 6 5
      Z←2 2 2⍴1 2 3 4 5 6 7 8
      ⍴⍴Z
3
      Z,1 2
RANK ERROR
      ∨
      Z,1 2                            (difference in ranks > 1)
```

TAKE

syntax: $R\leftarrow A\uparrow B$

conformability: $(\rho A)\leftrightarrow\rho\rho B$

This may not be circumvented by scalar extension unless $(\rho\rho B)=1$

result shape: $(\rho R)\leftarrow|A$ (see ABSOLUTE VALUE)

definition: Two cases exist:

- $(|A[I])\leq(\rho B)[I]$ ("ordinary" take)
- $(|A[I])>(\rho B)[I]$ ("too much" take)

"ORDINARY" TAKE

If B is a vector, and $A\geq 0$ the result is the first A elements of B. If $A<0$, the result is the last $|A$ elements of B.

If B is an array of rank ≥ 2, and $A[I]\geq 0$, the result is formed by selecting the first $A[I]$ positions along coordinate axis I. If $A[I]<0$, the last $|A[I]$ positions are selected.

"TOO MUCH" TAKE

If $A[I]\geq 0$, the elements of B occupy the first $A[I]$ positions along coordinate I of the result. If $A[I]<0$, the last $|A[I]$ positions are used.

When the selection is complete, fill elements are placed in any unoccupied positons of the result.

Take is <u>not</u> ORIGIN dependent.

(See examples on next page.)

examples:

```
      3↑1 2 3 4 5
1 2 3
      ¯3↑1 2 3 4 5
3 4 5
      X←3 4⍴⍳12
      X
1  2  3  4
5  6  7  8
9 10 11 12
      2 3↑X
1 2 3
5 6 7
      ¯2 ¯3↑X
 6  7  8
10 11 12
      2 ¯3↑X
2 3 4
6 7 8
      5↑1 2 3
1 2 3 0 0
      'D',¯4↑'ABC'
D ABC
      4 ¯5↑X
0 1  2  3  4
0 5  6  7  8
0 9 10 11 12
0 0  0  0  0
      2 3↑5
5 0 0                          (scalar extension)
0 0 0
```

DROP

syntax: `R←A↓B`

conformability: `(⍴A)=⍴⍴B`

This may not be circumvented by scalar extension, unless `(⍴⍴B)=1`

result shape: `(⍴R)←→(⍴B)-|A`

definition: Two cases exist:

- `(|A[I])≤(⍴B)[I]`
- `(|A[I])>(⍴B)[I]`

In this case, A is treated as if it were AI obtained from:

`A1←(×A)×(⍴B)⌊|A` (see signum, minimum)

If B is a vector, and `A≥0`, the result is all but the first A elements of B. If `A<0`, the result is all but the last `|A` elements.

If B is an array of rank `≥2`, and `A[I]≥0`, the result is formed by selecting all but the first `A[I]` positions along coordinate axis I of B. If `A[I]<0`, all but the last `|A[I]` positions are selected.

Drop is not ORIGIN dependent.

(See examples on next page.)

examples:

```
      2↓1 2 3 4 5
3 4 5
      ¯2↓1 2 3 4 5
1 2 3
      X←3 4⍴⍳12
      X
1  2  3  4
5  6  7  8
9 10 11 12
      ¯1 0↓X
1 2 3 4
5 6 7 8
      5↓1 2 3
(blank)                          (result is empty)
      5↓X
LENGTH ERROR
      ∨
      5↓X
      Y←5 1↓X
      Y
(blank)                          (Y is empty)
      ⍴Y
0 3
```

COMPRESS

syntax: `R←A/[K]B`

`R←A⌿B`

domain: A must be Boolean.

conformability: `(⍴⍴A)=1`

`(⍴⍴B)≥1`

`(⍴A)↔(⍴B)[K]`

result shape: `(⍴R)[I]= (⍴B)[I] FOR I≠K`

`+/A FOR I=K`

definition: If B is a vector, the result is formed by selecting `B[I]` if `A[I]=1`, or ignoring it if `A[I]=0`.

If B is an array of rank ≥2, the result is formed by using vector subarrays of B along the Kth coordinate axis.

Since K is an index, the result, if K is specified, is ORIGIN dependent. If `⌿` is used, the default coordinate axis is the first rather than the last.

(See examples on next page.)

examples:

```
      1 0 1/1 2 3
1 3
      1 0 1/'ABC'
AC
      1 1 1/1 2 3
1 2 3
      0 0 0/1 2 3
(blank)                                  (result is empty)


      1/1 2 3
1 2 3                                    (scalar extension of B)
      0/1 2 3
(blank)                                  (scalar extension of A)
      X←3 4⍴⍳12
      X
1  2  3  4
5  6  7  8
9 10 11 12
      1 0 1 0/X
1  3
5  7
9 11
      1 0 1/[1]X
1  2  3  4
9 10 11 12
      1 0 1⌿X
1  2  3  4
9 10 11 12
      )ORIGIN 0
WAS 1
      1 0 1⌿X
1  2  3  4
9 10 11 12
      1 0 1/[1]X
LENGTH ERROR
      ∨
      1 0 1/[1]X      ((⍴X)[1] is now 4, not 3)
```

EXPAND

syntax: $R\leftarrow A\backslash[K]B$

$R\leftarrow A\backslash B$

domain: A must be Boolean.

conformability: $(\rho\rho A)=1$

$(\rho\rho B)\geq 1$

$(\rho B)[K]=+/A$

result shape: $(\rho R)[I]=\begin{cases}(\rho B)[I] & \text{for } I\neq K\\ \rho A & \text{for } I=K\end{cases}$

definition: The result R is such that:

$$(A/[K]R)\leftrightarrow B$$

The positions of R defined by $(\sim A)/[K]R$ contain Fill elements.

Since K is an index, the result if an index is specified is ORIGIN dependent. If $\not$ is used, the index defaults to fhe first coordinate axis.

(See examples on next page.)

examples:

```
      1 0 1\1 2
1 0 2
      1 0 1 1\'ABC'
A BC
      1 1\1 2
1 2
      1 0 0\1 2
LENGTH ERROR
      ∨
      1 0 0\1 2          (B should have only 1 element)
      1 0 1\2
2 0 2                    (scalar extension of B)


      X←2 3⍴⍳6
      X
1 2 3
4 5 6
      1 0 1 1\X
1 0 2 3
4 0 5 6
      1 0 1\[1]X
1 2 3
0 0 0
4 5 6
      1 0 1⌿X
1 2 3
0 0 0
4 5 6
      )ORIGIN 0
WAS 1
      1 0 1⌿X
1 2 3
0 0 0
4 5 6
      1 0 1\[1]X
LENGTH ERROR
      ∨
      1 0 1\[1]X         ((⍴X)[1] is now 3, not 2)
```

MONADIC ROTATE: REVERSAL

syntax: $R\leftarrow\phi[K]B$

$R\leftarrow\ominus B$

conformability: $(\rho\rho B)\geq 1$

result shape: $(\rho R)\leftrightarrow\rho B$

definition: If B is a vector, the result is formed by selecting the elements of B in reverse order.

If B is an array of rank ≥ 2, the result is formed by reversing vectors selected along the Kth coordinate axis of B.

Since K is an index, the result if the index is specified is ORIGIN dependent. If $\ominus$ is used, the vectors are selected along the first coordinate axis of B.

identity: $(\phi[K]\phi[K]B)\leftrightarrow B$

examples:

```
      ϕ1 2 3
3 2 1

      X←2 3ρι6
      ϕX
3 2 1
6 5 4

      ϕ[1]X
4 5 6
1 2 3

      ⊖X
4 5 6
1 2 3

      )ORIGIN 0
WAS 1
      ⊖X
4 5 6
1 2 3

      ϕ[1]X
3 2 1
6 5 4
```

DYADIC ROTATE

syntax: `R←A⌽[K]B`

`R←A⊖B`

conformability: `(⍴⍴B)≥1`

`(⍴A)←→(K≠⍳⍴⍴B)/⍴B`

result shape: `(⍴R)←→⍴B`

definition: If B is a vector, and `A≥0`, the result is formed by cyclically rotating the elements of B, A positions to the left:

```
N←(⍴B)|A
R←(N↑B),N↓B
```

If `A<0`, the elements are cyclically rotated to the right:

```
N←-(⍴B)||A
R←(N↑B),N↓B
```

If B is an array of rank ≥2, the vectors to be rotated are selected along the Kth coordinate axis of B.

Each element of A specifies the rotation to be applied to the corresponding selected vector subarray of B.

Since K is an index, the result if the index is specified is ORIGIN dependent. If ⊖ is used, the index defaults to the first coordinate axis.

identity: `((-A)⌽[K]A⌽[K]B)←→B`

examples:

```
      2⍱1 2 3 4 5
3 4 5 1 2

      ¯2⍱1 2 3 4 5
4 5 1 2 3

      5⍱1 2 3 4 5
1 2 3 4 5

      ¯1⍱'AND'
DAN

      X←3 4⍴⍳12
      X
1  2  3  4
5  6  7  8
9 10 11 12
      0 1 2⍱X
 1  2 3  4
 6  7 8  5
11 12 9 10
      1⍱X
 2  3  4 1
 6  7  8 5
10 11 12 9
      1⍱[1]X
5  6  7  8
9 10 11 12
1  2  3  4
      1⍟X
5  6  7  8
9 10 11 12
1  2  3  4

      )ORIGIN 0
WAS 1
      1⍟X
5  6  7  8
9 10 11 12
1  2  3  4

      )ORIGIN 1
WAS 0
      1⍱[1]X
5  6  7  8
9 10 11 12
1  2  3  4

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

13 14 15 16
17 18 19 20
21 22 23 24
```

(scalar extension of A)

```
      P←2 3⍴¯2 ¯1 0 1 2 3
      P
¯2 ¯1 0
 1  2 3
      P⍱X
 3  4  1  2
 8  5  6  7
 9 10 11 12

14 15 16 13
19 20 17 18
24 21 22 23
```

MONADIC TRANSPOSE, DYADIC TRANSPOSE

dyadic syntax: $R\leftarrow A\lozenge B$

monadic syntax: $R\leftarrow\lozenge B$

(in this case, the left argument defaults to

$A\leftarrow(^{-}2\uparrow\rho B),\phi(-2\lfloor\rho\rho B)\uparrow\rho R$)

domain: $A\in\iota\rho\rho B$

conformability: $(\rho A)=\rho\rho B$

(this may not be overridden by scalar extension.)

Case 1: A has no repeated elements.

result shape: $(\rho R)\leftrightarrow(\rho B)[\blacklozenge A]$

The transpose operation simply reorders the coordinate axis of the argument as indicated by the left argument.

A useful rule-of-thumb for doing transpose operations is as follows: Write down the elements of ρB; below it write the elements of A; on a third line, place the elements of ρB in the position indicated by A. This line is then ρR.

Example: For the operation

```
3 1 2⍉4 5 6⍴⍳120
```

we write:

```
ρB:  4 5 6
 A:  3 1 2
     ─────
ρR:  5 6 4
```

The shape of the result is 5 6 4.

The effect of reordering the coordinates may be seen as follows:

```
      B←2 3⍴⍳6
      B
1 2 3
4 5 6
      R←⍉B
```

The elided left argument defaults to 2 1, so the shape of the result is 3 2.

The first coordinate has become the last, and the last has become the first. Thus, in the display the "rows" appear to have become "columns", and vice-versa.

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

Case 2: A has repeated elements,

domain: $A \in A \iota A$ (see DYADIC IOTA)

result shape: $(\rho R)[I]=\lfloor/(A=I)/\rho B \quad FOR\ ALL \quad I \in \iota \rho \rho R$

$(\rho \rho R)=0\ \ 1[1]+\lceil/A$ (see REDUCTION)

In the previous case, the transpose reordered the argument coordinate axes. Now, they are not only reordered but some of them are combined into a smaller set of new coordinate axes (as indicated by the rule-of-thumb given for Case 1).

The Ith coordinate axis of the result is formed from the coordinate axis $(A=I)/\iota\rho A$ of the argument array B. The resulting axis is the major diagonal of the axis from which it was formed. Only the elements along this axis are chosen for the result. The number of element positions along this axis is necessarily equal to the length of the shortest of the axes from which it was formed, i. e.,

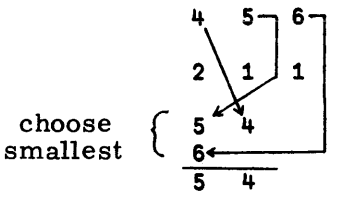
$$\lfloor/(\rho B)[(A=I)/\iota\rho A]$$

Since the left argument consists of coordinate axis indices, the result, if A is specified, is ORIGIN dependent.

For example, consider the operation

```
R←2 1 1⍉4 5 6⍴⍳120
```

Using the rule-of-thumb:



The shape of the result is 5 4.

The effect of combining coordinates may be seen as follows. Consider:

```
      B←3 3⍴⍳9
      R←1 1⌹B
```

The result is selected from the diagonal:

```
        R
1 5 9
        B
1 2 3
4 5 6
7 8 9
```

This is the classical trace of B.

identities: `(⌹⌹B)←→B`

For case 1 -

```
      ((⍋A)⌹A⌹B)←→B
```

If B is of rank ⩽ 1:

```
      (A⌹B)←→B     where A in this case must be ⍳1
```

examples:

```
        ⌹1 2 3
1 2 3                                  (A defaults to 1)

        X←2 2⍴'ABCD'
        X
AB
CD
        ⌹X
AC
BD

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

13 14 15 16
17 18 19 20
21 22 23 24

        ⌹X
 1  5  9                               (A defaults to 1 3 2)
 2  6 10
 3  7 11
 4  8 12

13 17 21
14 18 22
15 19 23
16 20 24
```

```
      3 2 1⍉X
 1 13
 5 17
 9 21

 2 14
 6 18
10 22

 3 15
 7 19
11 23

 4 16
 8 20
12 24

      1 2 2⍉X
 1  6 11
13 18 23
      2 3 3⍉X
DOMAIN ERROR
      ∨
      2 3 3⍉X                (⍺⌷A is 1 2 3)

      )ORIGIN 0
WAS 1
      1 2 2⍉X
DOMAIN ERROR
      ∨
      1 2 2⍉X                (⍺⌷A is 0 1 1)
      0 1 1⍉X
 1  6 11
13 18 23
      ⍉X
 1  5  9                     (A defaults to 0 2 1)
 2  6 10
 3  7 11
 4  8 12

13 17 21
14 18 22
15 19 23
16 20 24
```

SCALAR PRIMITIVE FUNCTIONS 7

GENERAL

The class of functions whose primary definition is in terms of operation on one or two scalars is called the SCALAR FUNCTIONS.

SCALAR MONADIC functions are defined in terms of a single scalar, while SCALAR DYADIC functions are defined in terms of a pair of scalars.

For all scalar functions, the following rules hold:

monadic syntax:	$R \leftarrow f B$	
dyadic syntax:	$R \leftarrow A f B$	
domain:	A and B must be numeric (unless otherwise specified).	
range:	R is numeric (unless otherwise specified). If R is outside the range of real numbers representable on the machine, a DOMAIN ERROR results. For APL*CYBER, this range is $^{-}1.265E322$ to $1.265E322$ (approximately).	
conformability:	$(\rho A) \leftrightarrow (\rho B)$	for scalar dyadics
result shape:	$(\rho R) \leftrightarrow \rho B$	for scalar monadics.
	$(\rho R) \leftrightarrow \rho B$ IF $(\rho\rho B) \geq \rho\rho A$ ρA IF $(\rho\rho B) < \rho\rho A$	for scalar dyadics

MONADIC DEFINITION

The result is formed by applying the function to each element of B, and placing the resulting element in the corresponding position in R.

DYADIC DEFINITION

The result is formed by applying the function to each element of B and the element in the corresponding position in A, and placing the resulting element in the corresponding position in R.

SCALAR MONADIC FUNCTIONS

MONADIC PLUS: IDENTITY

syntax: *R*←+*B*

definition: The result is the value of the argument.

examples:

```
      +23
23
      +¯1.5  2.7  1.7E¯3
¯1.5 2.7 0.0017
```

MONADIC MINUS: NEGATION

syntax: *R*←-*B*

definition: The result is the negated value of the argument.

examples:

```
      -23
¯23
      -¯1.5  2.7  1.7E¯3  0
1.5 ¯2.7 ¯0.0017 0
```

MONADIC MULTIPLY: SIGNUM

syntax: *R*←×*B*

definition: The result is ¯1, 0 or 1 depending on whether the argument is negative, zero or positive.

examples:

```
      ×23
1
      ×¯1.5  2.7  1E¯3  0
¯1 1 1 0
```

MONADIC DIVIDE: RECIPROCAL

syntax: *R*←÷*B*

domain: *B*≠0

definition: The result is the reciprocal of the argument.

examples:

```
      ÷5
0.2
      ÷¯10  .5E3
¯0.1 0.002
      ÷0
DOMAIN ERROR
      v
      ÷0
```

MONADIC POWER: EXPONENTIAL

syntax: *R*←***B*

definition: The result is the exponential of (e to the power of) the argument. e is approximated by 2.718281828459045.

examples:

```
      *1
2.718281828
      *1.5  0
4.48168907 1
      *1E8
DOMAIN ERROR
      v
      *100000000          (result outside machine range)
```

MONADIC LOGARITHM: NATURAL LOG

syntax: *R*←⍟*B*

domain: *B*>0

The natural logarithm function is the inverse of the exponential function.

examples:

```
      ⍟*1
1
      ⍟1
0
      ⍟¯1
DOMAIN ERROR
      ⍟¯1
      ∨
```

MONADIC MINIMUM: FLOOR

syntax: *R*←⌊*B*

definition: The result is the greatest integer less than or equal to the argument. The result of this function is dependent on the setting of FUZZ.

examples:

```
      ⌊1.5
1
      ⌊¯1.5 3 ¯5 ¯4.1 ¯4.9 5.1 5.9
¯2 3 ¯5 ¯5 ¯5 5 5
      ⌊1-0.1 1E¯15
0 1                                   (second element within FUZZ of 1)
```

MONADIC MAXIMUM: CEILING

syntax: *R*←⌈*B*

definition: The result is the smallest integer greater than or equal to the argument. The result of this function is dependent on the setting of FUZZ.

examples:

```
      ⌈1.5
2
      ⌈¯5 3 ¯5 ¯4.1 ¯4.9 5.1 5.9
¯5 3 ¯5 ¯4 ¯4 6 6
```

MONADIC MODULUS: ABSOLUTE VALUE

syntax: *R*←|*B*

defintion: The result is the absolute value of the argument.

examples:

```
      |¯1.5
1.5
      |¯3 0 15
3 0 15
```

MONADIC CIRCLE: PI TIMES

syntax: *R*←○*B*

definition: The result is ⍱times the value of the argument. ⍱ is represented as approximately 3.14159265358979.

examples:

```
      ○1
3.141592654
      ○75.3÷180
1.314232927                    (number of radians in 75.3 degrees)
```

FACTORIAL

syntax: *R←!B*

domain: If B < 0, B must not be integer.

definition: The result is obtained from applying the Gamma function to the elements of B as follows:

R←GAMMA B+1

Note that if B is a non-negative integer, the result is that of the classical factorial function.

```
      !3
6
      !0 1 2 3 4
1 1 2 6 24
      !¯0.5 3.7 10
1.772453851 15.4314116 3628800
```

MONADIC QUERY: ROLL

syntax: *R←?B*

domain: B must be a positive integer.

definition: The result is an integer pseudo-randomly selected from integers ⍳B. The roll function result is dependent on the settings of SEED and ORIGIN.

examples:

```
      ?5
 2
      ?5 5 5 5 5 5 5
 3 1 4 2 1 5 4
      ?1
 1                                  (the setting of ORIGIN)
      )ORIGIN 0
 WAS 1
      ?1
 0
```

MONADIC TILDE: NOT

syntax: *R←~B*

domain: B must be Boolean.

range: R is Boolean.

definition: The result is a 1 if the argument is zero, otherwise the result is zero.

examples:

```
      ~0
1

      ~1 1 0 1 0
0 0 1 0 1

      ~0.5
DOMAIN ERROR
       v
      ~0.5                    (argument not Boolean)
```

SCALAR DYADIC FUNCTIONS

DYADIC PLUS: ADDITION

syntax: *R←A+B*

definition: The result is A plus B.

examples:

```
      2+3
5
      1 2+¯1 5
0 7
      ¯2+6 7 4.5
4 5 2.5
```

(scalar extension of A)

DYADIC MINUS: SUBTRACTION

syntax: *R←A-B*

definition: The result is A minus B.

examples:

```
      2-3
¯1
      1 15 12-10
¯9 5 2
```

(scalar extension of B)

DYADIC MULTIPLY

syntax: *R←A×B*

definition: The result is A times B.

examples:

```
      2×3
6
      1 10 100×1 2 3
1 20 300
      1E200×1E200
DOMAIN ERROR
           ∨
      1E200×1E200          (result outside machine range)
```

DYADIC DIVIDE

syntax: *R←A÷B*

domain: *B≠0*

definition: The result is A divided by B.

examples:

```
      2÷3
0.666666667
      2 3 4÷4 3 2
0.5 1 2
      0÷0
DOMAIN ERROR
        ∨
      0÷0                  (B must be non-zero)
```

DYADIC MODULUS: RESIDUE

syntax: $R \leftarrow A \mid B$

range: $R \geq 0$

$IF\ A \neq 0, R < |A$

definition: If Q is the largest integer such that:

$$B \geq Q \times A$$

then: $R \leftarrow B - Q \times A$

This result may be expressed:

$$R \leftarrow B - (|A) \times \lfloor B \div |A \quad FOR \quad A \neq 0$$

$$R \leftarrow B \quad FOR \quad A = 0$$

Note that in the case A = 0, B < 0, no non-negative solution for R exists, and a DOMAIN ERROR results.

examples:

```
      10|15.3
5.3
      1|12.34 10 1.5
0.34 0 .5                    (fractional part of B)
      2|¯1 0 1 2 3 4
1 0 1 0 1 0                  (result 1 of B is odd)
      0|5
5
      0|¯5
DOMAIN ERROR
        ∨
      0|¯5                   (negative result not allowed)
```

DYADIC POWER

syntax: *R←A*B*

definition: The result is A raised to the power B. Note that if A is negative and B is not an integer, the result is not real, and a DOMAIN ERROR results.

In APL*CYBER, if A is negative, B must be a positive integer.

examples:

```
      2*3
8
      ¯10*¯1 0 1 2
DOMAIN ERROR
      v
      ¯10*¯1 0 1 2
      0.01 2 4 9*0.5
0.1 1.414213562 2 3          (square root of A)
      0.001 1 8 27*÷3
0.1 1 2 3                    (cube root of A)
      ¯1*0.5
DOMAIN ERROR
      v
      ¯1*0.5                 (if A is negative, B must be integer)
      0*0
1
```

DYADIC LOGARITHM

syntax: *R←A●B*

domain: *A*>0,*A*≠1

B>0

definition: The result is the logarithm of B in base A.

identity: *B←A***A●B*

examples:

```
      2●3
1.584962501
      10●0.1 1 10 1E2
¯1 0 1 2
      10●*1
0.4342944819                (common log of e)
      0●0
DOMAIN ERROR
        ∨
      0●0                   (A and B must be positive)
      1●1
DOMAIN ERROR
      ∨
      1●1                   (A must not be 1)
```

DYADIC MAXIMUM

syntax: *R←A⌈B*

definition: The result is the largest of A or B.

examples:

```
      2⌈3
3
      1 3 5⌈¯2 7 4
1 7 5
      0⌈¯3.5 0 1 5.2
0 0 1 5.2
```

DYADIC MINIMUM

syntax: *R←A⌊B*

definition: The result is the smaller of A and B.

examples:

```
      2⌊3
2
      1 3 5⌊¯2 7 4
¯2 3 4
      0⌊¯1 0 1 2
¯1 0 0 0
```

DYADIC CIRCLE

syntax: $R \leftarrow A \circ B$

domain: A must be integer, $A \leq 7, A \geq ^{-}7$

definition: The result is the trigonometric function of B indicated by A. The "normal" trigonometric functions are assigned to positive values of A, while their "inverse" is designated by the corresponding negative value of A.

The domain of the "inverse" functions is usually the range of the "normal" function. The possible values of A and their corresponding functions are listed below, along with their range and domain.

A	Function	Domain	Range	A	Function	Domain	Range
0	(1-B*2)*0.5	¯1 thru 1	0 thru 1*				
1	sin B		¯1 thru 1	¯1	arc sin B	¯1 thru 1	$-\frac{\pi}{2}$ thru $\frac{\pi}{2}$
2	cos B		¯1 thru 1	¯2	arc cos B	¯1 thru 1	0 thru π
3	tan B			¯3	arc tan B		$-\frac{\pi}{2}$ thru $\frac{\pi}{2}$
4	(1+B*2)*0.5		1 thru ∞	¯4	(1+B*2)*0.5	¯∞ thru ¯1 1 thru ∞	0 thru ∞
5	sinh B			¯5	arc sinh B		
6	cosh B		1 thru ∞	¯6	arc cosh B	1 thru ∞	0 thru ∞
7	tanh B		¯1 thru 1	¯7	arc tanh B	¯1 thru 1	

identity: $((-A) \circ A \circ B) \leftrightarrow B$ for $B > 0$

examples:

```
      2○3
¯0.9899924966
```
(cosine of 3 radians)

```
      1 2 3○○ 0.25 0.5 0.75
0.7071067812  0  ¯1
```
($\sin\frac{\pi}{4}$, $\cos\frac{\pi}{2}$, $\tan\frac{3\pi}{4}$)

```
      ¯5○2.3
1.570278543
```
(inverse hyperbolic sine of B)

```
      3○○0.5
DOMAIN ERROR
        ∨
      3○○0.5
```
($\tan\frac{\pi}{2}$ is infinite)

COMBINATION

syntax: `R←A!B`

definition: The result is obtained by applying the factorial function to the arguments as follows:

```
R←(!B)÷(!A)×!B-A
```

For $A \geq 0$ and $B \geq A$, the result may be expressed in terms of the Beta function:

```
R←÷(B-A)×(A+1) BETA B-A
```

If A and B are integers, the result is the number of combinations which can be made from B things taken A at a time. In this case, if $A > B$, the result can be seen to be zero.

examples:

```
      2!3
3
      2 4 6!6 4 2
15 1 0
      2.5 ¯2!7.3 ¯5
32.61766703 0
```

EQUAL, NOT EQUAL

syntax: `R←A=B` (equal)

`R←A≠B` (not equal)

domain: No restriction.

range: R is Boolean.

definition: Equal the result is one if A is equal to B, otherwise the result is zero. If A and B are numeric, the result is FUZZ dependent.

Not equal `R←~A=B`

examples:

```
      2=3
0

      'A+1'='A+4'
1 1 0

      2='A'
0

      3.5 0 1=1 0 1
0 1 1

      1=1+1E¯15
1                          (A is within relative FUZZ of B)

      0≠1E¯15
1                          (relative FUZZ of zero is zero)

      0 1 0 1≠0 1 1 0
0 0 1 1                    (exclusive OR of A and B)
```

```
      2≠3
1

      'A+1'≠'A++'
0 0 1

      2≠'A'
1

      3.5 0 1≠1 0 1
1 0 0
```

OTHER RELATIONALS

syntax:

```
R←A<B          (less than)
R←A≤B          (less than or equal)
R←A≥B          (greater than or equal)
R←A>B          (greater than)
```

range: R is Boolean.

definition: Less than - the result is 1 if A is less than B, otherwise it is 0.

Greater than - the result is 1 if A is greater than B, otherwise it is 0.

Greater than or equal - `R←~A<B`

Less than or equal - `R←~A>B`

The results of these functions are FUZZ dependent.

examples:

```
      2<3 2
1 0
      2>3 2
0 0
      2<'A'
DOMAIN ERROR
        ∨
      2<'A'          (A and B must be numeric)
      2≥3 2
0 1
      2≤3 2
1 1
```

BOOLEAN FUNCTIONS

syntax:

$R \leftarrow A \wedge B$	(and)
$R \leftarrow A \vee B$	(or)
$R \leftarrow A \barwedge B$	(Nand)
$R \leftarrow A \barvee B$	(Nor)

domain: A and B must be Boolean.

range: R is Boolean.

definition: And - the result is 1 if both A and B are 1, otherwise it is 0.

Or - the result is 1 if either A or B is 1, otherwise it is 0.

Nand - $R \leftarrow \sim A \wedge B$

Nor - $R \leftarrow \sim A \vee B$

examples:

```
      1^0 1
0 1

      1v0 1
1 1

      1⍲0 1
1 0

      1⍱0 1
0 0
```

COMPOSITE FUNCTIONS 8

Composite functions are formed from more than one APL symbol. They involve one or more primitive functions and one or more modifiers called OPERATORS.

The three composite functions currently defined in APL are REDUCTION, INNER PRODUCT, and OUTER PRODUCT.

In Reduction, the form of the composite function is "f/", where "/" is used as an operator to modify the function "f". In Inner Product, the form of the composite function is "f. g", where ". " is used as an operator to form a compound of the function "f" and "g". In Outer Product, the form of the composite function is "∘.f", where the pair "∘." is used to modify the function "f".

In all three forms, the functions which may be used are the scalar dyadic primitive functions. The domain and range of the composite operators is that implied by the functions used.

REDUCTION

syntax:

```
R←f/[K]B
R← f ⌿B
```

where f is a scalar dyadic primitive function.

The index [K] follows the rules for Indexed Functions.

conformability: `(⍴⍴B)≥1`

result shape: `(⍴R)←→(~(⍳⍴⍴B)∈K)/⍴B`

definition: If B is a vector, the result R is a scalar formed from the distributed operation of the function f on the elements of B as follows:

```
R←B[1] f B[2] f ...  f B[⍴B]
```

Note that if an identity element value I exists for the function, such that:

```
(B f I)←→B
```

or `(I f B)←→B`

for any one-element B, then the result expression can be written:

```
R←B[1]f...B[⍴B]fIfI...
```

or `R←IfI f B[1]f...B[⍴B]`

Thus if B has only one element, and that element is in the range of f, it is apparent that:

```
R←B[1]fIfI...(OR IfIfB[1])
```

so `R←→B[1]`

This is in fact true even if such an I does not exist.

Likewise, if B is empty, then:

```
R←IfIfI...
```

so `R←I`

if I exists and is in the range of f.

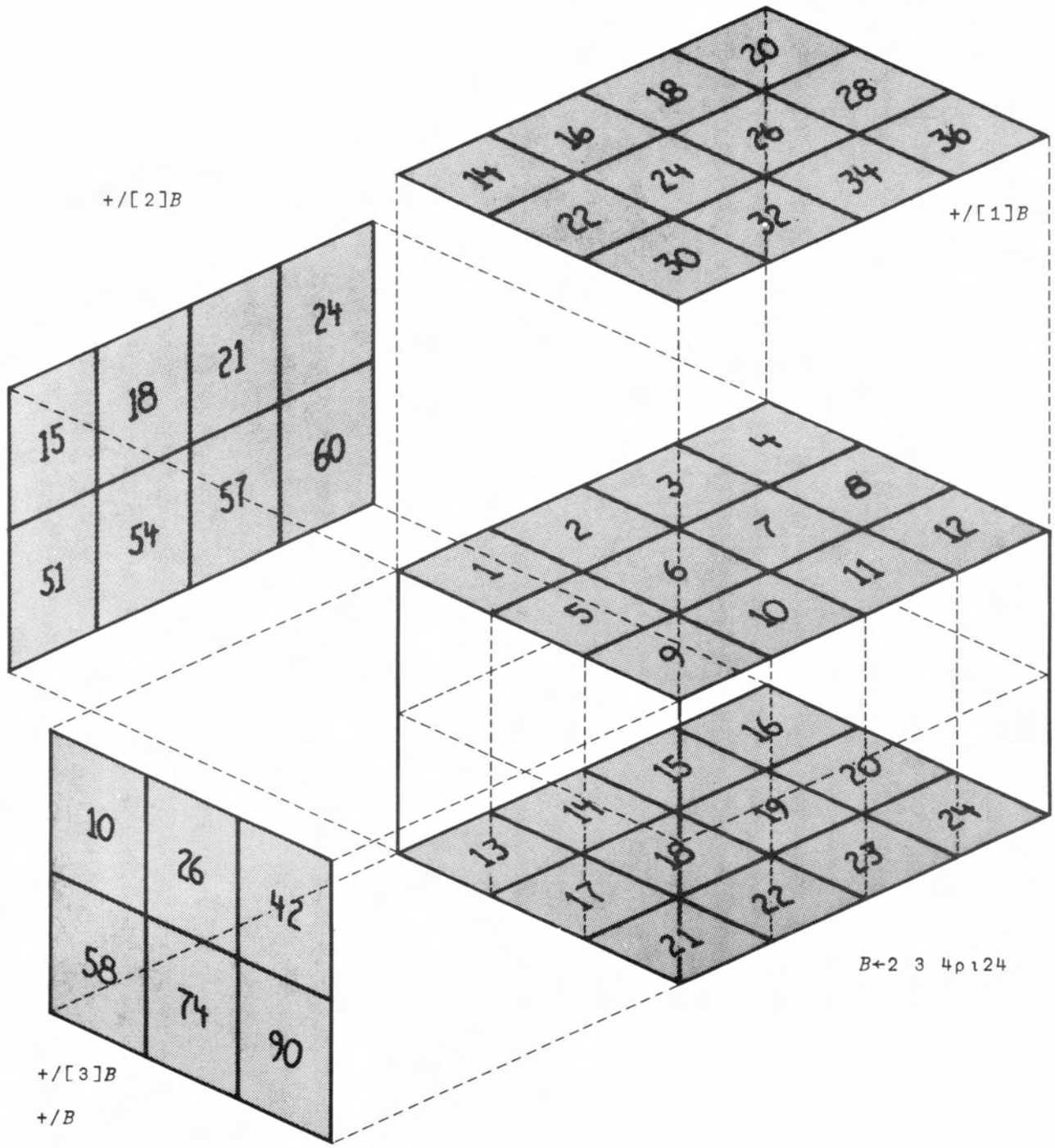
For non-commutative functions, an identity element, if it exists, may be only a left or right identity. The scalar functions and their respective identity elements are given in the table below:

Function	Identity Element	Comments
+	0	
-	0	right identity
×	1	
÷	1	right identity
⌈	≈¯1.265*E*322	smallest representable number
⌊	≈1.265*E*322	largest representable number
|	0	left identity
*	1	right identity
⍟	-	no identity
○	-	no identity
!	1	left identity
=	1	Boolean only
≠	0	Boolean only
>	0	Boolean right identity
<	0	Boolean left identity
≥	1	Boolean right identity
≤	1	Boolean left identity
∧	1	
∨	0	
⍲	–	no identity
⍱	–	no identity

For B of rank greater than 1, the operation is performed on vector subarrays of B as indicated by the index K. Since K is an index, the result, if an index is specified, is ORIGIN dependent. If f ⌿ is used, the index defaults to that of the first coordinate axis.

examples:

```
      +/1 3 4
8
      -/1 4 3
0
      X←2 3⍴1 3 4 6 2 5
      X
1 3 4
6 2 5

      +/X
8 13

      +/[1]X
7 5 9

      )ORIGIN 0
WAS 1
      +/[1]X
8 13

      +⌿X
7 5 9

      ⌈⌿X
6 3 5

      +/3 0⍴1
0 0 0

      ⍴X
2 3
      ×/⍴X
6

      ⍳/X
SYNTAX ERROR
      ⍳/X
       ∨

      ⌊/'A'
DOMAIN ERROR
      ⌊/'A'
        ∨
```


+/[2]B
+/[1]B
B←2 3 4⍴⍳24
+/[3]B
+/B

INNER PRODUCT

syntax: `R←A f . g B`

where f and g are scalar dyadic primitive functions.

conformability:

```
(⍴⍴A)≥1
(⍴⍴B)≥1
((⍴⍴A)+⍴⍴B)≤129        (APL*CYBER restriction)
(¯1↑⍴A)=1↑⍴B
```

extended conformability: A scalar or one element array may be used for either argument, in which case the last restriction above is not required.

result shape: `(⍴R)←→(¯1↓⍴A),1↓⍴B`

definition: If A and B are vectors, the result is obtained from:

```
R←f/A g B
```

If either A or B is of rank ≥2, the operation is carried out using vector subarrays of the argument in question. Subarrays from A are selected along the last coordinate axis, and subarrays of B are selected along the first coordinate axis.

Furthermore, for each vector subarray in A, the operation is carried out for all subarrays in B, in a fashion similar to Outer Product (q. v.).

identity: Let A1, B1 and RR be defined by:

```
A1←((×/¯1↓⍴A),¯1↑⍴A)⍴A
B1←((1↑⍴B),×/1↓⍴B)⍴B
RR←(¯1↓⍴A),1↓⍴B
```

Note that A1 and B1 are matrices. Then for all A and B:

```
(A f . g B)←→RR⍴A1 f . g B1
```

examples:

```
      1 2 3+.×10 1 0.1
12.3
      X←2 3⍴⍳6
      Y←3 2⍴10 4 1 5 0.1 6
      X
1 2 3
4 5 6
      Y
10.0 4
 1.0 5
 0.1 6
      X+.×Y
12.3 32
45.6 77
      PX←3 7 1
      XQ←4 2 7
      PX⌈.+XQ
9
      X←○ 0.25
      N←10
      (X*M)-.÷!M←2×¯1+⍳N
0.7071067812
      (1 1 1⍴1)+.×2 3⍴⍳6   (extended conformability)
5 7 9                      (result shape 1 1 3)
```

OUTER PRODUCT

syntax: $R\leftarrow A\circ.f\ B$

where f is a scalar dyadic primitive function.

conformability: $((\rho\rho A)+\rho\rho B)\leq 127$ (APL*CYBER restriction)

result shape: $(\rho R)\leftarrow(\rho A),\rho B$

definition: If A is a scalar, the result is:

$R\leftarrow A\ f\ B$

For A of rank ≥ 1, the result is formed by performing the above operation for each element (i. e., scalar subarray) in A, and placing the resulting array in the subarray position of R corresponding to the position of the element in A.

examples:

```
      2∘.+1 2 3
3 4 5

      1 2∘.+1 2 3
2 3 4
3 4 5

      2 10∘.*¯1 0 1 2 3
0.5 1  2   4    8
0.1 1 10 100 1000

      X←2 3 ⍴100×⍳6
      X
100 200 300
400 500 600

      Y←2 3⍴⍳6
      Y
1 2 3
4 5 6
```

```
      Z←X∘.+Y
      ρZ
2 3 2 3
      Z
101 102 103
104 105 106

201 202 203
204 205 206

301 302 303
304 305 306


401 402 403
404 405 406

501 502 503
504 505 506

601 602 603
604 605 606
```

MONADIC IOTA: INTERVAL

syntax: $R \leftarrow \iota B$

domain: B must be integer.
$B \geq 0$

conformability: $(\rho\rho B)=0$

result shape: $(\rho R) \leftrightarrow ,B$

definition: The result is a vector of the first B ordinals.

examples:

```
      ⍳5
1 2 3 4 5
      ⍳1
1                          (the setting of ORIGIN)
      ⍳0                   (the result is empty)

      ⍴⍳0
0
      )ORIGIN 0
WAS 1
      ⍳5
0 1 2 3 4
      ⍳1
0
      ⍳0

      ⍴⍳0
0
      ⍳1 2
LENGTH ERROR
       ∨
      ⍳1 2                 (B must be a scalar)
```

DYADIC IOTA: INDEX OF

syntax: `R←A⍳B`

domain: No restriction.

range: Ordinal.

conformability: `(⍴⍴A)←→1`
(Note: this requirement cannot be overridden by scalar extension.)

result shape: `(⍴R)←→⍴B`

definition: The result has the shape of B. For each element of B, the corresponding result is the lowest index of A which selects a match for that element in A, if one exists. If no matching A element exists, the result element is assigned the value `(⍴A)+⍳1` (i.e., one greater than the highest valid index for A).

- Since the elements of the result are indices, the result is ORIGIN dependent (see ORIGIN).
- If no element of A matches any element of B, for A not empty:

  `R←→(⍴B)⍴(⍴A)+⍳1`

- If A is empty, `R←(⍴B)⍴⍳1`
- For A and B both numeric, element comparisons are subject to the setting of FUZZ (see FUZZ).
- If `∧/,B∈A` then `B←→A[R]`

(See examples on next page.)

examples:

```
      )ORIGIN
1
      2 1 5 7⍳5
3
      'ABCD'⍳'B'
2
      2 1 5 7⍳6
5
      'ABCD'⍳'F'
5
      4 7 9⍳2 7 4 3
4 2 1 4
      'WXYZ'⍳1 2 3
5 5 5
      7⍳3
RANK ERROR
      ⍴
      7⍳3          (left argument must be a vector)
      (,7)⍳3
2
      (⍳0)⍳3 5 1
1 1 1
      1 3 5 7 9⍳3 3⍴⍳9
1 6 2
6 3 6
4 6 5
      ⍴2 1 9⍳⍳0
0                  (recall (⍴R)←→⍴B )
```

DYADIC EPSILON: MEMBERSHIP

syntax: $R\leftarrow A\in B$

domain: no restriction

range: Boolean.

conformability: None.

result shape: $(\rho R)\leftrightarrow\rho A$

definition: The result has the shape of A. For each element of A, the corresponding result element is a one if that element of A exists in B; otherwise it is a zero.

$R\leftarrow\rightarrow\vee/A\circ.=,B$

note: For A and B both numeric, element comparisons are subject to the setting of FUZZ (see FUZZ).

examples:

```
      2∈⍳7
1

      8∈⍳7
0

      A←2 9 7 3 4
      B←6 1 2 4
      A∈B
1 0 0 0 1
      B∈A
0 0 1 1
      2 3∈⍳0
0 0
```

```
      1 2 3∈'AXVR2'
0 0 0
      ⍴(⍳0)∈1 2 7
0
      'XAYQ3B7'∈'ABC3'
0 1 0 0 1 1 0
      'ABC3'∈'XAYQ3B7'
1 1 0 1
      (2 3⍴⍳6)∈2 6 9
0 1 0
0 0 1
```

DYADIC QUERY: DEAL

syntax: $R \leftarrow A ? B$

domain: A and B both integer: $A \geq 0$, $B \geq A$

range: Ordinal.

conformability: $(0=\rho\rho A) \wedge 0=\rho\rho B$

result shape: $(\rho R) \leftrightarrow , A$

definition: The result R is a vector of A elements of ιB selected pseudo-randomly without replacement, thus preventing duplicates.

note:

- Since the elements of the result are selected from ιB, the result is ORIGIN dependent (see ORIGIN).
- This function uses and modifies the SEED parameter (see SEED).
- If A=0, or both A=0 and B=0, an empty vector results.
- Repeated calls with the same arguments produce different results (see examples).

examples:

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

GRADE UP

syntax: $R\leftarrow\blacktriangle B$

domain: B must be numeric.

range: Ordinal.

conformability: $(\rho B)\leftrightarrow 1$

result shape: $(\rho R)\leftrightarrow\rho B$

definition: The result R is a vector of the indices of B suitably arranged such that $B[R]$ is the ascending sorted arrangement of the elements of B in which the relative order of equal elements of B is undisturbed.

note:

- All element comparisons are exact; this function does not use FUZZ.
- Since the elements of the result are indices, the result is ORIGIN dependent (see ORIGIN).

examples:

```
      ▲4 7.3 ¯3.7 1 5.27 8.1E7
3 4 1 5 2 6
      □←R←▲B←2 7.5 2 918.3 7.5
1 3 2 5 4
      B[R]
2 2 7.5 7.5 918.3
```

GRADE DOWN

syntax: $R\leftarrow\forall B$

domain: B must be numeric: $0=1\uparrow 0\rho B$

range: Ordinal.

conformability: $(\rho B)\leftrightarrow 1$

result shape: $(\rho R)\leftrightarrow\rho B$

definition: The result R is a vector of the indices of B suitably arranged such that B(R) is the descending sorted arrangement of the elements of B in which the relative order of equal elements of B is undisturbed.

note:

- All element comparisons are exact; this function does not use FUZZ.
- Since the elements of the result are indices, the result is ORIGIN dependent (see ORIGIN).

examples:

```
      ⍒4  7.3  ¯3.7  1  5.27  8.1E7
6 2 5 1 4 3
      ⎕←R←⍒B←2  7.5  2  918.3  7.5
4 2 5 1 3
      B[R]
918.3 7.5 7.5 2 2
```

BASE VALUE

syntax: `R←A⊥B`

domain: A and B must both be numeric

conformability: `(⍴⍴A)←→1`

`(⍴⍴B)←→1`

`(⍴A)←→⍴B`

result shape: `(⍴⍴R)←→0`

definition: The result is a scalar whose value is that represented by B in a number system with radices specified by A.

The result is formed by taking the classical inner product

```
R←W+.×B
```

where W is a weighting vector of the positional values of each digit of the represented number B, based on the radix scheme `A`.

```
W[I]=×/I↓A
```

(Recall `(¯1↑W)←→(×/⍳0)←→1`, the multiplication identity element.)

Note that `1↑A` is not used in forming W, but an element is required for conformability.

(See examples on next page.)

examples:

```
      10  10  10⊥1  2  3
123
      2 2 2⊥1 0 1
5
      2⊥1 0 1
5                                   (scalar extension of A)
      2 2 2⊥1
7                                   (scalar extension of B)
      2 2⊥1 0 1
LENGTH ERROR
      v                             (argument lengths different)
      2 2⊥1 0 1
      0 3 12⊥3  2  3.25
135.25
                                    (inches in 3 yards, 2 feet, 3 1/4 inches)
```

REPRESENTATION

syntax: `R←A⊤B`

domain: A and B must both be numeric.

range: `R≥0`

`R<|A IF A≠0`

conformability: `(⍴⍴A)=1`

`(⍴⍴B)=0`

result shape: `(⍴R)←→⍴A`

definition: The result is a vector, the elements of which form the representation of B in a number system with radices specified by A.

If: `(⍴A)=1`

then: `(A⊤B)←→A|B`

Case 1 - A contains no zero elements.

If: `S←¯1↑A`

`A1←¯1↓A`

then: `(A⊤B)←→((A1⊤(B-R1)÷S),R1←S|B`

Case 2 - A contains zero elements.

If: `N←(⍴A)-(⌽A)⍳0  (IN ORIGIN 1)`

then: `(A⊤B)←→(N⍴0),(N↓A)⊤B`

Note that this results in all elements of A to the left of the rightmost zero being ignored. This is because the `0|` operation causes the remainder of B to be exhausted; that is, `(R⊥B)=B` . If this necessitates an element of R being negative in violation of the range restriction, a DOMAIN ERROR results.

(See examples on next page.)

examples:

```
      10  10  10⍴123
1 2 3                          (decimal representation of 1 2 3)
      2  2  2⍴3
0 1 1                          (binary representation of 3)
      2  2  2⍴¯3
1 0 1                          (two's complement representation of ¯3)
      10  10⍴123
2 3
      0  10⍴123
12 3
      ¯10 ¯10 ¯10 ¯10⍴¯123
1 9 3 7
      0  ¯10⍴¯123
DOMAIN ERROR
           ∨
      0 ¯10⍴¯123               (result may not be negative)
      0  12⍴113
9 5                            (quotient and remainder of 113÷12)
      0  1⍴12.34
12 0.34                        (integral or fractional part of 12.34)
      0  1⍴¯12.34
DOMAIN ERROR
         ∨
      0 1⍴¯12.34               (result may not be negative)
      0  3  12⍴135.25
3 2 3.25                       (yrds., feet, inches in 135.25 inches)
      0  0.3  2⍴3
3 0.1 1                        (results with fractional radices)
```

EVALUATE

syntax:	$R \leftarrow \mathbf{a} B$
domain:	**Character.**
conformability:	$(\rho \rho B) \leftrightarrow 1$ $(\rho B) \leq 256$ (APL*CYBER restriction)
definition:	The character vector B is assumed to represent an evaluable APL expression. EVALUATE interpretively evaluates this APL expression and, upon successful completion, returns the value of that expression (if any) as its result.
note:	Error detection and reporting are similar to that which would result if the expression represented by B were input for immediate execution.
application:	Using EVALUATE, APL programs can be constructed which modify APL source expressions prior to their evaluation.

(See examples on next page.)

examples:

```
      ⍎'A←5'
      A
5
      2×⍎'A←5'
10
      A
5
      SPA←'A←5'
      B←2×⍎SPA
      B
10
      NAME←'B'
      1+⍎NAME,'←3'
4
      B
3
      ⍎'2÷0'
DOMAIN ERROR                    (error detection as in
                                 immediate execution)
        ∨
⍎       2÷0
      ⍎'→5'
SYNTAX ERROR                    (not evaluable)
        ∨
⍎       →5
      ⍎')DIGITS 5'
PAREN BALANCE                   (not evaluable)
        ∨
⍎       )DIGITS 5
```

examples: numeric test

```
      NUM←'0=0\0ρ'
      B←⍳5
      ⍎NUM,'B'
1
      C←'ABC'
      ⍎NUM,'C'
0
      ⍎NUM,'⍳0'
1
      ⍎NUM,''''''
0
```

nested execution

```
      X←'(0ρA←1↓A),(0ρB←(B-R)÷N),'
      Y←'(R←(N←¯1↑A)⊥B),'
      AΔREPΔB←'⍎Y,(((¯1+ρ,A)×ρX,Y)ρX,Y),''⍳0'' '
      A←100 100
      B←357.91
      ⍎AΔREPΔB
3 57.91
```

In the above example, the character vector *AΔREPΔB* contains an evaluate function designator as its first character. Evaluating *AΔREPΔB* involves first evaluating *1↓AΔREPΔB* and then evaluating the result of that. *⍎1↓AΔREPΔB* results in a character vector which is a tailored APL expression dependent on the shape of A.

For the values in the example we have:

```
      ⍎1↓AΔREPΔB
(R←(N←¯1↑A)⊥B),(0ρA←1↓A),(0ρB←(B-R)÷N),R←(N←¯1↑A)⊥B),⍳0
```

This expression is then evaluated, yielding the final result.

FORMAT

syntax: $R\leftarrow\blacktriangledown B$

domain: B must be numeric

conformability: $(\rho\rho B)\geq 1$

result shape: $(\rho\rho R)\leftarrow\rightarrow 1+\rho\rho B$

$\rho R\leftarrow\rightarrow(\rho B),M$

where M is the number of characters required to represent a row of B such that the decimal points are aligned in each column.

definition: R is the character representation of B, with the shape as defined above.

note: Since the result is not actually displayed, it is not sensitive to the setting of WIDTH, which is a terminal display parameter.

application: The purpose of FORMAT is to convert numeric cata to character data which can then be suitable edited, combined with other character data and, finally, displayed in any desired form. FORMAT gives the user much more flexibility in formatting output than composite data object displays allow.

(See examples on next page.)

examples:

```
      X←⍕10+⍳5

      X

11
12
13
14
15

      ⍴X

5 2

      3 1 ↑X

1
1
1

      ⌽X

11
21
31
41
51

      X←1 2 3 4 5

      NAME←'X'

      INDEX←3

      ⍎NAME,'[',(⍕INDEX),']←7'

      X

1 2 7 4 5
```

I—BEAM

syntax: $R \leftarrow \text{I} B$

domain: B must be integer (see table below).

conformability: $(\rho\rho B)=0$

result shape: $(\rho\rho R)=0$ for B = 18 through 26.

$(\rho\rho R)=1$ for B = 27, 28 and 29.

definition: The I-beam function provides a mechanism for the user to inquire about certain items of system information not part of the APL language. The particular piece of information desired is indicated by the value of B.

The values of B accepted by APL*CYBER and the information returned are indicated in the following table.

B	Information
18	Current amount of workspace in use (words).
19	Total time APL has been awaiting input from this user (since sign-on).
20	Time of day (sec. since midnight).
21	Total CPU used since sign-on (sec.).
24	Total time this user has been signed on APL (sec. since last sign-on).
25	Today's date ($YYMMDD_{10}$).
26	Value of current function line number.
27	Vector of line numbers of stacked functions.
28	Type of terminal signed on this port.
29	User ID signed on this terminal.

APL EXPRESSIONS 10

INPUT REPRESENTATION FORMAT

APL expressions input from the terminal are formed according to the following rules:

USE OF SPACES

- Spaces must not be used in forming identifiers, or in INNER or OUTER PRODUCT.
- Elements of numeric literal vectors must be separated by at least one space.
- At least one space must be placed between adjacent identifiers and between identifiers and numeric literal expressions.
- Spaces are explicitly interpreted as such where they occur in character literal expressions.
- Any other occurrence of spaces is optional, and is ignored.

USE OF PARENTHESES

- Parentheses are required to delimit the extent of an expression for the left argument of a function where that expression is other than a literal expression, a data identifier, a niladic function call, a QUAD or a QUAD-PRIME.
- Parenthesizing of any other expression (including one already parenthesized) is superfluous but allowed, unless the expression is the left argument of a specification.

CONVERSION OF INPUT REPRESENTATION

Input expressions are converted to a standardized internal format upon input. Superfluous space characters are ignored in this conversion. Arrays are created for literal expressions. If any element value of a numeric literal expression exceeds the range of the machine (see Appendix C), a DOMAIN ERROR occurs at this point in the line when the line is executed. All identifiers and function designators are also converted to an internal format. It is this internal format that is used by the interpreter in evaluating expressions.

EVALUATION OF EXPRESSIONS

ORDER OF EVALUATION

Any expression takes the overall form of a literal, a data identifier, or a function call. In the first two cases, evaluation is a one-step process. If the expression is a function call, evaluation proceeds as follows:

The right argument (if there is one) is evaluated first.

The function itself is then examined to determine if it is dyadic. For primitive functions which utilize the same designator character for both a monadic and a dyadic function, the function is interpreted as dyadic if the item to its immediate left is the rightmost item of an expression, namely: a literal expression, a data identifier, a niladic function call, a right parenthesis, a right bracket, a QUAD or a QUAD-PRIME. If no such item exists, the function is interpreted monadically.

If the function is determined to be dyadic, the left argument of the function is evaluated. If it consists of more than one syntactic element the desired left argument must be enclosed in parentheses. The interpreter utilizes the occurrences of the parentheses to determine the extent of the expression for the left argument.

With the argument(s) evaluated, the function call is then made and any returned result is the evaluated result for the expression.

The arguments, if present, are expressions in their own right and are evaluated in the identical manner as stated above.

ERROR DETECTION SEQUENCE

Error	Typical Causes
• { *QUOTE* / *PAREN* / *BRACKET* } *BALANCE*	non-matching { quotes / parentheses / brackets }
• *SYNTAX ERROR*	improper number of arguments supplied.
• *VALUE ERROR*	variable not established (could be misspelled).
• *DOMAIN ERROR*	supplied argument not in the domain of definition, or result not in the range of definition of the function.
• *RANK ERROR*	argument rank conformability requirement not met.
• *LENGTH ERROR*	other conformability requirement not met.
• *INDEX ERROR*	index out of range; applies to indexing and index notated primitive function calls.

Examples:

The following set of statements indicates the order in which execution is performed and errors are detected.

```
      Y←1 5 4 2 7 9
      Y(0.5+0 1×X+↑Y]
BRACKET BALANCE
                    v
      Y(0.5+0 1×X+↑Y]
      Y[0.5+0 1×X+↑Y]
SYNTAX ERROR
                   v
      Y[0.5+0 1×X+↑Y]
      Y[0.5+0 1×X+1↑Y]
VALUE ERROR
                 v
      Y[0.5+0 1×X+1↑Y]
      X←2 3ρ1 2 3 4 3 2
      Y[0.5+0 1×X+1↑Y]
LENGTH ERROR
             v
      Y[0.5+0 1×X+1↑Y]
      Y[0.5+(3 2ρ0 1)×X+1↑Y]
LENGTH ERROR
             v
      Y[0.5+(3 2ρ0 1)×X+1↑Y]
      Y[0.5+(2 3ρ0 1)×X+1↑Y]
INDEX ERROR
           v
      Y[0.5+(2 3ρ0 1)×X+1↑Y]
```

(Continued on next page.)

(Continued from previous page.)

```
      Y[⌊0.5+(2 3⍴0 1)×X+1↑Y]
INDEX ERROR
       ∨
      Y[⌊0.5+(2 3⍴0 1)×X+1↑Y]
      )ORIGIN 0
WAS 1
      Y[⌊0.5+(2 3⍴0 1)×X+1↑Y]
1 2 1
9 1 2
```

The following example indicates how a specific action within an expression is handled:

```
      A←2
      (A←5)+A
7
      A←2
      A+A←5
10
```

DISPLAYING EXPRESSIONS

When an expression is displayed, such as in an error report or in a requested display of a user-defined function line, an inverse conversion transforms the internal format to a display format. The display formatting follows the rules of canonical form.

CANONICAL FORM

- All displayed expressions must be in a form that is acceptable as input.
- Literal numeric expressions have the same form as employed in numeric data formatting. (See DISPLAYING DATA).
- Comments are displayed as they were entered.
- Except as required in the above points, spaces are not inserted in displayed expressions.

APL SYSTEM/USER INTERACTION 11

IMMEDIATE EXECUTION

When no other activity is taking place, the system awaits input for immediate execution. This is indicated by a 'prompt' from the system in the form of an indentation 6 spaces from the left margin. In this state, the user may enter:

- an expression to be evaluated.
- a system command.
- a function edit request.

When all processing resulting from the line input has been completed, the system again awaits input for immediate execution.

ABORTING EXECUTION OR OUTPUT

Whenever an expression evaluation, function execution, or output is taking place, processing may be interrupted. (This is accomplished on a terminal by pressing the 'BREAK' or 'ATTN' key.)

Any ongoing output is aborted. Expression evaluation is terminated at the end of the currently executing line. If a function is executing, it is suspended immediately before execution of the next line.

If the currently executing line was entered in response to a QUAD input request (see below), the request is not satisfied, and the QUAD prompt is reissued.

Example:

```
      X←⎕←3 4⍴⍳12
1 2 3 4
5 6▲_____________________ (output aborted at this point)
      ⍴X
3 4                        (note specification to X was done, since
                            evaluation continues until the end of the
                            line is reached)
```

QUAD INPUT

syntax: *R*←⎕

If the symbol ⎕ (QUAD) appears anywhere except in the construct ⎕← , it signifies that an expression is to be evaluated at that point, the source for which is to be supplied from input.

At the point where a QUAD in the above stated context is reached in the execution of an APL source line, further execution is pendant on an evaluated result for QUAD.

A 'prompt' to the user terminal is sent in the form:

⎕:

at the left of a line. This is followed on the next line by indentation 6 spaces from the left margin. At this point the system awaits input to be submitted.

Input must be in the form of an APL expression. Upon entering the line the APL expression is evaluated as for immediate execution.

Simply entering an empty line causes the ⎕: to reappear.

If no errors are detected on evaluating the submitted APL expression, the result obtained is returned as the result for the QUAD function and evaluation of the original source line continues.

If evaluation of the expression input after the prompt is not possible due to some error in the submitted line, the appropriate error report is issued, followed by another prompt, with the system again awaiting input.

The user may now resubmit the expression, correcting the error, or alternatively use the line editing feature to correct the existing line (see LINE EDITOR).

The symbol ⎕ used in this manner can be likened to an implicit result-returning niladic user-defined one-line function in which the user supplies the line each time the function is called. As such it has two properties in common with regular user-defined functions.

- Recursive calls can be made with QUAD by submitting as part of the input expression another QUAD.
- Exit from all further evaluation of expressions at all levels is possible by inputting after the prompt line a niladic branch:

  →

  This provides an exit mechanism from an infinite loop requesting and evaluating input.

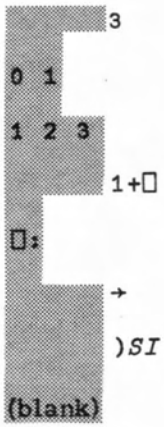
Instead of entering an APL expression, it is acceptable to enter a system command. All valid system commands will be carried out. If the system command replaces the existing active workspace with some other workspace, such as by)LOAD or)CLEAR, request for input is terminated.

Examples:

```
      2+⎕                    (immediate execution input)
⎕:
      3×5                    (response to QUAD)
17                           (result)
```

Another way in which QUAD appears like a user-defined function can be seen by issuing)SI or)SIV in response to a request for input.

```
      1,2+⎕←0.5×⎕-1          (immediate execution input)
⎕:                           (QUAD prompt issued)
      )SI
⎕                            (QUAD pendant)
⎕:                           (prompt reissued)
      +.                     (response to QUAD)
SYNTAX ERROR
        ∨
      +.
⎕:                           (prompt reissued)
      1,⎕
⎕:                           (prompt from second QUAD)
      )SI
⎕                            (two QUAD's pendant)
⎕
⎕:                           (prompt reissued)
```



(response to last QUAD)

(display from □←)

(result)

(immediate execution input)

(prompt issued)

(exit from last execution)

(nothing pendant or suspended)

(system again awaits input for immediate execution)

QUAD-PRIME INPUT

syntax: *R*←⍞

If the character ⍞ (QUAD-PRIME) appears anywhere except in the construct ⍞←, it signifies that character data is to be obtained from input.

At the point where a QUAD-PRIME in the context stated above is reached in the execution of an APL source line, further execution is pendant on a result obtained for QUAD-PRIME.

No prompt occurs with QUAD-PRIME other than a bell signal or keyboard unlock. The system simply awaits input at the left margin.

Input consists of a line of zero or more characters. Unlike normal input of explicit character literals, a quote character to mark the beginning and end of the literal is not used. Further, the quote character is represented by itself and not by two consecutive quote characters.

The explicit character literal, as input (subject to conversion of illegal characters to the canonical bad character), is returned as the result for QUAD-PRIME, and evaluation of the original source line continues until completed.

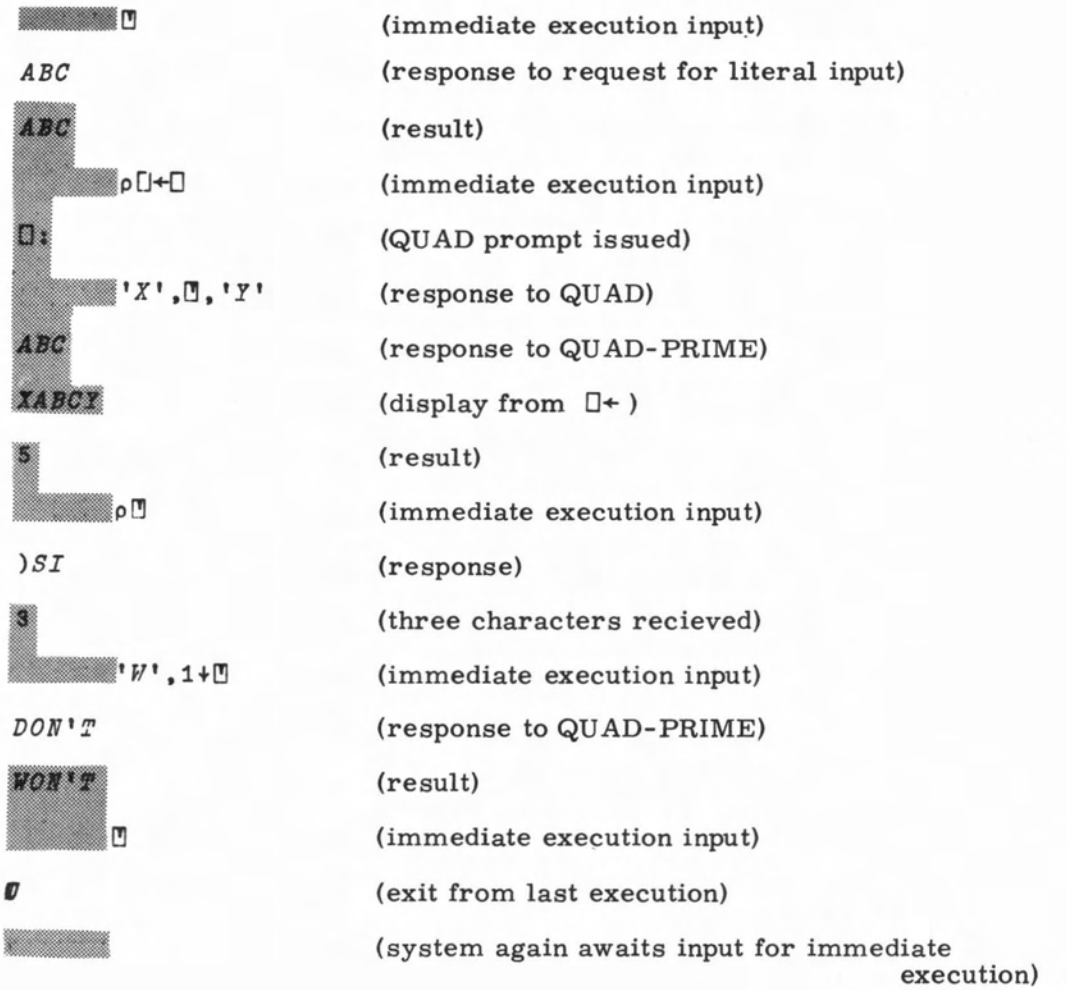
Input of a single character results in a character scalar. Input of no characters or more than one character results in a character vector.

Since all character inputs are taken literally and are not interpreted, this function cannot be used recursively. Likewise, system commands will not be interpreted as such.

A single exception to the above is a special character provided solely for the purpose of providing an escape mechanism identical to that provided by→for the QUAD function. This special character is the composite formed by overstriking the letters 'O', 'U', 'T'. (For terminals without overstrike capability, the mnemonic sequence is '$G.')

(See examples on next page.)

Examples:


(immediate execution input)
ABC
(response to request for literal input)
ABC
(result)
ρ□←□
(immediate execution input)
□:
(QUAD prompt issued)
'X',□,'Y'
(response to QUAD)
ABC
(response to QUAD-PRIME)
XABCY
(display from □←)
5
(result)
ρ
(immediate execution input)
)SI
(response)
3
(three characters recieved)
'W',1↓
(immediate execution input)
DON'T
(response to QUAD-PRIME)
WON'T
(result)
(immediate execution input)
(exit from last execution)
(system again awaits input for immediate execution)

VISUAL FIDELITY

The underlining concept in entering a line of input is visual fidelity; i.e., that the appearance of the line upon submission is what is conveyed, rather than the sequence used to form the line. The implications of this concept are as follows:

- The position of the terminal carriage, type ball or cursor is immaterial upon hitting the return key.
- The order in which characters are keyed is immaterial.
- On terminals with a destructive overstrike (CRTs) any character may be replaced by any other, including blank, prior to hitting return; only the final appearance will be conveyed.

ABORTING AN INPUT LINE PRIOR TO SUBMISSION

- Position to the right of the right-most input character.
- Press the 'BREAK' or 'ATTN' key.

The system returns to the same input mode as existed prior to entering the line.

The next entered line will be processed as a new line for the input mode in effect unless its left-most non-blank character is a '/' (see LINE EDITOR, below).

CORRECTING AN INPUT LINE PRIOR TO SUBMISSION

Method (A) (requires a terminal with a backspace key and a 'BREAK' or 'ATTN' key).

1. Position via any combination using the backspace key and/or space bar to the left-most character to be erased.
2. Press the 'BREAK' or 'ATTN' key.
   - A marker is displayed beneath the point reached.
   - Indentation on the following line occurs to the point beneath the marker.
   - The system awaits input to append to that portion of the original line to the left of the marker.
3. Key in appended text (if any).
4. Submit the corrected line for execution.

Method (B) (requires a terminal with a 'BREAK' or 'ATTN' key).
See LINE EDITOR (below, case 1).

Method (C) (A combination of methods A and B above.)

1. Employ method A steps 1 thru 3.
2. Employ method B to make other changes.

LINE EDITOR

The line editor can be used to edit lines in the following three cases.

(1) An input line prior to its submission, may be edited by positioning to the right of the right-most input character and keying 'BREAK' or 'ATTN'.

(2) A line submitted for immediate execution may be edited immediately after issue of an ensuing error report for that line.

(3) A function line which has just been displayed by the function editor, may be edited without leaving the editor. (See FUNCTION EDITOR.)

In each of these cases, if the next input line contains a '/' as the left-most non-space character, it will be interpreted as a line edit command for editing the line in question.

LINE EDIT COMMAND

syntax: / [<delim.> <char. seq.> <delim.> <char. seq.> <delim.>]

The line edit command consists of the '/' character followed by two delimited character sequences of text. Any non-blank character may be used as a delimiter, but must be consistently used to delimit both character sequences, and must not appear in either character sequence. Each character sequence may independently be of any length including empty.

action: The line editor updates the line in question by replacing the left-most occurrence of the first character sequence by the second one. An empty first sequence implies the whole line.

The specific edit process for various forms of the line edit command is as follows:

('.' is used here as the delimiter character.)

COMMAND	EDIT PROCESS
/.<*SEQ*1>.<*SEQ*2>.	replace the left-most occurrence of sequence 1 with sequence 2.
/.<*SEQ*1>..	delete the left-most occurrence of sequence 1.
/..<*SEQ*2>.	replace the line with sequence 2.
/...	display the entire line and await additional input at the end.

Upon completion of the editing process, the edited line is displayed and the system awaits input to append to the end of the edited line. Keying 'BREAK' (or 'ATTN') at this point allows the line editor to be reentered for making further changes. After submission of the appended text or a blank line, immediate execution of the edited line is initiated in cases 1 and 2. In case 3 the edited line is inserted into the function definition and an appropriate prompt is issued. (See FUNCTION EDITOR.)

error reports: MISSING DELIMITER

This may be caused by omitting one or more delimiters.

NOT IN LINE

Specifying a sequence, for which no match can be found in the line.

Additional attempts at entering a line edit command to edit the original line are allowed following this error report.

examples:

```
      B1←((¯1↓ρB),AρB▲                    ('BREAK' keyed before submitting)
      /.A.A).                             (replace "A" with "A)")
      B1←((¯1↓ρB),A)ρB←C,D                ("←C,D" appended to edited line
                                            and then submitted)
      B2←×/[3]A,B

INDEX ERROR
            ∨
      B2←×/[3]A,B                         (this line can now be edited)
      /,[3].,                             (delete [3])
      B2←×/A,B                            (edited line is displayed, then submitted)
                                          (the line executes successfully)
      ∇SQUISH[2□]
[2]   R←¯1↓((¯1↓L)∨1↓L)/1↓X
[2]   /..
MISSING DELIMITER.
      /.X.X,Y.                            (display text of line, changing "X" to "X,Y")
[2]   R←¯1↓((¯1↓L)∨1↓L)/1↓X,Y             (submit without appending)
[3]                                       (close definition - line 2 is replaced as edited)
```

USER-DEFINED FUNCTIONS 12

FUNCTION DEFINITION

To provide an open-endedness to APL, a user may supplement the primitive functions with those he defines himself.

The syntax of a user-defined function definition consists of a function header followed by a function body. The function header declares the name of the function and its syntactic form. The function body consists of zero or more lines of APL, each of which may be preceded by a label (see LABELS).

FUNCTION HEADER

In addition to the monadic and dyadic syntax of primitive functions, user-defined functions may be defined having no arguments (niladic syntax).

User-defined functions may be result-returning, as are primitive functions, or non result-returning.

The above criteria and the function name are established by the function header. The form of a function header is as follows:

$$\left[\ \langle\text{result}\rangle\leftarrow\ \right]\ \left\{\begin{array}{l}\langle\text{l. arg.}\rangle\ \langle\text{function name}\rangle\ \langle\text{r. arg.}\rangle\\ \langle\text{function name}\rangle\ \langle\text{r. arg.}\rangle\\ \langle\text{function name}\rangle\end{array}\right\}\ \left[\ ;\ \langle\text{explicit local list}\rangle\ \right]$$

where: <result> is the local result name
<l. arg.> is the local left argument name
<r. arg.> is the local right argument name
<explicit local list> is a list of identifiers separated by semicolons.

Identifiers in the function header other than the function name (i.e., arguments, result, and explicit local list) declare variables local to the function environment. (See ENVIRONMENT OF AN ACTIVE FUNCTION.)

FUNCTION BODY LINE

The form of a function body line is as follows:

$$\left[\langle\text{label}\rangle:\right]\langle\text{executable portion}\rangle\left[\text{A}\langle\text{comment}\rangle\right]$$

FUNCTION CALL

A dyadic function name FLIP having numeric arguments could be invoked by:

```
2 3 7 FLIP 8 1
```

If the function header for FLIP is

```
R←A FLIP B ; X ; Y
```

then at the time FLIP is invoked, A has the value 2 3 7 and B has the value 8 1 .

The process of assigning values to A and B at the time of function call is similar to specification.

FUNCTION EXECUTION

Upon function call values are supplied to the function arguments (if any), and the body of the function is executed.

Each line is interpretively executed in the normal right-to-left manner starting with the first line.

Lines are executed in sequence in order of occurrence unless otherwise directed by a branch (see BRANCH). When the last line of the function is executed, if no branch is taken, the function exits.

Upon completion of function execution, the value returned is the value of the local result at that time. If no specification has been made to the local result, no result is returned.

BRANCH

Syntax: →B

Domain: non-negative integer

Conformability: $(\rho\rho B)\leq 1$

A branch must be the left-most operation on the line in which it appears. The domain of the argument B is integer. No result is returned from the operation. Those cases exist:

1. If B is empty, the branch is ignored.

   If B is not empty, all but the first element are ignored. Let I←1↑B .

2. If $I\epsilon 0\ 1[1]+\iota N$, where N is the number of lines in the body of the function, the next line to be executed will be line I .

3. Otherwise, execution of the function is terminated and the function exits.

4. B must be within FUZZ of a positive integer. Otherwise, a DOMAIN ERROR will result.

Note that numbering of function lines is not dependent on the index origin. Thus 1 (if it exists) is always the first line of the function, and →0 always causes an exit.

Niladic Branch

A second form of the branch directive exists which consists solely of the branch directive on a line by itself:

→

Execution of a niladic branch causes an exit, not only from the current function being executed but from the entire set of functions in the calling sequence initiated by the outermost function call, including the immediate line in which the outermost call was made.

The exit mechanism utilised when niladic branch is invoked bypasses all result-returning procedures for all currently invoked functions in the calling chain.

The purpose of the niladic branch is twofold:

1. To provide a termination path which stops all function execution.

2. To reinstate the workspace environment to as near as can be obtained to what it was prior to calling the initial function in the calling sequence.

LABELS

In forming expressions which evaluate to the number of some desired function line, it may prove difficult to predict what that number will be. Furthermore, the number will be subject to change if, subsequently, additional lines are inserted in the function or some lines are deleted.

The above difficulty is eliminated by the ability to reference function line numbers symbolically. This is accomplished by the use of labels.

An identifier followed by a colon may be placed to the left of the executable portion of any line to be referenced. Only one label may be placed on a line.

This identifier is the name of the label for the line. This label is local to the function (see ENVIRONMENT OF AN ACTIVE FUNCTION). When the function is called, it is given the value of the number of that line, in much the same way as the arguments are assigned values. The value of a label is always an integer scalar.

Labels have a property which distinguishes them from all other variables. During their existence they cannot be respecified (i. e., their value cannot be changed). Labels are thus the only named constants in APL. In all other respects, they are normal variables.

NOTE

As will be seen in the following section, label values are available to functions called by the function containing them. As labels are indistinguishable from any other variable, branching to such a label in a function called by that function will not cause a branch back to the labelled line in the calling function, but rather a branch to the line in the called function having the same line number. If no such line exists, an exit from the called function will occur.

ENVIRONMENT OF AN ACTIVE FUNCTION

When a function is called, values are assigned to its arguments and labels. All of its other local variables (the result and explicit locals) become undefined (i.e., have no value).

This constitutes an initial local environment at function call.

A function possesses a local environment from the time it is invoked until exit from the function occurs. During this time the function is said to be active.

The fact that the local environment disappears upon function exit is a useful mechanism for minimizing workspace requirements and for keeping the workspace from being cluttered with data objects which are no longer required.

Since explicit locals and the result have no value until first specified, while the function is active, prior reference to such variables inside the function results in a VALUE ERROR.

Also, since the local environment disappears on exit from the function, values specified to locals on one function call are not available to the function on subsequent calls.

In addition to the local environment, the total function environment initially consists of the entire workspace environment prior to function invocation, except for those objects whose names are identical to identifiers appearing in the formal parameters or local list of the function header, or label identifiers.

These latter objects are said to be masked while the function is active. Note that all masking occurs at the time of function invocation, and not when subsequent specification for some local is first made.

Objects in the function environment which are not part of the local environment are termed the global environment.

The global environment includes, in addition to those workspace objects not masked on function invocation, the workspace environmental parameters Origin, Digits, Fuzz and Seed.

Functions can thus make reference to objects and respecify variables which are part of their global environment. New global variables can also be created by specifying to a name not appearing in the local list. This ability provides the function with a communication facility separate from that provided by the argument and result parameters, and is the only method available to niladic non result-returning functions.

NESTED FUNCTION CALLS

At any point during execution, it is possible for a function to invoke any other function defined in its environment.

When a function calls a function, the calling function still remains active (since an exit from it has not yet occurred); however, it is no longer executing, but rather waiting for the called function to complete its execution. During this time the calling function is said to be pendant. When the called function has completed its execution, it exits back to the calling function, returning a result if any.

Execution of the calling function then recommences at the point where it left off, and the calling function is now no longer pendant.

Calls to non result-returning functions from a function must be placed alone on a separate line within the body of the called function, otherwise a VALUE ERROR will result when the line attempts to reference the non-existent result of the function.

Result-returning functions, on the other hand, can appear as arguments in more complex expressions to be evaluated, including additional function calls.

The environment of a function while pendant is kept intact, while the called function creates its own local environment. The total environment of the calling function becomes the potential global environment of the called function from which certain objects may be excluded due to masking. Objects which were masked by the calling function remain masked to the called function.

The origin of objects in the called function's global environment is indistinguishable to it. It may indiscriminately reference, change and create global objects which are either local or global in the calling function.

The state of the workspace environment upon the completion of all function execution (known as the absolute global environment) will be affected, however, if the inner function respecifies one of these objects or creates new ones. If, on the other hand, only objects which were part of the local environment of one of the functions in the calling sequence were effected, no change to the absolute global environment would occur.

Note that a called function's local environment is invisible to the calling function, whereas both its own local environment and global environment can be affected while pendant.

The process of having a function call a function can be continued by having that function call another function, etc. This gives rise to a calling chain of function calls. The calls are said to be nested from the outermost call to the innermost one. All called functions except

the innermost are pendant. Local environments exist in the workspace for all the function calls in the sequence. Masking can occur at each call level.

The number of calls in the call sequence is termed the depth of nest of the innermost function call. Nesting can occur to any level for which sufficient available space in the workspace exists to create a local environment for the function called at that level. An attempt to nest deeper than this results in the error message WS FULL and the function attempting to make a call is suspended on the line in which the call occurs.

A NOTE ON RECURSIVE CALLS

Recall that a function may issue a function call to any function in its global environment. As long as the called function is not masked on calling the function, it will exist in the function's global environment and can just as validly be called as any other function in its environment.

Any call sequence in which a function calls itself or any function in the current calling sequence that is pendant, is said to be a recursive call. Recursive calls give rise to the situation where one call of a function is currently executing while one or more other calls of the same function are pendant in the same calling sequence.

The fact that multiple pendant calls and a currently executing call, all to the same function can co-exist, in no way causes problems. This is due to the fact that each call of the function creates a separate local environment to be used by that function call as long as that call is active. In this way each function call keeps track of its own environment and is oblivious to all other local environments.

Each recursive call nests deeper in the calling sequence. Since successive recursive calls usually emanate from the same line in the calling function, that line when executed on successive calls causes further recursion to occur. If care is not taken, the nesting depth will become excessive, filling up the workspace with local environments of pendant calls to the point where a WS FULL message occurs.

A function employing a recursive call must therefore provide an alternative path to be taken when some limiting condition occurs which bypasses the line invoking a further recursive call. The limiting condition must be met by some innermost recursive call within an allowable nesting depth. This call must then be allowed to complete without invoking further recursive calls and exit to its caller. In like manner, each called function in turn uses any returned result in completing its execution and exits in turn to its caller, progressively reducing the nesting level until the outermost call is completed, whereupon all function execution terminates.

PURPOSE

The APL*CYBER system contains a utility called the function editor which accepts suitable input in the form of a function definition, and upon completion stores in the active workspace a defined function suitable for subsequent execution.

The utility can also be used to display all or part of a function definition or to modify an existing defined function as desired.

INVOKING THE EDITOR

Whenever the system is awaiting input for immediate execution, the function editor can be entered by placing the APL character ∇ ('del') as the left-most non-space character of an input line. This must be followed on the same input line with the name of an existing defined function in the workspace, which the user wishes to modify or display, or the function header of a new function which the user wishes to define.

If the syntax of the function header is invalid, or contains the name of a currently existing global object, the error report DEFN ERROR results, and the function editor exits.

Notation: In the examples in this section, shaded text indicates APL system response; unshaded text is entered by the user.

```
      ∇R←A NEW B
[1]
```

NEW is a new function being created. The editor prompts for an entry for line 1.

```
      )FNS
OLD
      ∇OLD
[4]
```

OLD has 3 body lines. the editor prompts for an entry for line 4.

```
      FUN←5
      ∇R←FUN B
DEFN ERROR
```

FUN is a currently existing global object, and thus cannot be used as a function name.

NOTE: The function editor can be entered while a defined function is suspended. The local environments may cause masking of the function being modified or created. Masking does not effect the ability of the function editor to access or create defined functions. Masking will, however, prevent calling these functions until the local environments of the active functions are removed. (See)SI , NILADIC BRANCH)

SUPPLYING FUNCTION DEFINITION BODY LINES

Upon sucessfully entering the function editor with an input line in the form as stated above, subsequent lines of input are implicitly considered to be consecutive lines of the body of the function definition, unless their form indicates otherwise. The editor 'prompts' the user for each such line by displaying a line number in brackets at the left of the line to be entered. For a function being newly created, the first prompt is [1]. For a previously defined function, the first prompt is [<L+1>], where L is the number of body lines in the previous definition of the function.

```
[4] R←(¯1↑⍴C)⌈1↑⍴A          |  [1] C←(⍴A),⍴B
[5]                         |  [2]
```

The prompt number always indicates the relative position an input body line will have in the completed function, unless that input is suitably annotated to override this placement.

Overriding is accomplished by entering a line number in brackets, optionally followed by the body line entry all on the same input line. If only the line number in brackets is entered, the editor responds with a prompt as entered.

```
[5] [7] A←((3≤⍴⍴A),⍴A)⍴A
[8] [6]
[6] →0⌈⍳(⍴⍴A)≠⍴⍴B
[7]
```

When overriding the prompt line number, a non-integer decimal numeral with a fraction part of up to 4 decimal digits can be supplied. (Using more than this results in the error report EDIT ERROR.) By this means, a line position in the function body between two previously entered lines can be indicated.

```
[3] [2.3] →0,⍴⎕←'DOMAIN ERROR'
[2.4] [2.1]
[2.1] →L1×⍳(0↑,A)=0↑,B
[2.2]
```

After entering a body line of the function definition, the editor again returns a prompt. The line number of this prompt is obtained by incrementing the number of the previously entered line by .1*D where D is the number of fraction digits last used in overriding a prompted line number. (D is set to zero initially.)

REPLACEMENT OF AN EXISTING LINE

In the same manner that new lines are placed in a function definition, a previously existing line can be replaced with a new entry. The prompted line number is overridden by the line number of the existing line, and the new body entry is supplied which then replaces the old entry.

```
[2.2] [3] L1:A←((3<ρρA),ρA)ρA
[4]
```

NOTE: The function header can be changed in this manner by designating the line to be changed as zero. If the entered header results in a DEFN ERROR, a prompt for line zero is issued and the previous function header is maintained.

```
[4] [0] R←A OLD B;C
[1]
```

DELETING AN EXISTING LINE

Deletion of an existing line is done using the same procedure as for replacement of a line. The editor does not incorporate a body entry line that is completely blank into the function definition. Thus, replacing an entry with a blank entry effectively deletes the line. A blank entry cannot be conveyed on the same input line that specifies an override line number. The override directive must be submitted on one input line, and the blank line entry submitted on the line on which the revised prompt appears. For keyboard terminals, a blank line entry is produced by merely depressing the RETURN key.

Example:

```
[1] [2]
[2]
[3]
```

NOTE: A blank entry will cause a revised prompt of N+. 1 *D to be issued where N is the previous prompt, even if there was no line N.

RESTRICTION ON EDITING ACTIVE FUNCTIONS

- The only active function that can be edited is the one indicated at the top of the state indicator display (see STATE INDICATOR), and then only if it does not additionally appear elsewhere in the display. This corresponds to the case where the function is the currently suspended one, and where no calls prior to the suspended one are incomplete.
- Any attempt to change the local environment of a function in the above circumstances, such as by modifiyng the function header, or deleting, changing or adding labels to the function definition, results in a DEFN ERROR.

NOTE: In the event of a DEFN ERROR described above, the local environment must be changed back to its former state or the name of the function must be changed.

However, if as the result of editing, the line number of a labeled line is changed, the value of the label will be updated to the correct value upon exit from the function editor.

CREATING SEPARATE VERSIONS OF A FUNCTION

If while editing a non-active function, the name of the function is changed by editing the function header, then upon exit from the editor, all such changes will be reflected in a user defined function having the new name supplied. The old version of the function will still exist under the old name. Both function definitions will be available for subsequent editing.

DISPLAY DIRECTIVES

A display directive may be entered after any prompt in lieu of a body entry or override directive, or as the last part of the function editor invoking line.

(A) Displaying the complete function definition.

- Enter the directive [□] after any prompt, or as the last part of the invoking line.
- The complete function definition is displayed followed by a prompt for line 1+⌊*L* where *L* is the last existing line.

Example:

```
[4] [□]
        ∇R←DATE YMD;M
[1] YMD←⌊0.5+¯3↑0 0 0,,YMD
[2] →0×⍳∨/YMD≠0 1 1⌈99 12 31⌊YMD
[3] R←1 1 0 0 0 0 0 0 0 1 1\2↑,FM 3 1⍴55,YMD[2 0+⍳1]
[4] M←12 3⍴'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
[5] R[3+⍳6]←M[YMD[1+⍳1]+¯1+⍳1;],' 19'
        ∇
[6]
```

(B) Displaying a function definition from line N to the end.

- Enter the directive [□*N*] after any prompt, or as the last part of the invoking line.
- The specified lines are displayed followed by a prompt for line 1+⌊*L*.

Example:

```
[6]   [□4]
[4] M←12 3⍴'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'
[5] R[3+⍳6]←M[YMD[1+⍳1]+¯1+⍳1;],' 19'
        ∇
[6]   [□7]
        ∇
[6]
```

NOTE. If N > L, no lines are displayed.

(C) Displaying a single line.

- Enter the directive [*N*□] after any prompt, or as the last part of the invoking line.
- The specified line is displayed if it exists.
- This is then followed by a prompt for the line number indicated.

Example:

```
[4]    [2□]
[2] →0×⍳V/YMD≠0 1 1[99 12 31[YMD
[2]
```

EDITING AN EXISTING LINE

An existing line may be edited by first displaying it and then, following the subsequent prompt for that line, employing the line edit procedure (see LINE EDITOR) to edit the displayed line. This technique is more expedient than replacing the entire line if the change is minor and the line is quite long.

The line to be edited is exactly as displayed, including the line number in brackets. An appropriate search string must be chosen with this fact in mind.

If the line number is changed by editing, a new line having that number is created, and any other editing changes made in the current line editing process apply only to it. The originally displayed line is, in this case, left unchanged. The new line will be inserted in the appropriate position in the function definition.

Upon completion of the line edit procedure and incorporation of the line as edited into the function definition, a prompt for line N+.1*D follows, where N is the line number of the edited line.

REPOSITIONING AN EXISTING LINE

An existing line can be repositioned in the function definition by editing the line number of the line after displaying it, as discussed above. This results in the line appearing at both the old and new positions. The delete procedure can then be used to remove the line from its old position.

TERMINATING THE FUNCTION EDITOR

When the user is satisfied with the function definition he has supplied to the editor, or with any changes or displays he may have requested, he may indicate termination from the editor by placing a ∇ ('del') as the last non-blank character on any input line. Upon successful completion of any request of the input line, exit from the editor occurs and the system awaits input for immediate execution.

Example:

```
[3]      [2⎕]∇
[2] →0×ιV/YMD≠0 1 1⌈99 12 31⌊YMD
(system awaiting input for immediate execution)
      ∇DATE
[6]      ∇
(system awaiting input for immediate execution)
```

If, however, the request cannot be accomplished, the appropriate error is issued and exit from the editor does not occur. Instead an appropriate prompt is issued.

As part of the function exit procedure, the lines of the function definition body are assigned contiguous integer values starting with one, independent of the ORIGIN setting.

DOCUMENTING USER-DEFINED FUNCTIONS

Since an APL user can display any part or all of a function at any time, it would be useful for function lines to be capable of containing non-executable character data which could serve as documentation for the function, supplied while it is being created or subsequently edited.

Useful information could include purpose of the function, acceptable shapes, domains and conformability of the arguments (if any), nature of the result (if any), as well as explanatory comments to clarify any body line in the function definition.

To achieve this capability, special symbol ⍝ (verbalized 'lamp') is available. The lamp symbol acts as a delimiter. Just as the colon delimits a label from the executable portion of a line to its right, the lamp symbol delimits any executable portion of a line from documentation to its right.

Example:

```
[3] LINE3:B←(ρA)ρB   ⍝ GIVE B SHAPE OF A
[7] ⍝ THIS WHOLE LINE IS A COMMENT
```

NOTE: Comments must not appear in a function header line.

USING SYSTEM COMMANDS WHILE EDITING

While the editor is invoked, an input line in the form of a system command will be interpreted as such. The most useful system command to issue while editing is the)SAVE command.)SAVE will save a current edition of the active workspace, including the latest form of the function definition currently being created or modified. Should the APL session be aborted due to transmission line difficulties or system failure, recovery from the SAVE'd workspace will minimize the number of lines to be reentered. The saved version of the function definition is stored as if an exit from the editor had taken place. That is, definition lines are renumbered, and if the SAVE'd workspace is loaded, the editor must again be invoked before it can be used.

FUNCTION EDITOR ONE-LINERS

For an existing function, the line invoking the editor can specify a one-line addition or replacement or a display directive, followed by a closing ∇ . Thus a single input line can invoke the editor, direct one task to be done, and cause exit from the editor, with the system then awaiting input for immediate execution.

Example:

```
      ∇SQUISH[□]∇
      ∇R←SQUISH X;L
[1]   L←X≠1↑X←X,1↑X
[2]   R←¯1↓((¯1↓L)∨1↓L)/1↓X
      ∇
(system awaits input for immediate execution)

      ∇OLD[2.5] R←(¯1↑⍴A)⌈1↑⍴B∇
(system awaits input for immediate execution)
```

SUMMARY

A complete summary of the possible input combinations for invoking and using the function editor are listed on opposite page.

To invoke the editor (new function):

∇ <function header> ⊂∇⊃

To invoke the editor (existing function):

∇ <function name> [line entry> | <display directive>]

To enter a line:

⊂[< line number>]⊃ <body line> ⊂∇⊃

note: function header is line 0.

To display entire definition:

[□] ⊂∇⊃

To display all lines from N to last:

[□*N*] ⊂∇⊃

To display line N:

[*N*□] ⊂∇⊃

To delete line N:

[*N*]

<blank line>

To change current line number:

[<line number>]

To edit a line:

[*N*□]

<line edit procedure> (see line editor)

SYSTEM COMMANDS 14

INTRODUCTION

In addition to the APL language, the APL system provides for an additional method of communication in the form of system commands. System commands complement the facilities provided in the APL language and allow the user to monitor, vary and protect his processing environment.

SYNTAX

) <command name> [<parameter list>]

The above is the most general syntax of a system command. The valid syntactic form for a specific command will be stated under the description of that command. Items in the parameter list are delimited from each other and from the command name by one or more spaces. Any error in the syntax of the command results in the error report INCORRECT COMMAND.

DOMAIN

Certain system commands can have numeric parameters. The domain of these parameters is stated for each such command. Any value not in the required domain results in the error report INCORRECT COMMAND.

INPUT REQUIREMENTS

System commands will be interpreted as such in any of the following input situations:

- the system is awaiting input for immediate execution.
- the system is awaiting quad input (quad prompt at left).
- the system is awaiting a function line edit request.

In each of these cases, an input line in which the left-most non-space character is a right parenthesis will be interpreted as a system command.

Only one system command may be entered on any one input line.

Nothing else in addition to a system command may be entered in an input line.

Where system command names are longer than four letters, the first four or more letters of the name may be used in lieu of the complete name.

CATEGORIES OF SYSTEM COMMANDS

- Listing the global workspace objects.
- Forming, modifying and listing groups.
- Erasing global objects.
- Debugging aids.
- Determining and altering workspace environment parameters.
- Workspace library facilities.
- Termination of APL session.
- Examining and altering display device parameters.

ACTIVE WORKSPACE

Each currently active user is provided with an environment in which to process his data. This environment is called the active workspace.

The active workspace is a directly accessible storage allocation sufficient in size to contain the workspace objects currently defined, the function environments of currently active functions, the state indicator, stop lists, and the four environmental parameters: ORIGIN, DIGITS, FUZZ and SEED.

For APL*CYBER, the maximum size of a workspace is installation dependent, but is in the order of 10,000 words less than the maximum field length allowed for a user. Any attempt to exceed this capacity results in the error report WS FULL.

The active workspace has provision for an identification (ID) in the same format as in library workspaces. (See WORKSPACE IDENTIFICATION.)

CLEAR COMMAND

syntax:)CLEAR

action:
- provides an active workspace with the following:
  1. workspace ID empty
  2. no objects
  3. empty state indicator
  4. ORIGIN 1
  5. FUZZ 2*⁻43 (approx. IE⁻13)
  6. DIGITS 10
  7. SEED 192527075924404
  8. input mode: awaiting input for immediate execution
- successful completion of the command is indicated by the report CLEARED WS.

example:
```
)CLEAR
CLEARED WS
```

GLOBAL OBJECT INVENTORY

A system command listing global object names exists for each of the three kinds of workspace objects.

VARS COMMAND

syntax:)VARS [<letter>]

action: lists the names of global variables currently defined in the active workspace in alphabetic order:

no letter list all names.

with letter - list all names starting with this or any higher letter in the aphabetic sequence.

note: Alphabetic sequence is as follows:

A - Z
0 - 9 (cannot use digit as letter)
Δ
a - z

also note: Names of global variables currently masked will be listed.

examples:

```
a←ac←A←A2←a7←aB←5
          )VARS
A    A2    a    aB    a7    ac
          )VARS a
a    aB    a7    ac
```

FNS COMMAND

syntax:)FNS [<letter>]

action: lists the names of user-defined functions currently existing in the active workspace in alphabetic order (see)VARS).
The use of a letter causes the same action as when used with)VARS.

GRPS COMMAND

syntax:)GRPS [<letter>]

action: lists the names of group definitions in the active workspace in alphabetic order (see)VARS).

The use of a letter causes the same action as when used with)VARS.

GROUPS

A group is a named set of potentially existing global workspace objects. It is useful to be able to reference a package set of defined functions and their global variables as a group when using)COPY ,)PCOPY and)ERASE (q. v.). A group is defined by a group definition which, when supplied, is itself a workspace object.

A group definition is a named set of identifiers. The name of the set is the name of the group. The identifiers are names of potentially existing global workspace objects. If and when a global workspace object exists having a name identical to an identifier in the group definition, it is a member of the defined group. A group definition is supplied using the)GROUP command.

GROUP COMMAND

syntax:)GROUP <group name> <identifier list>

action: creates a group definition.

error report: NOT GROUPED - NAME IN USE.

The group name is the same as the name of an existing function or global variable.

REFERENCING GROUPS

When a reference is made to a group via the)COPY ,)PCOPY or)ERASE commands, reference is made initially to the group definition in the indicated workspace, and additionally to all existing global objects in that workspace referenced by identifiers in the identifier list. Such objects are said to referents of the corresponding identifiers.

Groups may themselves be referents of a group. In this case, reference is implied to all of that group's referents in the same manner as applied to the first group. Indirect referencing of referents to as many levels as there are groups within groups is thus accomplished.

example:

```
      X←Y←Z←T←'DATA'
      )GROUP G1 X Y G2
      )GROUP G2 Z T
      )SAVE WS
WS  SAVED - 73/05/01. 08.13.01.
      )CLEAR
CLEARED WS
      )COPY WS G1
WS  SAVED - 73/05/01. 08.13.01.
      )VARS
T       X       Y       Z
      )GRPS
G1      G2
```

An object can be a referent to more than one group. An object can be multiply-referenced directly and indirectly by the same group definition. Even circular definitions are possible. The mechanics of)COPY ,)PCOPY and)ERASE are such that the end result is identical to a single reference of such an object.

example:

```
)GROUP GP1 A A GP2
)GROUP GP2 B GP1 A
A←5
B←7
)SAVE WS
WS  SAVED - 73/05/03. 08.06.59.
)CLEAR
CLEARED WS
```

```
)COPY WS GP1
WS  SAVED - 73/05/03. 08.06.59.
)VARS
A       B
)GRPS
GP1     GP2
```

ALTERING A GROUP DEFINITION

(a) Adding members to the group.

)GROUP <group name> <addendum identifier(s)>

If the group name is the same as an existing group, it is an immediate reference to the old definition of the group. As such it implies the old identifier list. Identifiers in the current)GROUP command imply additions to the list.

example:

```
)GROUP X A B C
)GROUP X Z
)GRP X
A       B       C       Z
```

(b) Dispersing a group.

)GROUP <group name>

If a group command consists solely of a group name, it implies an empty identifier list, and thus a group with no defined members. This causes any previous group definition by that name to be destroyed, and no new one to be formed.

```
)GROUP X
)GRPS
(blank)
```

(c) Any general change in a group definition.

Disperse the existing group, then create a new group definition.

DISPLAYING A GROUP DEFINITION

A group definition can be displayed via the)GRP command.

GRP Command

syntax:)GRP <group name> [<letter>]

action: the identifier list of the group definition is displayed in alphabetic order (see)VARS). The use of a letter causes the same listing action as when used with)VARS.

error report: OBJECT NOT FOUND

<identifier list>

indicates a group definition could not be found in the active workspace with a name identical to the identifier listed.

```
      )GROUP X A B C
      )GRP X
A B C
      )GRP X B
B C
```

GENERAL NOTES ON REFERENCING GROUPS

1.)COPY and)PCOPY references to groups refer to the group definition and group members existing in the workspace being copied.
2. If)PCOPY is used to copy a group and a global object in the active workspace has the same name as the referenced group in the workspace being copied, no copying using that group name can occur.
3.)ERASE reference to a group refers to the group definition and existing group members in the active workspace.
4. Creation, modification, display and dispersing of groups can occur only in the active workspace and only reference the group definition, not its members.

ERASING GLOBAL OBJECTS

ERASE COMMAND

syntax:)ERASE <object name list>

action: Global objects having names corresponding to those in the object name list are erased from the active workspace.

If a name in the object name list is a group name for which there is a group definition in the active workspace, then in addition to erasing the group definition, all referents in the group definition are erased. If one of the referents is another group definition, erasure of that group definition and all of its referents is also performed. Indirect referents to all levels of group structuring are thus erased.

error report: If a referenced object cannot be found no message is reported, since this is the desired result upon completing the command.

NOT ERASED:

<identifier list>

Active user-defined functions cannot be erased.

example:

```
)VARS
V1      V2      V3
)FNS
F1      F2      F3
)GROUP G1 V1 F1
)GROUP G2 V2 F2
)GROUP G3 V3 F3
)GRPS
G1      G2      G3
```

```
)ERASE V1 F2 G3
)VARS
V2
)FNS
F1
)GRPS
G1      G2
)ERASE X
)VARS
V2
```

DEBUGGING AIDS

SI COMMAND

syntax:)SI

action: The)SI command produces a display of the State Indicator, a list of all the function calls that are currently active, displayed in reverse order to the sequence of the calls; i.e., the most deeply nested call in the current sequence is at the top of the list.

The line on which the function is pendant or suspended is placed in brackets after the function name. Function calls that are suspended are flagged with an asterisk (*).

Although not generally advisable, it is possible to initiate an additional calling sequence after a current sequence is suspended. If this is done, the state indicator will reflect the complete status of all such stacked suspended calling sequences, the most current listed first.

note: Each issuing of a niladic branch will remove the local environments of the most current calling sequence, and remove the corresponding entries in the state indicator up to the next suspended function. Thus in order to completely clear the state indicator, it is necessary to issue as many niladic branches as there are asterisks (suspensions) in the state indicator.

(See examples on next page.)

examples:

```
      ∇R←A FUN1 C;Z
[1]   R←A+FUN2 C
[2]   ∇
      ∇R←FUN2 C
[1]   +. ⍎ THIS LINE WILL SUSPEND ON SYNTAX ERROR
[2]   ∇
      2 FUN1 3
SYNTAX ERROR
         ∨
FUN2[1] +.⍎ THIS LINE WILL SUSPEND ON SYNTAX ERROR
      )SI
FUN2[1] *              Indicates FUN2 is suspended on line 1.
FUN1[1]                Indicates FUN1 is pendant on line 1.
      4 FUN1 5
SYNTAX ERROR
         ∨
FUN2[1] +.⍎ THIS LINE WILL SUSPEND ON SYNTAX ERROR
      )SI
FUN2[1] *  }
FUN1[1]    }           Second suspended calling sequence.
FUN2[1] *  }
FUN1[1]    }           First suspended calling sequence.
      →
      )SI
FUN2[1] *  }           First suspended calling sequence (environment
FUN1[1]    }           of second sequence is removed from the work-
                       space).
      →
      )SI
(blank)                State indicator is empty.
```

SIV COMMAND

syntax:)SIV

action: The action of SIV is similar to SI, but in addition to providing the function call names and line numbers, the local variables (including labels, arguments, and result) for each function call are listed.

example: using tbe same functions as the example in)SI :

```
      2 FUN1 3
SYNTAX ERROR
         ∨
FUN2[1] +.
      )SIV
FUN2[1] *
C       R  <---------  Note: not the same variable
FUN1[1]
A       B       R       Z
      )VARS
                       Note: no global variables
      A
2
      B
3
      R                No value has been assigned to the
VALUE ERROR            result variable for FUN2
        ∨
        R
```

STOP COMMAND

The STOP command provides a useful debugging tool for allowing examination of the function environment at strategic points in the function.

syntax:)STOP <function name> [<function line numbers (stop list)>]

action:
- The line numbers in the stop list are added to previously set line stops (if any).
- The function is modified so that it will be suspended prior to starting execution of the lines specified.

consequence: If during subsequent execution of the named function a stop-designated line is encountered for execution, suspension of the function occurs on that line prior to its execution.

The function name followed by the line number in brackets is output, followed by a request for input for immediate execution.

notes:
- The line numbers need not be in order in the stop list.
- If line 1 appears in the stop list, suspension occurs initially before any lines of the function are executed. In this case, all local variables are undefined except for the arguments. However, any masking of the global environment will have taken place.
- If the function is subsequently edited and additional lines are inserted or others deleted so as to change the line numbers of the lines designated in the stop list, the lines originally designated, even though their line numbers may now be different, are the ones on which suspension will occur.
- If a stop-designated line is edited, the stop designation is removed.
- If a stop-designated line is deleted, the designation is removed from the stop list.
- If a function with a stop list is copied to another workspace, the copied version will have a stop list.
- Complete removal of stop control for a function is provided by issuing a STOP command for the function with an empty stop list:

)STOP <function name>

error reports: OBJECT NOT FOUND

```
      ∇R←A FUN3 B
[1]   R←A+2×B
[2]   R←R⌈B*2
[3]   ∇
      )STOP FUN3 2
      5 FUN3 3
FUN3[2]
      )SI
FUN3[2] *
      A
5
      B
3
      R
11
      →2
11
      )SI

      )STOP FUN3
      5 FUN3 3
11
```

ENVIRONMENTAL PARAMETERS

ORIGIN COMMAND

syntax:)ORIGIN { 0 | 1 }

action: (a) 0 or 1 supplied.

- ORIGIN is set to the value supplied.
- The previous value of ORIGIN is reported.

(b) no parameter supplied.

- The current value of ORIGIN is reported.

note:) CLEAR sets ORIGIN to 1.

example:

```
      )ORIGIN
1
      )ORIGIN 0
WAS 1
      )ORIGIN
0
```

DIGITS COMMAND

syntax:)DIGITS [<integer>] $1 \leqslant \text{Integer} \leqslant 15$

action: (a) Valid parameter supplied.

- DIGITS is set to the value specified and the previous value is reported.

(b) no parameter supplied.

- The current value of DIGITS is reported.

consequence: DIGITS is used in numeric element formatting in formatting output displays and by the format primitive function (see Displaying Data.)

DIGITS is the maximum number of significant digits that can appear in a numeric element representation display.

note:)CLEAR sets DIGITS to 10.

example:

```
      )DIGITS
10
      )DIGITS 12
WAS 10
      )DIGITS
12
```

SEED COMMAND

syntax:)SEED [<integer>] $0 < \text{integer} < (2*48) - 1$

action: (a) Valid parameter supplied.

- SEED is set to the integer specified if $\geq 2*47$, otherwise the integer supplied is multiplied by a power of 2 sufficient to create a SEED such that $(2*47) \leq \text{SEED} < (2*48) - 1$
- The previous value of SEED is reported.

(b) no parameter supplied.

- The current value of SEED is reported.

examples:

```
      )SEED
1.925270759E14
      )SEED 129653
WAS 1.925270759E14
      )SEED
2.784278974E14
```

note:)CLEAR sets SEED to 192527075924404.

Valid Settings for SEED

The randomness of generated numbers is very dependent on the setting of SEED. Good randomness is achieved by numbers whose binary representation contains a fairly even distribution of ones and zeros.

Zero, powers of 2 and small numbers should not be used.

When to set SEED

While debugging an APL program that uses the primitive functions Roll or Deal it is highly desirable that the same sequence of random numbers be generated on each test, so that successive sets of results may be readily compared. This can be accomplished by resetting the SEED to the same value prior to each test.

An alternative procedure would be to)SAVE the workspace prior to each test; then)LOAD the saved workspace after execution and evaluation of each test, but prior to modifying any functions or test data.

LIBRARY FACILITIES

Each APL user is provided with facilities for preserving his user environment (the active workspace) at any point in a session as a permanent library workspace. This allows him to subsequently reinstate that workspace as the active one, thus reestablishing the environment exactly as it was when saved.

A user may maintain as many saved workspaces as he wishes in his user private library. Each stored workspace has a workspace identification (ID) by which it can be referenced. Facilities exist for listing the ID's of workspaces in any user's library, for updating or deleting individual workspaces in a user's own library, and for incorporating specified objects or groups from stored workspaces in any library into the currently active one.

In addition, the user is provided with a security of access to, and erasure or modification of, his saved workspaces by a password and user key facility.

WORKSPACE IDENTIFICATION

<workspace ID> := [<library ref>] <workspace name> [:<password>]

Every workspace has an identification (ID) consisting of:

- a library reference (owner's user ID).
- a workspace name.
- an optional password.

The library reference consists of one of the following:

- * (references the APL standard public library)
- * <alpha library owner's user ID>
- [*] <numeric library owner's user ID>

The workspace name is formed according to the same rules as apply to an identifier, but in addition is restricted to the character set and number of characters allowed by the host operating system. Currently, this restriction is 7 or fewer characters selected from the set A - Z, 0 - 9. The password is formed according to the same rules as the workspace name.

Note that spaces are not allowed between the asterisk and the library ID, or between the colon and the password.

Defaults

If the workspace ID is omitted in a command which references a workspace, the workspace ID defaults to that of the currently active workspace.

If the library reference is omitted, it defaults to the user's own library.

If the password is omitted, it defaults to no password.

Reports

Several commands report workspace ID's. In such cases, the following rules are used:

- The library ID is not reported.
- The password (if any) is never reported.
- If there is no workspace ID, it is reported as CLEAR.

NOTE:

1. A workspace initially saved without a password cannot have a password added by a subsequent)SAVE command.
2. A workspace initially saved with a password cannot have the password removed or changed by a subsequent)SAVE command.
3. However, the workspace password condition can be changed by loading the workspace, dropping it, and then saving it again with a new or different password or without a password.
4. The creator of a workspace need not use passwords for referencing his own workspace.

SAVE COMMAND

syntax:)SAVE [<workspace ID>]

action:

- A workspace identical to the currently active workspace is created in the specified library.
- The created workspace is designated with the workspace ID supplied or, if not supplied, with the workspace ID of the active workspace.
- Any previous workspace in the user's library bearing the ID of the newly created one is dropped.
- The name of the active workspace is changed to that given in the SAVE command, if supplied.
- Upon successful completion of the command, the following is reported:

  <workspace ID> SAVED <date><time>

error reports:

- IMPROPER LIBRARY REFERENCE

  an attempt was made to save a workspace in another user's library.

- NOT SAVED - THIS WS IS <active workspace ID>

  An attempt was made to SAVE a workspace under an ID of a currently existing library workspace while the active workspace ID was different. (This protects one from inadvertently overwriting a desired library workspace. ...If such action is intended, precede the SAVE command with a)WSID command (q.v.) supplying the ID desired.)

- NOT SAVED - THIS WS IS CLEAR

)SAVE with no parameters was issued with an active workspace having no workspace ID.

consequences:

- If a SAVED workspace ID includes a password, subsequent referencing of the workspace must include the password.

examples:

```
      X←3
      )SAVE XIS3                          (save as workspace named
                                           "XIS3", and set WSID to same)
XIS3  SAVED - 73/05/09. 15.37.04.
      )SAVE                               (resave under same name)
XIS3  SAVED - 73/05/09. 15.37.16.
      )SAVE XIS3:Y                        (resave with password)
XIS3  SAVED - 73/05/09. 15.38.21.
      )CLEAR
CLEARED WS
      Y←1+X←3



      )SAVE XIS3
NOT SAVED - THIS WS IS CLEAR              (XIS3 already exists)
      )WSID XIS3                          (declare this WS to be "XIS3")
WAS CLEAR
      )SAVE                               (and save it)
XIS3  SAVED - 73/05/09. 15.43.12.
```

LOAD COMMAND

syntax:)LOAD [<workspace ID>]

action:

- A search is made for the library indicated.
- If the library is found, a search is made for a workspace with workspace name as indicated.
- If the workspace is found and includes a password in its ID, a check is made for a match with the password supplied.
- If found, the indicated workspace is loaded as the active workspace, replacing the previous environment of the active workspace.
- The active workspace ID becomes the ID of the loaded workspace.
- Upon successful completion of the command, the following is reported:

  <workspace ID> SAVED <date (YY MM DD)><time (HH MM SS)>

error reports:

- <workspace ID> NOT FOUND

  no library could be found from the reference given.
  password does not match.
  no workspace by that name in referenced library.

- APL SYSTEM ERROR 1.

  workspace is damaged.

examples: user A logs in

```
      X←3
      )SAVE XIS3
XIS3  SAVED - 73/05/09. 15.44.59.        (time stamp)
      )CLEAR
CLEAR WS
      )VARS

      )LOAD XIS3
XIS3  SAVED - 73/05/09. 15.44.59.        (time stamp)
      )VARS
X
      X
3
      X←2                                (change X)
      )LOAD                              (reload)
XIS3  SAVED - 73/05/09. 15.44.59.
      X
3                                        (X restored)
      )SAVE XIS3:ABC
XIS3  SAVED - 73/05/09. 15.57.04.
      )OFF                               user A logs off
```

```
                                         user B logs in
      )LOAD XIS3
XIS3 NOT FOUND                           (password not given)
      )LOAD XIS3:PQR
XIS3 NOT FOUND                           (password incorrect)
      )LOAD XIS3:ABC
XIS3  SAVED - 73/05/09. 15.57.04.
```

COPY COMMAND

syntax:)COPY [<workspace ID> [<object list>]]

action:

- A search is made for the workspace indicated as for)LOAD.
- If found, the specified objects are searched for in the workspace global environment and, if found, copied into the active workspace, replacing any existing global object in the active workspace having the same name.
- If a specified object is found to be a group definition in the referenced workspace, then in addition to copying the group definition, all referents in the group definition are copied. If one of the referents is another group definition, it and all its referents are copied. Indirect referents to all levels of group structuring are thus copied.
- If no object list is provided, all global objects in the referenced workspace are copied.
- Successful completion of the command results in the report:

  <workspace ID> SAVED <date> <time>

note: Only global objects are copied. The function local environments, state indicator, stop lists and environmental parameters cannot be copied, and those in the active workspace are undisturbed.

error reports: <workspace ID> NOT FOUND

as for)LOAD

OBJECTS NOT FOUND
<identifier list>

the objects reported in <identifier list> could not be found in the referenced workspace.

APL SYSTEM ERROR 1

the referenced workspace is damaged.

note: Objects in the object list that have the same name as functions that are pendant or suspended in the active workspace will not be copied.

examples:

```
      X←3
      Y←4
      )SAVE XIS3
XIS3  SAVED - 73/05/09. 16.06.54.
      X←2
      Y←3
      Z←5
      )COPY XIS3                          (copy entire WS)
XIS3  SAVED - 73/05/09. 16.06.54.         (time saved)
      X,Y,Z
3     4     5                             (X, Y, & Z restored)
      X←2
      Y←3
      )COPY XIS3 Y                        (copy Y only)
XIS3  SAVED - 73/05/09. 16.06.54.
      □←P←X,Y,Z                           (create P)
2     4     5                             (Y only restored)
      )GROUP GRP1 X Z A                   (create a group)
      )SAVE                               (save it)
XIS3  SAVED - 73/05/09. 16.07.27.
      )CLEAR
CLEARED WS
      )COPY XIS3 GRP1 P                   (copy GRP1 & P)
XIS3  SAVED - 73/05/09. 16.07.27.
      )VARS
P       X       Z                         (A is not defined in "XIS3")
      )GRPS
GRP1
      )GRP GRP1                           (But A is still part of the group
                                           definition)
A       X       Z
```

PCOPY COMMAND

syntax:)PCOPY [<workspace ID> [<object list>]]

action: Action is identical to)COPY exept that objects whose names are identical to the names of objects in the active global workspace are not copied, thus protecting the objects already there.

error reports: <workspace ID> NOT FOUND - as for)LOAD

OBJECT NOT FOUND - as for)COPY

<identifier list>

note that objects which would have been prevented from being copied if found, are nonethless reported if not found.

examples:

```
      X←3
      Y←4
      )SAVE XIS3
XIS3  SAVED - 73/05/09. 16.10.39.
      X←2
      )ERASE Y
      )PCOPY XIS3
XIS3  SAVED - 73/05/09. 16.10.39.
      X,Y
2       4                              (Y only restored)
```

DROP COMMAND

syntax:)DROP [<workspace ID>]

action:

- A search is made in the specified library for a workspace with the specified name.
- If a workspace is found and includes a password in its ID, a check is made for a supplied matching password.
- If found, the workspace is removed from the library.
- The date and time when dropped are displayed to indicate successful execution of this command.

error reports:

- <workspace ID> NOT FOUND

  password does not match.

  no workspace by that name in the user's private library.

- IMPROPER LIBRARY REFERENCE

  an attempt was made to reference a library other than the user's own.

examples:

```
      X←3
      )SAVE XIS3                          (create it)
XIS3  SAVED - 73/05/09. 16.01.14.         (timestamp)
      )LOAD XIS3                          (load it)
XIS3  SAVED - 73/05/09. 16.01.14.         (time saved)
      )DROP XIS3                          (drop it)
73/05/09. 16.01.14.                       (time dropped)
      )LOAD XIS3
XIS3 NOT FOUND                            ("XIS3 no longer exists)
      )SAVE XIS3:ABC
XIS3  SAVED - 73/05/09. 16.01.58.
      )DROP XIS3:ABC
73/05/09. 16.01.58.
```

WSID COMMAND

syntax:)WSID [<workspace name>]

action: (a) No parameters provided. The active workspace ID is reported.

example:

```
      )WSID
TEST
```

note: an empty ID is reported as CLEAR; this does not necessesarily mean a CLEAR workspace:

```
      )CLEAR
CLEARED WS
      A←5
      )WSID
CLEAR
```

(b) If a workspace ID is provided, it becomes the ID of the active workspace. The previous active workspace ID is reported.

example:

```
      )WSID MODEL
WAS CLEAR
      )WSID
MODEL
```

examples:

```
      X←3
      )WSID
CLEAR
      )SAVE XIS3
XIS3  SAVED - 73/05/09. 16.12.42.
      )WSID
XIS3
      )CLEAR
CLEARED WS
      )COPY XIS3                    (restore objects in "XIS3")
XIS3  SAVED - 73/05/09. 16.12.42.
      )SAVE
NOT SAVED - THIS WS IS CLEAR
      )WSID XIS3                    (declare WS same as "XIS3")
WAS CLEAR
      )SAVE                         (now it can be saved)
XIS3  SAVED - 73/05/09. 16.13.27
```

LIB COMMAND

syntax:)LIB {<library ref>}

action:

- The workspace names of the user's own library are listed if no parameter is supplied.
- Password protected workspaces are listed in the above case, but the passwords are not listed.

examples:

```
      )LIB

      X←3
      )SAVE XIS3
XIS3  SAVED - 73/05/09. 16.15.41.
      )LIB
XIS3
      )SAVE NEWX:ABC
NEWX  SAVED - 73/05/09. 16.17.03.
      )LIB
NEWX    XIS3
```

LIBRARY ACCESS

Suppose the above user, JOE, logs out, and FRED logs in, and does the following:

```
      )LIB

      )LIB *JOE
NEWX XIS3
```

```
      )LOAD *JOE XIS3
XIS3  SAVED - 73/05/09. 16.17.03.
      X
3
      )LOAD *JOE NEWX
NEWX NOT FOUND
      )LOAD *JOE NEWX:ABC
NEWX  SAVED - 73/05/09. 16.17.03.
      )LIB


      )WSID
NEWX
      )SAVE *JOE NEWX:ABC
IMPROPER LIBRARY REFERENCE


      )DROP *JOE  NEWX:ABC
IMPROPER LIBRARY REFERENCE
      )WSID MYX
WAS NEWX
      )WSID
MYX
      )SAVE
MYX  SAVED - 73/05/09. 21.32.16.
      )LOAD NEWX
NEWX NOT FOUND
      )LOAD MYX
MYX  SAVED - 73/05/09. 21.32.16.
      )LOAD *FRED MYX
MYX  SAVED - 73/05/09. 21.32.16.
```

USER LIBRARY WORKSPACE SECURITY

A user may quite safely divulge his system user ID to some other user so that the latter may load or copy any or all objects from unprotected workspaces. The alternate user may interrogate workspaces available to him by)LIB [<library ref>].)SAVE or)DROP is not permitted using an alternate library reference. A user can save workspaces only in his own user library and can only)DROP his own workspaces.

Knowing a person's system user ID is in itself not sufficient to allow that person to log-on under that ID. To do so, he requires that person's system log-on password. This password should never be revealed to anyone under any circumstances! Possession of this password permits the holder complete freedom of access to workspace libraries under the corresponding user ID.

THE CONTINU WORKSPACE

A user may terminate his session with the system command)CONTINU. This effects a)SAVE to a user library workspace with the name CONTINU and then terminates the session. A)SAVE to CONTINU is also permitted.

At the start of every session, an automatic LOAD of the CONTINU workspace takes place if one exists.

start of session:

```
        APL
APL*CYBER.  V1.0   TASC

73/05/19    14:19:03

CONTINU  SAVED - 73/05/18   16:29:56
      )WSID
CONTINU
```

Note that the date and time CONTINU was last saved are shown in the same manner as in response to a)LOAD command.

The CONTINU workspace can be referenced in any system command that validly references a workspace: i.e.,)SAVE ,)LOAD ,)DROP ,)COPY ,)PCOPY .)LIB will list CONTINU if it exists.

TERMINATING AN APL SESSION

An APL session can be terminated in any of the following ways:

1. With the system command)OFF
2. With the system command)CONTINU
3. With the system command)SYSTEM
4. By intentionally disconnecting the terminal connection.
5. By unintentional dropping of the terminal connection due to a variety of transmission difficulties.
6. By a computer malfunction or operating system problem occurrence.

In case 2, the active workspace will be saved in the user's CONTINU workspace.

)SYSTEM returns control to the operating system command mode. All other cases cause a log-out of the user from the system.

Cases 1, 2 and 3 are termed normal session termination and respond with a display of session statistics and terminating status information.

Cases 4, 5 and 6 do not generate a response and do not save the workspace.

```
      )OFF
CPU TIME      3.560 SEC.
WORKSPACE      1493 WORDS.
```

NOTE: In the event of a dropping of the terminal connection, recovery may be attempted by redialing the computer, typing one's user name and password and in response to the display of RECOVER/SYSTEM: type RECOVER. If recovery is possible, enter a carriage return. The APL session will be in the state in which it was left, save data lost during the transmission.

DISPLAY DEVICE PARAMETERS

The two display device parameters maintain their settings, unless specifically altered, for the entire APL session. They are WIDTH and LINES. Default settings are assigned at the start of a session based on the declared terminal type (including batch). These parameters do not reside in the active workspace, and thus cannot be saved in library workspaces.

WIDTH COMMAND

)WIDTH [<integer>] 30 ⩽integer ⩽254

action:

(a) valid parameters supplied.

- WIDTH is set to the value supplied.
- The previous value of WIDTH is reported.

(b) no parameter supplied.

- The current value of WIDTH is reported.

examples:

```
      )WIDTH 50
WAS 72
      )WIDTH
50
```

consequence: Until again changed later in the session, all displayed output will be formatted in lines not exceeding WIDTH characters in width. Data which otherwise would appear on the same line will be continued on the following line or lines. The line continuation format for the declared display device will be used.

default value: See appendix for default values for specific terminals.

LINES COMMAND

)LINES [<integer>] 0 ⩽integer ⩽¯1+2*48

action:

(a) valid parameter supplied.

- LINES is set to the value supplied.
- The previous value of LINES is reported.

(b) no parameter supplied.

- The current value of LINES is reported.

consequence: If the setting of LINES is non-zero, output is displayed on the output device in "pages" LINES lines long. At the end of each "page", the display will halt and request go-ahead according to the device type. This consists of a request for input with a '?' at the left margin. Any input will then cause the display to continue. The last such "page" does not request go-ahead, as the display is complete. Neither does "fill" to the end of the page occur.

If the setting of LINES is zero, no paging occurs.

examples:

```
      )LINES 0
WAS 14
      )LINES
0
```

This provides users on terminals with volatile displays a chance to peruse an entire screen of information for any desired period. To continue displaying output, the user inputs a blank line.

Additional halts will occur on subsequent screensful of output until the entire display has been sent.

note: The remainder of the display is aborted by pressing the 'BREAK' or 'ATTN' key.

default value: The default setting of lines will be equal to the line capacity of the display less 2 (to allow for the prompt line and input line).

The most usual non-default setting of LINES is zero which causes continuous scrolling of output without halts for the entire display. Zero is the default setting for all hard copy terminals.

TERMINAL ACCESS TO APL*CYBER SYSTEM ON KRONOS A

1. Turn on terminal; set for on-line use. Set to half duplex except as noted in 8(b). If your terminal is "hard-wired" to the computer system, go to step 6.

2. Turn on data set. Set to half duplex if acoustic coupler.

3. Dial the computer telephone number.

4. Wait for a steady high pitched tone.

5. Set for data; with acoustic couplers place the reciever in the proper position in the coupler cradle.

6. This step is required only if the Terminal Protocol Routine is included in the KRONOS operating system on the CYBER computer you are attempting to use. Otherwise the display stated below will appear automatically. Depending on the type of terminal, key one of the following:

IBM 2741 correspondence with APL ball:	A attention
IBM 2741 correspondence with standard ball:	{carriage return / attention}
Memorex 1240 with APL belt:	M
All other terminals:	T

The system will return the following output display:

```
73/01/15. 19.51.57.     (current date and time)
KRONOS TIME SHARING SYSTEM - VER. 2.1.
USER NUMBER:
```

7. Key in your KRONOS user number.

```
JOEBLOW     return
```

The system responds with:

```
PASSWORD
```

On non-destructive display terminals, the second line consists of 9 multiply overstruck characters. On CRT displays (such as the CONTROL DATA 713) the net result will appear as XXXXXXXXX. Backspacing to the first line position then occurs.

8. Key in your KRONOS user password It is extremely important that this password not be revealed to any other person. On non-destructive display terminals, this password will be keyed directly on top of the mask provided, so as to make the password illegible.

   On CRT display terminals this cannot be done, as each displayed character replaces the previously displayed one. Instead, use one of the following procedures:

   (a) Place one hand over the screen, key in the password followed by return with the other hand, then key the clear key which blanks the screen.

   (b) Set the switch at the back of the terminal to full duplex permanently if using an acoustic coupler. Set the switch on the coupler to half duplex except when keying the password, for which use the full duplex setting. This will cause the characters to be sent to the KRONOS system without being displayed on the terminal. Don't forget to reset the coupler to half duplex after this is done.

   (c) Reduce screen intensity and key in the password, now key the return and then clear and remember to reset the screen intensity when done.

   If an error is detected in the entered KRONOS user number or password, the system responds with the error message IMPROPER LOG IN, TRY AGAIN.

   This is followed by a repeat of the prompt USER NUMBER: whereupon steps 7 and 8 must be repeated.

   An automatic disconnect from the system will occur after three unsuccessful attempts to do this properly.

9. If steps 7 and 8 are completed satisfactorily the user is said to be 'logged in'. This is confirmed by the response TERMINAL: port number, TTY.

   The system then prompts the user to indicate which system he wants:

   RECOVER/SYSTEM:

   If the APL system is desired, the response to be entered depends on the type of terminal.

CONTROL DATA 713:	APL, 713
TELETYPE 33:	APL, TTY
TELETYPE 38 with APL:	APL, T38
TEKTRONIX 4013:	APL, TEK
IBM 2741	APL, 2741
MEMOREX 1240	APL, MEM
FULL ASCII	APL APL, ASC

The system replies to a proper request for the APL system with:

APL*CYBER, V1.0 T<terminal type>

73/01/15. 19.55.04. (logged-in date and time).

If a workspace with the name CONTINU existed in the user's private library at the termination of the previous session, that workspace is loaded as the active workspace to commence the current session. This fact is conveyed with the message:

CONTINU SAVED - 73/01/15. 19.25.23.

(last saved date and time)

At this point the user is now in direct communication with the APL*CYBER system.

COMMUNICATING APL CHARACTERS B

METHODS

The APL characters are summarized in Table B-1. Communicating these characters between a terminal (or batch input and output device) and the APL*CYBER system is achieved in one or more of the following ways:

1. Terminal keys corresponding to APL characters communicate those characters when struck.
2. Specific terminals may have a certain key defined as a substitute for a certain APL character and convey that character when depressed. Such particulars are listed under the appropriate section of supported terminals.
3. On non-destructive display terminals (i.e., hard copy or storage tube) which are equipped with a backspace key, certain APL characters may be communicated by overstriking (explained below).
4. A scheme of three character mnemonics exists for conveying any desired APL character and can be used on any terminal or batch I/O device.

Output displays will, for each required character, utilize one of the above methods, according to device capabilities, in the preferred order as listed.

OVERSTRIKES

On terminals with a non-destructive overstrike, such as hard copy or storage tube terminals, repositioning to the line position of a previously keyed character and keying a second non-blank key (called overstriking) creates a compositely formed display graphic. If this graphic is a reasonable facsimile of the symbol for an APL character, that character is conveyed; otherwise the character is illegal, and is converted to the canonical 'bad' character. Note that underscored alphabetics and underscored Δ (delta) are equivalent symbols for lower case alphabetics and δ respectively and may be formed by overstriking. On terminals with a standard APL keyboard, all but a very few special characters can be conveyed by direct keying or overstriking.

Note that as a consequence of the Visual Fidelity criterion, the keying sequence used in forming overstrikes is immaterial. Also, repeated overstriking the same key in the same line positon still conveys the same character. Overstriking with the space bar does not change the character conveyed.

MNEMONICS

On terminals not equipped with a standard APL keyboard, and as an alternative for any type of terminal, desired APL characters can be conveyed by means of mnemonics.

Mnemonics exist for the entire APL character set except for the following 46 characters which are standard on any terminal:

A...Z 0...9 . , ( ) + - = * / and space

Further, there are no mnemonics for backspace, return, or any other non-graphic characters.

All mnemonics are formed by a three-character combination consisting of a dollar sign ($) followed by two upper case alphabetic characters. The $ character acts as a flag character and conveys that it along with the following two characters are to be treated as a group which compositely represents a single APL character. The $ character is standard on all terminals except some of those with APL keyboards. An overstrike combination exists to convey the $ character in this case. If the $ character itself is desired as a literal character, the mnemonic for dollar sign can be used.

Also, a dollar sign is considered literal if it is not followed by an upper case alphabetic (e.g., '$1.50').

The two upper case alphabetic characters following the $ character have been chosen by the following scheme to aid in remembering them:

- Lower case Roman alphabetic characters are conveyed by the double appearance of the corresponding upper case character.
- If the APL character is used only as a character and not as a primitive function designator the two characters are an abbreviation for the name of the symbol.
- If the APL character is used as a primitive function designator, but is a character which has a generally known name, the two characters are an abbreviation for the name of the symbol.

- If the APL character is used as a primitive function designator and is a character for which no name exists, or which is not widely known, the two characters are an abbreviation for the name of the primitive function. If the function is known by more than one name, an abbreviation of the most frequently used name is chosen.
- For those APL primitives for which an alternate APL character exists, implicitly indicating 'first' for the indicated ordinal processing of the right argument, the two characters used are obtained from the two characters used in the mnemonic for the APL character which represents the standard form of the function call, replacing the second character by the next higher in the alphabet.

The complete set of APL character mnemonics is listed in Table B-1.

COMPATIBILITY

It should be noted that, no matter how APL characters are communicated from whatever type of terminal, the APL system converts each APL character representation to a standard internal representation for processing. SAVE'd workspaces are also stored in this format.

This means that workspaces created while on one type of terminal may subsequently be loaded while on a different terminal type. Compatibility of workspace contents is thus ensured for users of all terminal types.

TABLE B-1. APL CHARACTER SET

Graphic	Mnemonic	Meaning	Graphic	Mnemonic	Meaning
?	$QU	QUery	A thru Z	--	(upper case alphabetics)
⍵	$OM	OMega			
∈	$EP	EPsilon	a(A) thru z(Z)	$AA thru $ZZ	(lower case alphabetics)
⍴	$RO	RhO			
~	$TL	TiLde	0 thru 9	--	(numerics)
↑	$TA	TAke			
↓	$DR	DRop		--	(space)
⍳	$IO	IOta			
○	$CI	CIrcle	"(¨)	$DQ	Double Quote
⌽	$RT	RoTate	¯(^,¬)	$NG	NeG
⍉	$RU	(reverse indexed rotate)	<	$LT	Less Than
⍉	$TP	TransPose	≤	$LT	Less than or Equal
*	--	asterisk	=	$EQ	EQual
⍟	$LG	LoG	≥	$GE	Greater than or Equal
→	$GO	GOto	>	$GT	Greater Than
←	$IS	IS	≠	$NE	Not Equal
			∨	$OR	OR
⍺	$AL	ALpha	⍱	$NR	NoR
⌈	$MX	MaX	∧	$AN	ANd
⌊	$MN	MiN	⍲	$ND	NanD
_	$UL	UnderLine	-	--	minus
∇	$DL	DeL	+	--	plus
⍫	$LD	Locked Del	÷	$DV	DiVide
⍒	$DG	DownGrade	⌹	$XD	matriX Divide
∆	$DT	DelTa	×	$ML	MuLtiply
δ(∆)	$DU	Delta Underscored (lower case delta)			
⍋	$UG	UpGrade			

TABLE B-1. APL CHARACTER SET (Cont'd)

Graphic	Mnemonic	Meaning
⍬	$NL	NuLl
'	$QT	QuoTe
!	$EX	EXclamation mark
⍞	$QD	QuaD
⎕	$QP	Quad-Prime
(	--	paren (left parenthesis)
)	--	close (right parenthesis)
[	$OB	Open Bracket (sub)
]	$CB	Close Bracket (bus)
⊂	$ID	ImbeD
⊃	$IN	INclusion
∩	$IX	InterseXion (?!)
⍺	$LP	LamP
∪	$UN	UNion
⊥	$BV	Base Value
⊤	$RP	RePresentation
⌶	$IB	I-Beam
|(¦)	$MD	MoDulus
;	$SC	SemiColon
:	$CL	CoLon
\	$BS	BackSlash
⍀	$BT	(reverse indexed backslash)
,	--	comma
.	--	dot
/	--	slash
⌿	$SM	(reverse indexed slash)

Graphic	Mnemonic	Meaning
#	$NM	NuMber sign
$($)	$DO	DOllar sign
%	$PC	PerCent sign
&	$AM	Ampersand
@	$AT	AT sign
{	$LB	Left Brace
}	$RB	Right Brace
¢	$CT	CenT sign
⋄	$DM	DiaMond
⊣	$RK	Right tacK
⊢	$LK	Left tacK
`	$GV	GraVe accent
⍎	$EV	EValuate
⍕	$FM	ForMat
⍪	$CN	(reverse indexed comma)

Special Characters:

(⎕)	$G.	(quad-prime escape)
	$BC	(canonical bad character)

NUMERIC REPRESENTATION ON CYBER COMPUTERS C

1. An exact representation for zero exists.

2. The sum of any selection of any 47 consecutive terms of the power series of 2

   $$2^{-1023}, 2^{-1022}, 2^{-1021}, \ldots 2^{-1}, 2^{0}, 2^{1}, \ldots 2^{1068}, 2^{1069}$$

   can be represented exactly.

3. The negation of any such number can also be represented exactly.

4. Any number whose magnitude is larger than the sum of the last 47 terms of the above series

   $$(\text{i.e., } 2^{1070} - 2^{1022})$$

   cannot be represented, and is not in the domain of definition of any numeric APL function.

5. Any number whose magnitude is less than 2^{-1023} will be approximated by the representation for zero by all numeric APL functions.

6. All other real numbers will be represented by the exact representation of the approximation to the desired value obtained by summing the 47 most significant terms of the value expressed as a power series of 2.

   As a consequence of the above, the following are true:

   - The numbers with the largest magnitude which can be represented in CYBER computers and for which APL numeric functions are defined are:

     $\pm$1.2650140831706E322 i.e., $\pm(2^{1070}-2^{1022})$

   - The numbers with the smallest non-zero magnitude which can be represented in CYBER computers exactly are:

     $\pm$3.1316E⁻294 i.e., $\pm 2^{-1023}$

INDEX

COMMENT SHEET

MANUAL TITLE CONTROL DATA® APL*CYBER

Reference Manual

PUBLICATION NO. 19980400 REVISION B

FROM: NAME:

BUSINESS ADDRESS:

COMMENTS:

This form is not intended to be used as an order blank. Your evaluation of this manual will be welcomed by Control Data Corporation. Any errors, suggested additions or deletions, or general comments may be made below. Please include page number references and fill in publication revision level as shown by the last entry on the Record of Revision page at the front of the manual. Customer engineers are urged to use the TAR.

CUT ALONG LINE

AA3419 REV. 11/69 PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Technical Publications Department

4201 North Lexington Avenue

Arden Hills, Minnesota 55112

CUT ALONG LINE

FOLD

FOLD

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.