CONTROL DATA
CORPORATION

CONTROL DATA®
STAR COMPUTER SYSTEM

APL* STAR REFERENCE MANUAL

REVISION RECORD

REVISION	DESCRIPTION
A	Original Printing.
(3-74)	
B	
(9-74)	Reflects Version 1.1.
Publication No. 19980800	

Address comments concerning this manual to:

Control Data Corporation
Documentation Department
215 Moffett Park Drive
Sunnyvale, California 94086

or use Comment Sheet in the back of this manual.

PREFACE

This is the reference manual for APL*STAR, Version 1.1. APL*STAR runs on all models of CONTROL DATA® STAR Series computers.

CAUTION

This product is intended for use only as described in this document. Control Data cannot be responsible for the improper functioning of undescribed features or unidentified parameters.

CONTENTS

APPENDIXES

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 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*STAR SYSTEM

The implementation of APL on STAR computers is known as the APL*STAR 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.
- 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*STAR system and exchange data and programs.

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.

≈ approximately equal

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 set of data elements which in general are multiply-ordered in a coordinated way.

COORDINATE

Each coordinated ordering is termed a coordinate and is designated by a canonical ordinal for reference purposes.

RANK

The number of coordinates is called the rank of the array.

LENGTH

Each coordinate has an associated non negative integer called its length which is the number of different order positions used in that ordering.

ORDER POSITIONS

Order positions for a given coordinate are those constituting the set of the first L ordinals. (see ORDINALS), where L is the length of that coordinate.

Each order position for each coordinate is assigned to an equal number of distinct elements, such that all elements have one such assignment.

The assignment of order positions to elements by all coordinates is such that the canonically ordered set of order positions for each element is unique.

CANONICAL RAVEL

The canonical ravel of an array is the single-ordered set of original array elements in canonical coordinate precedence sequence.

SHAPE

The shape of an array is the single-ordered set of coordinate lengths in canonical order.

EMPTY

An array is said to be empty if it has no elements. At least one of its coordinates has a length of zero.

DATA TYPES

Three data types are defined in APL*STAR: numeric, character and list. The value of a numeric data element is a single real number. The value of a character data element is a single character. The value of a list data element is the value of the array imbedded in the element (see IMBED).

CHARACTERISTIC DATA TYPE

All elements of an array in APL must be of the same data type, called the characteristic data type of the array. Even empty arrays have a characteristic data type.

VALUE OF AN ARRAY

The value of an array is the totality of its intrinsic attributes as characterized by its shape, canonical ravel and characteristic data type.

RANK TERMINOLOGY

An array of rank 0 is called a scalar. It has no coordinates and exactly one data element.

An array of rank 1 is called a vector. It has one coordinate and zero or more data elements.

An array of rank 2 is called a matrix. It has 2 coordinates and zero or more data elements.

An array of rank 3 or greater has no special name. It has as many coordinates as its rank and zero or more data elements.

RANK LIMITATION

The APL language does not define any limit to the rank of an array. However, the APL*STAR 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 RANK ERROR).

LIMITATION ON NUMBER OF ELEMENTS

The APL language does not define any limit to the number of elements in an array. However, the APL*STAR implementation currently will not allow the user to create arrays having more than 65,535 elements. An attempt to exceed this limit results in the error message NONCE ERROR.

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'.
- The character e may be used in place of the character E.

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

Numerals having any number of digits may be formed, but will only express the value represented by the 14 (in some cases 15) most significant digits. Proper scaling will always take place.

Numerals expressing a value beyond the number representation capability of STAR computers will result in the error message SYNTAX ERROR. (See appendix C.)

A numeric vector is expressed by juxtaposing two or more numerals 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 variable 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 4095 characters in length.
- The first character must be an alphabetic character (A to Z, a to z, Δ, ∂).
- 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          |          2+2
4                    |      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.

List arrays may not be displayed. An attempt to display a list array will result in a NONCE ERROR.

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*STAR is 8. 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
8
      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
8
      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
8
      )WIDTH 60
63
      X←.275396  14.3E3  692738  12345  678
      X
0.275396 14300 692738 12345 678
      )WIDTH 30
60
      X
0.275396 14300 692738 12345
      678                                  (display continuation indented)
      Y←3 2ρ42 1.7E9 ¯173.52 6.8345E¯10 .9 0
      Y
  42.00 1.7000E009
¯173.52 6.8345E¯10
   0.90 0.0000E000
      )DIGITS 4
5
      Y
 4.200E01 1.700E009
¯1.735E02 6.835E¯10
 9.000E¯1 0.000E000
```

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 their designators 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. "J" is used as a reverse index and follows the same rules.
- 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←Aϕ[K]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*STAR is the set of real numbers for which an exact or approximate representation exists on STAR 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. A scalar is treated as a one-element vector where singular conformability requires a vector argument. This process is known as scalar extension.

   Exceptions:

   - A scalar cannot be indexed.
   - The left argument of DYADIC IOTA must be a vector.
   - The argument of GRADE UP and GRADE DOWN must be a vector.

2. A one-element vector is treated as a scalar where singular conformability requires a scalar argument.

3. 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, and EXPAND
   - Both arguments of TRANSPOSE
   - Both arguments of MATRIX DIVIDE

   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 result 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*STAR allows the ORIGIN to be set to either 0 or 1.

The normal setting for ORIGIN in APL*STAR 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
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 subarrays are to be chosen. From this it is apparent that for an index K, and an array of rank N, the domain of K is:

$K \in \iota N$

If an index is not specified for the function, it defaults to the ordinal of the last coordinate namely:

$^{-}1 \uparrow \iota N$

The functions which may be so indexed are:

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

REVERSE INDEXING

All indexable functions as described above have an alternate designator which specifies reverse indexing. The corresponding alternate designators are:

/ → ⌿

\ → ⍀

f/ → f⌿

⌽ → ⊖

, → ⍪

If an index J is specified with a reverse index designator, the result obtained is equivalent to that obtained using the regular designator with an index $(\phi\iota N)[J]$

If no index is specified in this case, J defaults to $^{-}1\uparrow\iota N$ as did K so that the designated coordinate is

$$(\phi\iota N)[^{-}1\uparrow\iota N]\leftrightarrow 1\uparrow\iota N$$

i.e., the first coordinate.

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 STAR series computers can only be represented with 14 significant digits of accuracy (15 digits for integers with a magnitude less than 2*47).
- 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 $B \pm |B \times factor$.
- 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.

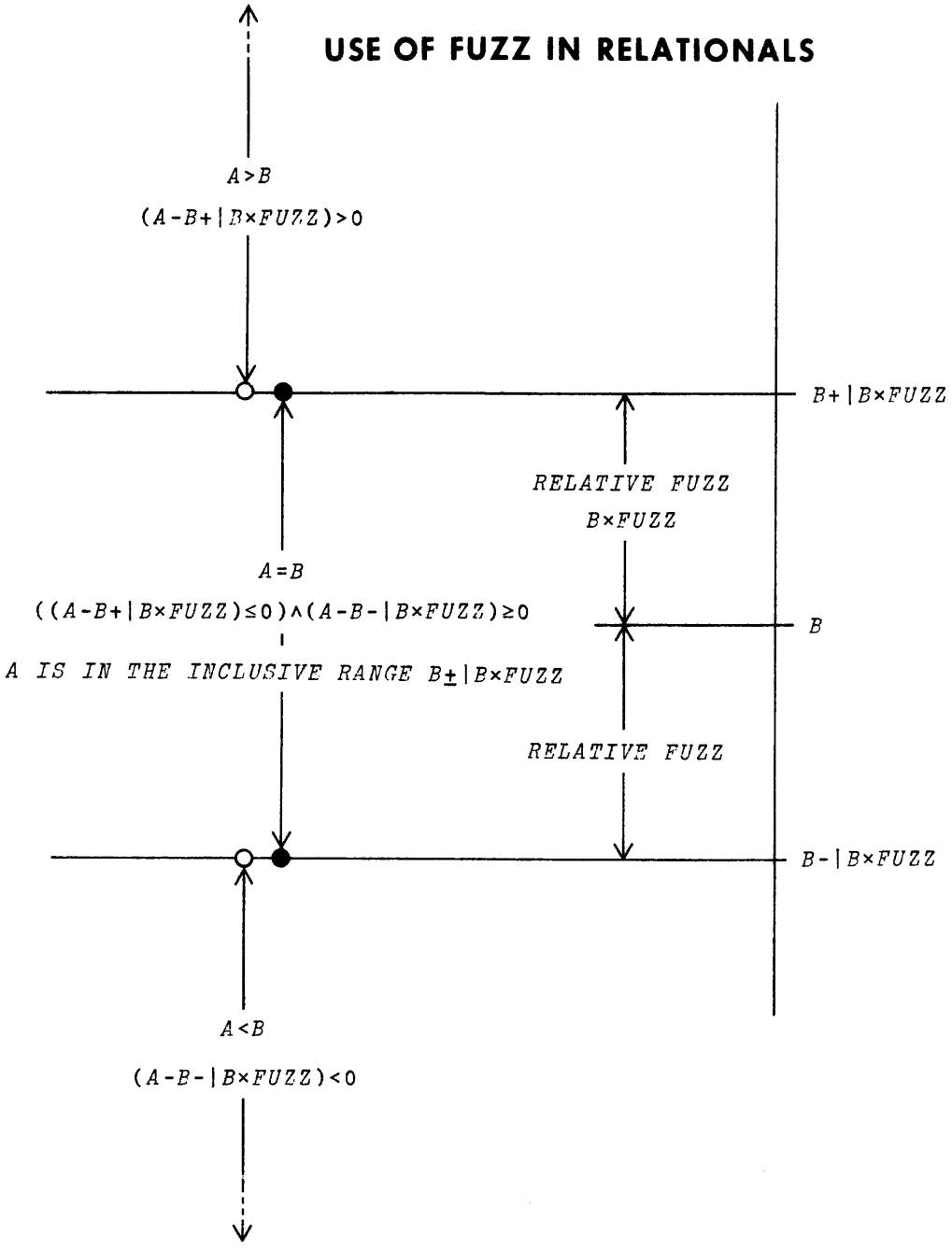
The following primitive functions also perform comparisons between data elements in the same manner as the relationals:

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

However, ⍋ and ⍒ do not use FUZZ.

The normal setting for FUZZ in APL*STAR is

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:

```
(⌊B)←→⌊B+FUZZ
```

CEILING

In a similar manner, ceiling operates as follows:

```
(⌈B)←→⌈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:

```
((⌈B)-⌊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*47.

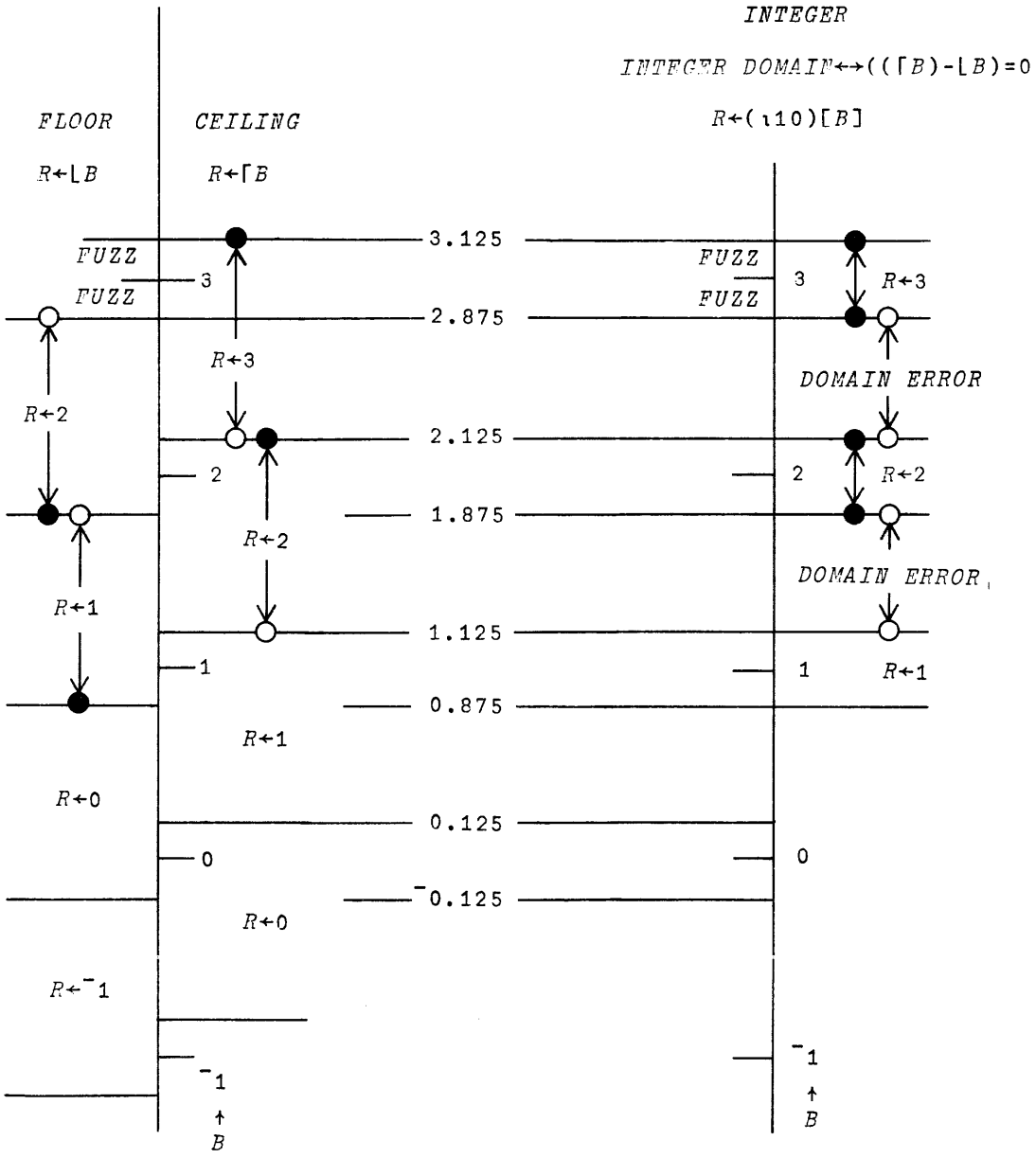
GENERAL NOTES

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*47.

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. For list arrays, the fill element is a zero length list vector.

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.

DYADIC RHO: RESHAPE

syntax: $R\leftarrow A\rho B$

domain: $A\geq 0$ and integer; A may be type character if empty.

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

$(\rho A)\leq 127$ (APL*STAR limitation)

$0=\times/A$ 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: $(,B)\leftrightarrow(\times/\rho B)\rho B$

Note: If A is empty and B is a list array, the result is not defined.

examples:

```
      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
      $: 2ρι0           (A must be empty if
                              B is empty)
```

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 <u>not</u> 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

examples:

```
      ρ5
(blank)
      ρ'A'
(blank)
```
(a scalar has an empty shape)

```
      ρ3 7.9 ¯3.2
3
      ρ'ABX73Y'
6
```
(the shape of a vector is the number of elements)

```
      ρ⍳0
0
      ρ''
0
```
(an empty vector has a shape of zero)

```
      ρρ7
0
      ρρ'A'
0
```
(the rank of a scalar is zero)

```
      ρρ4 7.2 5 3 ¯8
1
      ρρ'ABCD'
1
```
(the rank of a vector is one)

```
      ρ2 3ρ1 7 9 2 3 6
2 3
      ρ2 7 9ρ'XYZ'
2 7 9
```
(shape of higher rank arrays (see RESHAPE))

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
```

INDEXING

syntax: $R\leftarrow B[\,<\text{index list}>\,]$

The form of the index list for a B of rank N is

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

where each I_J is an expression of any rank, or may be elided. N-1 semicolons must appear.

domain: I_J ordinal

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

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

result shape: Let $RI\leftarrow(\rho I_1),(\rho I_2),\ldots,\rho I_N$

Then $(\rho R)=RI$ unless RI is empty and B is a list array (see below).

definition: The result is formed from elements selected from B as designated by the index list. The index list also governs the shape of the result and the position each selected element will occupy in the result. Each expression in the index list evaluates to an array of ordinals called the index for the corresponding coordinate. Each ordinal specifies a position of that coordinate. If no such position exists, an INDEX ERROR results.

Index expressions may be elided. In this case the index expression defaults to

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

that is, a vector of all position ordinals of coordinate J in position order.

An element is selected for every combination of coordinate ordinals. The position of each selected element in the result is determined as follows.

The first element of the canonical ravel of the result is that element of B having coordinate positions as indicated by the first ordinal in every canonically raveled index. This is followed by those

elements of B in which only the last coordinate position is changed to that of successive ordinals of the ravel of the last index. This is followed in like manner by those elements of B having coordinate positions corresponding to index ordinal combinations obtained by using succeeding elements from each index in turn until all combinations are exhausted.

The shapes of the indices do not affect element selection but compositely dictate the shape of the result.

Note that since the index elements are ordinals, the selection process and hence the result is ORIGIN dependent. (See ORIGIN).

If B is a list array, and RI is empty (i.e., a scalar is indexed from B), then R is the value of the array which was imbedded in the selected element of B. This is the inverse operation to that of the IMBED function (q.v.).

identity: `B←→(⊂B)[?1]`

```
      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)
      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]
3 8
      X[;,2]
3
8
      )ORIGIN 0
1
      X[1;1]
8
      X[0;0]
4


      B←2 3 4⍴⍳24
      B[1;2;1 2]
5 6
      B[1;;][2;]                    (indexing twice)
5 6 7 8
      B[,1;;][1;2 3;1][2]           (indexing 3 times)
9


      L←⊂'ABC'                      (create list - see IMBED)
      L[1]                          (Scalar indexing reveals
ABC                                        imbedded element)
      L[1][1 3]
AC
      ⍴⍴(1↑0⍴L)[?1]                 (test for L type list)
1                                        1 if list
      ⍴⍴(1↑0⍴'A')[?1]                    0 if not list
0
```

INDEXED SPECIFICATION

syntax: $R\leftarrow X[\underline{<\text{index list}>}]\leftarrow B$

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.
X is the name of a variable.
The form of the index list for an X of rank N is

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

where each I_j is an expression of any rank, or may be elided. N-1 semicolons must appear. B is the result of the most recent expression evaluation on the same line and must exist.

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

$(\rho\rho X)\geq 1$ (may not be circumvented by scalar extension)

Let $RI\leftarrow(\rho I_1),(\rho I_2),\ldots,\rho I_N$

Then $((1\neq\rho B)/\rho B)\leftrightarrow(1\neq RI)/RI$

Unless RI is empty and X is a list array (see below).

domain: B must be a of the same type as X unless RI is empty and X is a list array (see below).

I_J ordinal

definition: Indexed specification like ordinary specification is not a primitive function but a directive. The operation requires and uses the result value of the most recent expression evaluation on the same line but leaves that result undisturbed and available as the right argument to some subsequent function whose designator appears further to the left in the source line. Such a result may also be used by successively left occurring directives, all of which leave the result undisturbed.

Indexed specification assigns the value of each element of B to an element of the variable X as designated by the index list. Any element of X not so designated retains its current value.

Rules pertaining to the index list are the same as for INDEXING. In addition the following rules apply:

1. The additional conformability rule stated above.

2. The index list has bad form and the operation is not defined if multiple elements of B are specified to the same position in X i.e. required is $\wedge/1=+/(,I_J)\circ.=,I_J$

3. X must be an existing defined variable.

If X is a list array and RI is empty, (i.e., a scalar element of X is indexed), B is imbedded in the selected element of 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
1
      X
1 9 3
4 8 6
      X[;2]←4 5
1     X
1 9 4
4 8 5
```

```
      M←3 1 4⍴⍳12
      M[⍳2;1;⍳3]←1 2 3 1⍴0
      M
0 00  0  4

0  0  0  8

9 10 11 12
      X←4 5⍴'ABC'                (create list variable - see IMBED)
      X[2;3]←3 4⍴⍳12             (imbed into X [ 1;3] )
      X[1;4]                     (reveal X[ 1;4 ])
ABC
      X[2;3]                     (reveal X[ 2;3 ])
1  2  3  4
5  6  7  8
9 10 11 12
      ⍴X
4 5
      X[2;3][1;]
1 5 9
```

DYADIC COMMA: CATENATE

syntax: `R←A,[K]B`

`R←A⌿[J]B` (reverse indexed)

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

Three cases exist:

- `(⍴⍴A)=⍴⍴B`
- `(⍴⍴A)=1+⍴⍴B`

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

  `B1←(((K≠⍳⍴⍴A)\⍴B)+K=⍳⍴⍴A)⍴B`

- `(1+⍴⍴A)=⍴⍴B`

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

  `A1←(((K≠⍳⍴⍴B)\⍴A)+K=⍳⍴⍴B)⍴A`

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)←→(⍴B)+(K=⍳⍴⍴B)\(⍴A)[K]`

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, Reverse Indexing applies.

examples:

```
      2 3,4 5
 2 3 4 5
      X←2 3ρ1 2 3 4 5 6
      Y←2 3ρ7 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;]
 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
```

(first coordinate used)

```
      X,[0]Y
INDEX ERROR
      $: X,[0]Y
      )ORIGIN 0
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
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
4 5 6 5
      Z←2 2 2ρ1 2 3 4 5 6 7 8
      ρρZ
3
      Z,1 2
RANK ERROR
      $: Z,1 2
```

DYADIC COMMA: LAMINATE

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

$R\leftarrow A⌿[J]B$ (reverse indexed)

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

K is not integer $J\leftarrow(\phi\iota 1+\rho\rho B)[\lceil K\rceil-1\mid\mid K$

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

result shape: $N\leftarrow+/K>\iota\rho\rho B$

$(\rho R)\leftrightarrow(N\uparrow\rho B),2,N\downarrow\rho B$

definition: A result is created with a dimension of 2 inserted after the Nth dimension of B. A is placed in the first position on this new coordinate axis, and B is placed in the second position:

$R1\leftarrow(N\uparrow\rho B),1,N\downarrow\rho B$

$R\leftarrow(R1\rho A),[\lceil K\rceil R1\rho B$

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

If ⌿ is used, Reverse Indexing applies.

```
      1 2 3,[.5]4 5 6
1 2 3
4 5 6
      1 2 3,[1.5]4 5 6
1 4
2 5
3 6
      1 2 3,[2.5]4 5 6
INDEX ERROR
      1 2 3,[2.5]4 5 6
      ∧
      (2 3⍴6+⍳6),[.5]2 3⍴⍳6
 7  8  9
10 11 12

 1  2  3
 4  5  6
```

```
      (2 3⍴6+⍳6)
 7  8  9
 4  5 (6

10 11 12
 4  5  6
      )ORIGIN 0
1
      1 2 3,[¯.2]4 5 6
1 2 3
4 5 6
      1 2 3⌿[¯.4]4 5 6
1 4
2 5
3 6
```

TAKE

syntax: $R\leftarrow A\uparrow B$

conformability: $(\rho A)\leftrightarrow\rho\rho B$ This may not be circumvented by scalar extension of A 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 a scalar, A must be empty, and the result is 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 not 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

      S←⎕←(⍳0)↑7
7
      ⍴⍴S
0
```

DROP

syntax: `R←A↓B`

conformability: `(⍴A)=⍴⍴B` This may not be circumvented by scalar extension of A 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 a scalar, A must be empty, and the result is B

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 <u>not</u> 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
      Y←⎕←(⍳0)↓7
7
      ⍴⍴Y
0
```

COMPRESS

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

$R\leftarrow A\neq[J]B$

domain: A must be Boolean.

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

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

$(\rho A)\leftrightarrow(\rho B)[K]$

result shape: $(\rho R)[I]=(\rho B)[I]$ for $I\neq K$

$+/A$ 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 $\neq$ is used, Reverse Indexing applies.

(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 A)
      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
1
      1 0 1≠X
1  2  3  4
9 10 11 12
      1 0 1/[1]X
LENGTH ERROR                             ((⍴X)[1] is now 4, not 3)
     $: 1 0 1/[1]X
      1 1 1/'A'
AAA                                      (scalar extension of B)
      1 0 1≠[1]X
1  2  3  4
9 10 11 12
```

EXPAND

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

$R\leftarrow A\backslash[J]B$

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 is formed from B by extending the length of coordinate K to ρA and inserting subarrays of fill elements into the positions along coordinate K corresponding to the positions of zeros in A.

In addition, if $(\rho B)[K]=1$ and A contains more than one 1, replication of the existing subarray will occur in all succeeding positions along coordinate K corresponding to the positions of ones in A.

Since K is an index, the result, if K is specified, is ORIGIN dependent.

If $\backslash[J]$ is used, Reverse Indexing applies.

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
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)

```
      1 0 1 1\3 1⍴4 9 7
 4 0 4 4
 9 0 9 9
 7 0 7 7

      1 1⍀[2]2 1 3⍴⍳6
 1 2 3
 1 2 3

 4 5 6
 4 5 6
```

MONADIC ROTATE: REVERSAL

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

$R\leftarrow\ominus[J]B$

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 $\ominus[J]$ is used, Reverse Indexing is implied.

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

examples:

```
      ϕ1 2 3
3 2 1

      X←2 3ρ⍳6
      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
1
      ⊖X
4 5 6
1 2 3
      ϕ[1]X
3 2 1
6 5 4
      ⊖[0]X
3 2 1
6 5 4
```

DYADIC ROTATE

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

$R\leftarrow A\ominus[J]B$

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

$(\rho A)\leftrightarrow(K\neq\iota\rho\rho B)/\rho B$

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

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

$N\leftarrow(\rho B)\mid A$

$R\leftarrow(N\downarrow B),N\uparrow B$

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

$N\leftarrow-(\rho B)\mid\mid A$

$R\leftarrow(N\downarrow B),N\uparrow B$

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 $\ominus$ is used, Reverse Indexing applies.

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

examples:

```
      2φ1 2 3 4 5
3 4 5 1 2

      ¯2φ1 2 3 4 5
4 5 1 2 3

      5φ1 2 3 4 5
1 2 3 4 5

      ¯1φ'AND'
DAN

      X←3 4⍴⍳12
      X
1  2  3  4
5  6  7  8
9 10 11 12
      0 1 2φX
 1  2 3  4
 6  7 8  5
11 12 9 10
      1φX
 2  3  4 1
 6  7  8 5
10 11 12 9
      1φ[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
1
      1⊖X
5  6  7  8
9 10 11 12
1  2  3  4

      )ORIGIN 1
0
      1φ[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 3ρ¯2 ¯1 0 1 2 3
      P
¯2 ¯1 0
 1  2 3
      PφX
 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 $A\leftarrow\phi\iota\rho\rho B$)

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

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

(this may not be overridden by scalar extension of A or B.)

Case 1: A has no repeated elements.

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

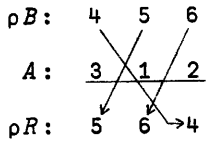
The transpose operation simply reorders the coordinate axis of the argument as indicated by the left argument.

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:



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: $(\rho R)[I]=\lfloor/(A=I)/\rho B$ for all $I \in \iota \rho \rho R$

$(\rho\rho R)=0\ 1[1]+\lceil/A$ (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.,

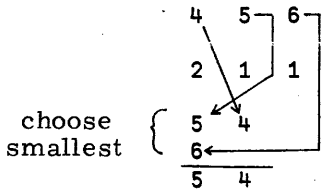
$$\lfloor/(\rho B)[(A=I)/\iota\rho A]$$

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

For example, consider the operation

$$R\leftarrow 2\ 1\ 1\diamond 4\ 5\ 6\rho\iota 120$$

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
```

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 ⍳⍴⍴B
```

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

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

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

```
      ⍉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
      )ORIGIN 0
1
      1 2 2⍉X
DOMAIN ERROR
     $: 1 2 2⍉X
      0 1 1⍉X
 1  6 11
13 18 23
      ⍉X
 1  5  9
 2  6 10
 3  7 11
 4  8 12

13 17 21
14 18 22
15 19 23
16 20 24
      (⍳0)⍉7
7
```

(A defaults to 3 2 1)

(A⍳A is 1 2 3)

(A⍳A is 0 1 1)

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*STAR, this range is $^{-}9.54E8644$ to $9.54E8644$ (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
      $: ÷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
      $: *1E8                    (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 π times the value of the argument. π 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
 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
      $: ~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
     1E6000×1E6000
DOMAIN ERROR
     $: 1E6000×1E6000          (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: $R \leftarrow A \mid B$

definition: $R \leftarrow B - A \times \lfloor B \div A + A = 0$

Note that this function does not use FUZZ. The FLOOR and EQUALS operations in the definition are performed with FUZZ=0. (See ABSOLUTE FUZZ).

examples:

```
      10|15.3
 5.3

      1|12.34 10 1.5
0.34 0 0.5

      3|¯4 ¯3 ¯2 ¯1 0 1 2 3 4
2 0 1 2 0 1 2 0 1

      0|¯4 ¯3 ¯2 ¯1 0 1 2 3 4
¯4 ¯3 ¯2 ¯1 0 1 2 3 4

      ¯3|¯4 ¯3 ¯2 ¯1 0 1 2 3 4
¯1 0 ¯2 ¯1 0 ¯2 ¯1 0 ¯2

      ¯14.3 ¯2.7 5.4 3.2|¯6.1 47 ¯3.8 11.6
¯6.1 ¯1.6 1.6 2
```

DYADIC POWER

syntax: *R←A*B*

domain: In APL*STAR, if A is negative, B must be integer.

If *A=0* then *B>0*

If *A<0* then B is integer.

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.

examples:

```
      2*3
8
      ¯10*¯1  0  1  2
¯0.1 1 ¯10 100
      0.01 2 4 9*0.5
0.1 1.4142136 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
      $: ¯1*0.5            (if A is negative, B must be integer)
      0*0
DOMAIN ERROR
      $: 0*0
```

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 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 MAXIMUM

syntax: *R←A⌈B*

definition: The result is the larger of A and 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 CIRCLE

syntax: *R←AOB*

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	(○0.5)≠(○1)|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	$\begin{cases} -\infty \text{ thru } -1 \\ 1 \text{ thru } \infty \end{cases}$	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	

Note: The domain of sin, cos and tan and the range of arcsin arcos and arctan are expressed in radian measure.

examples:

```
      2○3
¯0.9899924966
      1 2 3○○ 0.25 0.5 0.75
0.7071067812 0 ¯1
      ¯5○2.3
1.570278543
      3○○○.5
DOMAIN ERROR
      $: 3○○○.5
```

(cosine of 3 radians)

($\sin\frac{\pi}{4}$, $\cos\frac{\pi}{2}$, $\tan^{3}\frac{\pi}{4}$)

(inverse hyperbolic sine of B)

($\tan\frac{\pi}{2}$ is infinite)

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←A∧B        (and)
R←A∨B        (or)
R←A⍲B        (Nand)
R←A⍱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
      1∨0 1
1 1
      1⍲0 1
1 0
      1⍱0 1
0 0
```

COMBINATION

syntax: *R←A!B*

domain: If B is a negative integer, then A must be an integer.

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

R←(!B)÷(!A)×!B-A

For *A≥0* and *B≥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!5
10
      1!5
5
      0!5
1
      5!5
1
      7!5
0
      7!0
0

      2.5!7.3
32.617667
      ¯2.5!5
0.0036180411
      ¯2!5
0
      ¯4.5!6.5
0.00017356873

      5!¯2
¯6
      4.4!¯6.5
69.067588

      7.5!¯5
DOMAIN ERROR
      $: 7.5!¯5

(A must be integer
 if B negative integer)
```

```
      ¯2!¯5
0
      ¯7!¯3
15
      ¯4!¯4
1
      ¯3.5!¯2.5
¯2.5
      ¯3.5!¯5
DOMAIN ERROR
      $: ¯3.5!¯5

(A must be integer
 if B negative integer)


      A←4 ¯7.5 ¯4   ¯4.5  0
      B←5  5   ¯2.5 ¯2.5 ¯5
      A!B
5 ¯4.1785964E¯5 0 4.375 1
```

COMPOSITE FUNCTIONS 8

In addition to element-by-element application, three additional general procedures are defined for applying scalar dyadic functions to general array arguments.

They are: OUTER PRODUCT, REDUCTION and INNER PRODUCT.

A syntactic form exists which designates the desired procedure and the specific scalar dyadic function(s) to be employed in the procedure.

Each procedure can be considered a composite function in which the domain and (with some exceptions as noted) the range is that implied by the scalar dyadic functions designated.

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*STAR 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
      ⍴Z
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
```

REDUCTION

syntax:

$R\leftarrow f/[K]B$

$R\leftarrow f\not/[J]B$

where f is a scalar dyadic primitive function.

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

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

result shape: $(\rho R)\leftrightarrow(\sim(\iota\rho\rho B)\in K)/\rho B$

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:

$$R\leftarrow B[1]\ f\ B[2]\ f\ \ldots\ f\ B[\rho B]$$

If B is a one element vector the result is:

$R\leftarrow B[1]$ (a scalar)

If B is an empty vector and the function f has an identity element I, then

$R\leftarrow I$ (a scalar)

If B is an empty vector and the function f has no identity element, then a DOMAIN ERROR results.

For B of rank greater than 1, the operation is performed on vector sub arrays of B as indicated by the index K. Since K is an index, the result, if an index is specified, is ORIGIN dependent. If $f\not/$ is used, reverse indexing applies.

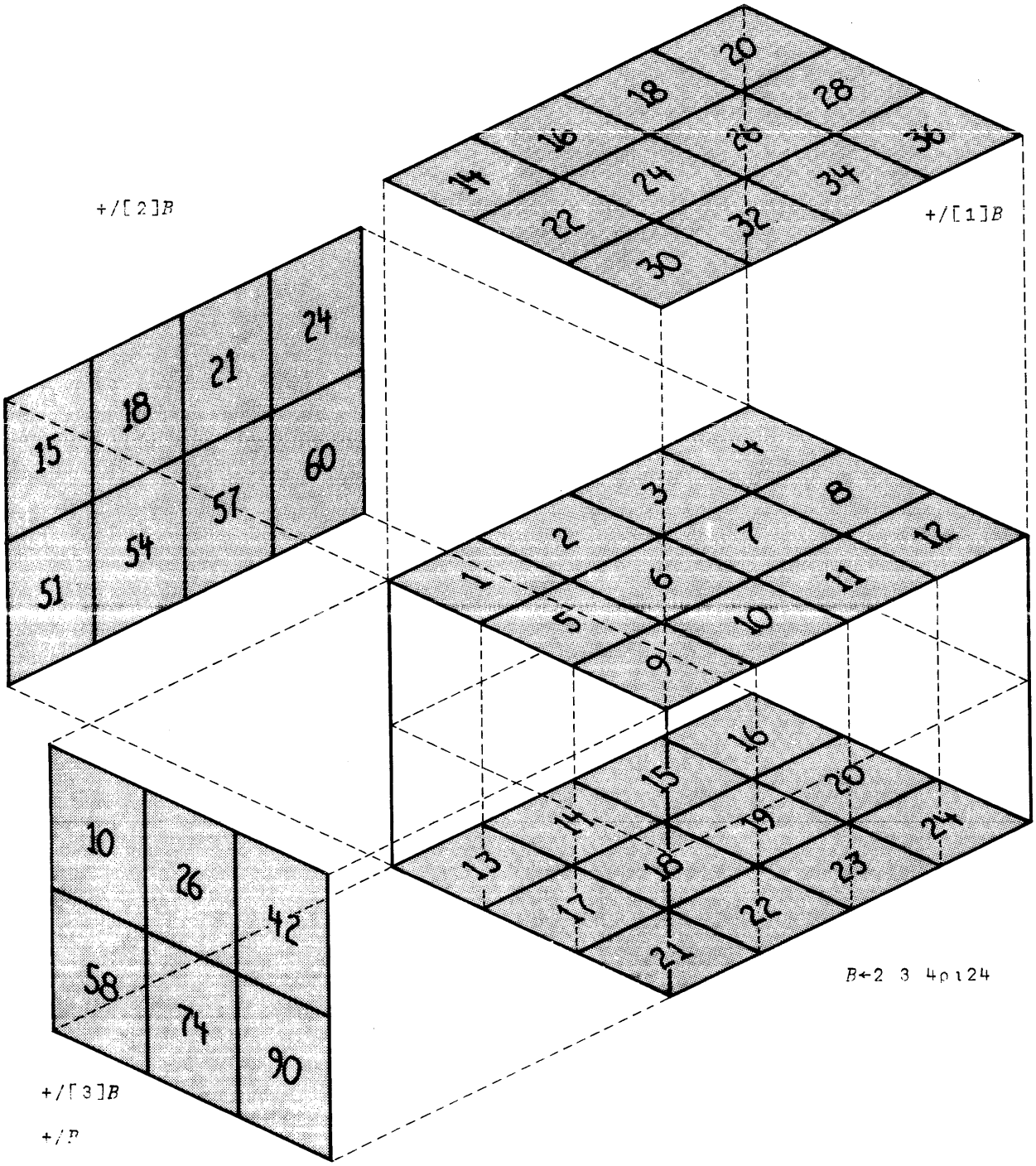
IDENTITY ELEMENTS

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
⌈	¯9.54E8644	smallest representable number
⌊	9.54E8644	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

examples:

```
      +/1 4 3
8

      -/1 4 3
0

      X←2 3ρ1 3 4 6 2 5
      X
1 3 4
6 2 5

      +/X
8 13

      +/[1]X
7 5 9

      )ORIGIN 0
1

      +/[1]X
8 13

      +⌿X
7 5 9

      +/3 0ρ1
0 0 0                          (addition identity elements)

      ρX
2 3
      ×/ρX
6

      ⍳/X
SYNTAX ERROR
     $: ⍳/X

      ⌊/'A'
DOMAIN ERROR
     $: ⌊/'A'

      ⍟⌿[2]4 7 0 3ρ9
DOMAIN ERROR
     $: ⍟⌿[2]4 7 0 3ρ9          ( ⍟ has no identity element)
```


+/[2]B
+/[1]B
B←2 3 4⍴⍳24
+/[3]B
+/B

INNER PRODUCT

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

where f and g are scalar dyadic primitive functions.

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

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

$((\rho\rho A)+\rho\rho B)\leq 129$ (APL*STAR restriction)

$(^{-}1\uparrow\rho A)=1\uparrow\rho B$ via extension if $1=^{-}1\uparrow\rho A$ or $1=1\uparrow\rho B$

result shape: $(\rho R)\leftarrow\rightarrow(^{-}1\downarrow\rho A),1\downarrow\rho B$

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

$R\leftarrow$ 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.).

If the length of the last coordinate of A or the first coordinate of B is one, scalar extension along that coordinate shall occur such as to make the two coordinate lengths the same.

Recall REDUCTION: Note that if A and/or B are empty but $0\neq\times/^{-}1\downarrow\rho A$ and $0\neq\times/1\downarrow\rho B$ the result will be non empty, consisting of function f identity elements as required for the resulting shape. If function f has no identity element a DOMAIN ERROR results.

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)
```

```
      (3 1ρι3)+.×3 1ρ10 15 20
 45
 90
135
      (1 3ρι3)+.×1 3ρ10 15 20
60 90 120
      (2 0ρι0)+.×0 3ρι0
0 0 0
0 0 0
      (2 0ρι0)=.>1 3ρ5
1 1 1
1 1 1
      (4 1ρ10)⍟.*0 2 3ρ7
DOMAIN ERROR
     $: (4 1ρ10)⍟.*0 2 3ρ7          ( ⍟ has no identity element)
```

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
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: $R\leftarrow A\iota B$

domain: A and B may be independently numeric or character.

range: Ordinal.

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

(Note: this requirement cannot be overridden by scalar extension.)

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

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 $(\rho A)+\iota 1$ (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:

  $R\leftrightarrow(\rho B)\rho(\rho A)+\iota 1$
- If A is empty, $R\leftarrow(\rho B)\rho\iota 1$
- For A and B both numeric, element comparisons are subject to the setting of FUZZ (see FUZZ).
- If $\wedge/,B\epsilon A$ then $B\leftrightarrow A[R]$

(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 3⍳⍳0
0                     (recall (⍴R)←→⍴B )
```

DYADIC EPSILON: MEMBERSHIP

syntax: $R\leftarrow A\in B$

domain: A and B may be independently numeric or character.

range: Boolean.

conformability: None.

result shape: $(\rho R)\leftarrow\rightarrow\rho A$

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

      9∈⍳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: $R\leftarrow\Delta B$

domain: B must be numeric.

range: Ordinal.

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

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:

```
      Δ4 7.3 ¯3.7 1 5.27 8.1E7
3 4 1 5 2 6
      □←R←ΔB←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: $R\leftarrow\Psi B$

domain: B must be numeric

range: Ordinal.

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

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
```

REPRESENTATION

syntax: $R\leftarrow A\top B$

domain: A and B must both be numeric.

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

definition: If A has zero or one element the result is $A\circ.|B$.
If A is a multi-element vector and B is a scalar, the result is a vector, the elements of which form the representation of B in a scheme with radices specified by A.

if: $S\leftarrow{}^{-}1\uparrow A$
then: $(A\top B)\leftrightarrow(({}^{-}1\uparrow A)\top(0\neq S)\times(B-S|B)\div S+S=0),S|B$

Note that if the above A has zero-value elements, all elements of the result whose ordinal is less than the highest zero-value element ordinal of A are zero.

With general arguments, the result is obtained by using each vector subarray of A along the first coordinate as a radix scheme in forming the representation of each element of B as a corresponding vector along the first coordinate of the result. The correspondence is as per OUTER PRODUCT.

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
      0  12⊤113
9 5                                    (quotient and remainder of 113÷12)
      0  1⊤12.34
12 0.34                                (integral and fractional part of 12.34)
      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)
      2 0 ¯2⊤¯13
0 ¯7 1
      2 0 ¯2⊤¯13
0 ¯6 ¯1
      ρ□←(,0)⊤1 2 3
1 2 3
1 3


      10 10 10 10⊤6473 2196 857 42
6 2 0 0
4 1 8 0
7 9 5 4
3 6 7 2

      (3 2ρ10 1)⊤823 457 91 147
8 4 0 1
0 0 0 0

2 5 9 4
0 0 0 0

3 7 1 7
0 0 0 0
```

BASE VALUE

syntax: $R\leftarrow A\perp B$

domain: A and B must both be numeric

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

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

$(^{-}1\uparrow\rho A)=1\uparrow\rho B$ | via extension if $1=^{-}1\uparrow\rho A$ or $1=1\uparrow\rho B$

result shape: $(\rho R)\leftrightarrow(^{-}1\downarrow\rho A),1\downarrow\rho B$

definition: If A and B are both multi-element vectors, the result is a scalar whose value is that represented by B in a radix scheme A.

With general arguments, each vector subarray of A along the last coordinate is used as a radix scheme to evaluate each number whose representation is a vector along the first coordinate of B, the value being placed in the corresponding position of the result. The correspondence is as per INNER PRODUCT.

Recalling INNER PRODUCT, note that if A and/or B are empty but $0\neq\times/^{-}1\downarrow\rho A$ and $0\neq\times/1\downarrow\rho B$ the result will be non empty, consisting of addition identity elements (zeros) as required for the resulting shape.

If the length of the last coordinate of A or the first coordinate of B is one, scalar extension along that coordinate shall occur such as to make the two coordinate lengths the same.

If the length of both of the above coordinates is one, the result can be expressed as $R\leftarrow((\rho A)\rho 1)+.\times B$

```
      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                        (argument lengths diff-
     $: 2 2⊥1 0 1                    erent)
      0 3 12⊥3 2 3.25
135.25                              (inches in 3 yards, 2 feet,
      (3 1⍴2)⊥17                     3 1/4 inches)
17 17 17
      2 2 2⊥1 3⍴17
119 119 119
      (1 3⍴2)⊥3 1⍴17
119
      ⍴(1 3⍴2)⊥3 1⍴17
1 1
      A←(0 3⍴2)⊥3 1⍴17
      ⍴A
0 1
      A←(1 3⍴2)⊥3 0⍴17
      ⍴A
1 0
      A←(0 3⍴2)⊥3 0⍴17
      ⍴A
0 0
      A←(3 0⍴2)⊥0 2⍴17
      ⍴A
3 2
      A
0 0
0 0
0 0
      (1 2⍴2)⊥1 3⍴17
51 51 51                            (1↑⍴B extended to 2)
      (2 1⍴2)⊥3 2⍴17
119 119                             (¯1↑⍴A extended to 3)
119 119
```

EVALUATE

syntax:	$R \leftarrow \blacktriangle B$
domain:	Character.
conformability:	$(\rho\rho R) \leftrightarrow 1$ $(\rho B) \leq 65535$ (APL*STAR restriction)[1]
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
      $: ⍎'2÷0'
       ⍎'→5'
SYNTAX ERROR                    (not evaluable)
      $: ⍎'→5'
       ⍎')DIGITS 5'
VALUE ERROR
      )$: DIGITS 5
      $: ⍎')DIGITS 5'
```

numeric test

```
      NUM←'0=0\0ρ'
      B←ι5
      ⍎NUM,'B'
1
      C←'ABC'
      ⍎NUM,'C'
0
      ⍎NUM,'ι0'
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),''ι0'' '
      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),ι0
```

This expression is then evaluated, yielding the final result.

IMBED

syntax: *R←⊂B*

domain: No restriction

conformability: None

range: List

result shape: *(⍴R)←→,1*

definition: The result is a one-element vector list whose element is the imbedded array B .

examples:

```
      L←⊂'ABC'
      L
NONCE ERROR                          (list cannot be displayed)
      ⍴L
1
      L[1]
ABC                                  (imbedded list element revealed)
      E←0↑L
      ⍴E                             (E is an empty list vector)
0
      F←1↑E                          (F is list fill element)
      ⍴F
1
      L3←(⊂'ABCD'),(⊂2 3⍴⍳6),⊂L
      L3[1]
ABCD
      L3[2]
1 2 3
4 5 6
      L3[3][1]
ABC
      L22←L3,⊂L3
      L22[4][2]
1 2 3
4 5 6
      L22[4][1]
ABCD
      L22[4][3][1]
ABC
```

FORMAT

syntax: $R\leftarrow\blacktriangledown B$

domain: Numeric or character

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

If B is numeric -

result shape:

$$(\rho\rho R)=\rho\rho B$$
$$(^{-}1\downarrow\rho R)\leftrightarrow{}^{-}1\downarrow\rho B$$
$$(^{-}1\uparrow\rho R)=\begin{cases}0 & \text{if } 0={}^{-}1\uparrow\rho B\\ {}^{-}1+{+}/W+1 & \text{otherwise}\end{cases}$$

where $W[I]$ is the width required to format the Ith column of B.

definition: If $(\rho\rho B)\geq 3$, B is treated as a restructured array B1 as described under DISPLAYING DATA. The columns of B1 are then formatted according to the rules given in that same section. This results in $(^{-}1\uparrow\rho B)$ character arrays FB_j of column width W_j, and shape:

$$(\rho FB)\leftrightarrow(^{-}1\downarrow\rho B),W$$

The result is then formed by catenating the formatted columns, with a blank separator column between each:

$$R\leftarrow FB_1,' ',FB_2,' ',\ \ldots\ ,' ',FB_W$$

where: $W=\rho B$

note: Since the formatting is done according to the rules given under DISPLAYING DATA, the result is dependent on the setting of DIGITS. However, since the result is not actually displayed, it is <u>not</u> sensitive to the setting of WIDTH, which is a terminal display parameter.

application: The purpose of FORMAT is to convert numeric data to character data which can then be suitably 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.

If B is character -

definition: $R\leftarrow B$

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
      ⍪2 3⍴'ABCDEF'
ABC
DEF
```

NULL

syntax: *R←A∘B*

domain: No restriction

conformability: None

definition: *R←A*

In addition, if B is a non-result-returning function, no error occurs, as the result of B is never referenced.

This function is primarily used for placing on one line expressions which logically (but not syntactically) constitute a single task.

examples:

```
      5∘3
5
                                  (note only 5 displayed)
      X←5∘Y←3
      X;Y
53
      Z←1↑Y∘2 F 9                 (F need not be result returning)
      Z                           Right argument of TAKE is
3                                 effectively just Y.
```

MONADIC I-BEAM

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

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

conformability: $(\rho\rho B)=0$ $(\rho\rho B)=1$ $(\rho B)\leq 65535$

result shape: $(\rho\rho R)=0$ (see below)

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.

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

B	Information
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.).
22	Current amount of workspace unused (bytes).
24	Sign-on time (seconds since midnight).
25	Today's date ($YYMMDD_{10}$).
26	Value of current function line number.
27	Vector of line numbers from State Indicator.

2. If B is character, the action taken is to perform the System Command indicated by B. The System Command SAVE and DROP may not be used. The result returned is the same as that which would have been reported by the command in question. If an error occurs in executing the command, this error is reflected to the I-beam function.

examples:

```
      24 60 60⊥⍳20
13 23 47                    (Time is now 1:23:47 p.m.)
      ⍳')DIGITS'
8
      ORG←⍳')ORIGIN 0'
      )ORIGIN
0
      ORG
1
```

DYADIC I-BEAM

syntax: `R←A⌶B`

domain: A must be integer (see table below)

conformability: `(⍴⍴A)=0`

definition: The dyadic I-beams are not normally used by a non-system user. They perform the system functions indicated below:

A	Meaning
6	Workspace/session parameter interface
8	Special character interface

6 I-BEAM

domain: B must be numeric

`B[1]` must be integer

conformability: `(⍴⍴B)=1` `(⍴B)∈1 2`

range: R is numeric

result shape: `(⍴⍴R)=0`

definition: `B[1]` indicates the workspace or session parameter being accessed. The result is the current value of the parameter in question. If `(⍴B)=2` , the parameter is subsequently set to the value of `B[2]`

B[1]	Parameter	Domain of B[2]
0	Index Origin	Boolean
1	Random Number Seed	Integer `1≤B[2]<2*47`
2	Digits	Integer `1≤B[2]≤13`
3	Print Width	Integer `1≤B[2]≤65535`
4	Comparison Fuzz	Real `0≤B[2]≤1`

8 I-BEAM

domain: B must be integer

conformability: `(⍴⍴B)=0`

range: Result is character

result shape: `(⍴⍴R)=0`

definition: The result is the special character indicated by B according to the table below:

B	Character
0	New Line

examples:

```
      V←'ABCD',(8⍳0),'WXYZ'
      V
ABCD
WXYZ
      ⍴V
9
```

(Note V is a vector of 9 characters)

MATRIX INVERSE, MATRIX DIVISION

dyadic syntax: `R←A⌹B`

monadic syntax: `R←⌹B` (here the left argument defaults to an identity matrix with shape `2↑⍴B`)

domain: A and B are numeric.
B must be non-singular (see Solving Linear Equations)

conformability:

`2≥⍴⍴A`

`2≥⍴⍴B`

`(1↑⍴A)←→1↑⍴B` (after scalar extension if A and/or B scalar)

if `2=⍴⍴B` then `(⍴B)[1]≥(⍴B)[2]`
(necessary but not sufficient to ensure the non-singularity of B)

if `1=⍴⍴B` then `0≠⍴B`

result shape: `(⍴R)←→(1↓⍴B),(1↓⍴A)`

definition: The result R is such that each element (if any) of `+⌿(A-B+.×R)*2` is minimized.

MATRIX INVERSE $R\leftarrow\boxminus B$

The monadic process $\boxminus B$ results in a matrix which is the left inverse of the matrix B. The result R has a shape and value such that the matrix product with B is an identity matrix I:

$$I\leftarrow R+.\times B$$

Note that this requires the shape of R to be identical to the transpose of B:

$$(\rho R)\leftrightarrow\rho\boxminus B$$

An identity matrix is one which when multiplied by any other conformable matrix results in that matrix:

$$C\leftarrow I+.\times C \qquad (\rho C)[1]\leq(\rho C)[2]$$

$$C\leftarrow C+.\times I \qquad (\rho C)[1]\geq(\rho C)[2]$$

An identity matrix is a square matrix with ones along its diagonal, and all other elements zero. It is its own inverse.

Matrix inverse is best thought of as a process identical to the dyadic case of matrix division in which the left argument is an identity matrix of conformable shape.

The resulting product of a matrix inverse $\boxminus B$ and a matrix A is identical to the matrix division $A\boxminus B$ when B is non-singular:

$$(A\boxminus B)\leftrightarrow(\boxminus B)+.\times A$$

example:

```
      □←R←⊟B←3 3⍴1 0 3 0 1 3 3 0 1
¯0.125 0  0.375
¯1.125 1  0.375
 0.375 0 ¯0.125
      R+.×B
1 0 0
0 1 0
0 0 1

      ⊟3 4
0.12 0.16
      ⊟2
0.5
      ⍴⊟5 0⍴⍳0
0 5
```

LINEAR EQUATIONS

A linear equation is one in which variable terms occur only to the first power; i.e., having the form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$$

If only two variables are present, the equation becomes:

$$a_1x_1 + a_2x_2 = c$$

and represents a straight line. That is why equations of this form are called linear equations.

It is customary to regard the last stated variable as the dependent variable and the rest as independent variables. The dependent variable can then be regarded as the value of a function of the independent variables such that the linear relationship expressed by the equation holds:

$$x_2 = f(x_1) = \frac{-a_1}{a_2} x_1 + \frac{c}{a_2}$$

$\frac{-a_1}{a_2}$ is the slope of the line.

$\frac{c}{a_2}$ is the intercept on the x_2 (function) axis.

A linear equation with three variables:

$$a_1x_1 + a_2x_2 + a_3x_3 = c$$

is the general equation of a plane. It expresses the linear relationship of a function of two independent variables:

$$x_3 = f(x_1, x_2) = \frac{-a_1}{a_3} x_1 \frac{-a_2}{a_3} x_2 + \frac{c}{a_3}$$

In general, a linear equation involving N variables is the general equation of a hyperplane of N-1 dimensions. It expresses the linear relationship of a function of N-1 independent variables.

A set of linear equations is thus a set of lines, planes or hyperplanes depending on the number of variables present. All equations can be raised to the same dimension by supplying coefficients of zero as required.

The solution of a set of linear equations is the location of a point common to all given lines, planes, etc. To determine a solution, there must be as many equations in the set as there are variables: the common point of two lines is their point of intersection, the common point of three planes is their mutual point of intersection.

Even when a sufficient number of equations are present, a solution may not be possible. No common point exists for instance for two lines that are parallel. Likewise, no common point exists for three planes, each pair of which intersects along lines that are mutually parallel.

In such cases, it is possible to show that one or more of the equations can be derived (with a possible difference in the constant term) from some algebraic combination of one or more of the remaining equations. Such an equation is said to exhibit a linear dependence with respect to the remaining equations. For a solution to exist in the case where the number of equations is equal to the number of variables present, all such equations must be mutually, linearly independent.

SOLVING LINEAR EQUATIONS

Matrix division provides a systematic way of solving a set of linear equations. Each row of the matrix B constitutes the coefficients of the variables for an equation. To solve a single set of equations, A is a vector constituting the set of corresponding constant terms for the set of equations, and the result R is a solution vector for the variables in the equation set.

To determine a solution for a set of linear equations, there must be as many equations as variables. Further, each equation must express a relationship independent of (i.e., not capable of being derived from, yet consistent with) any or all other equations in the set. This criterion is determined solely by the shape and value of B. If the criterion is not met, B is said to be singular and a LENGTH ERROR or DOMAIN ERROR results.

Solve:

$$2x + 3y + z = 23$$
$$x + 4y + 2z = 27$$
$$3x + y - 2z = 10$$

```
      23 27 10⌹3 3⍴2 3 1 1 4 2 3 1 ¯2
3 5 2
```

(The solution is x = 3, y = 5, z = 2)

(The solution

Solve:

$$5x + 2y + 3z = 23$$
$$x + y + 2z = 5$$
$$4x + y + z = 18$$

```
      23 5 18⌹3 3⍴5 2 3 1 1 2 4 1 1
DOMAIN ERROR
      ⌹: 23 5 18⌹3 3⍴5 2 3 1 1 2 4 1 1
```

(The third equation can be derived from the first two by subtraction and is thus not independent. Note further that if the constant term of the third equation were other than 18 a contradiction would result, thus illustrating that the non-singularity of B is the only criterion required to ensure a solution exists.)

Multiple sets of linear equations in which only the constant terms differ in each set can be simultaneously solved in one matrix division operation. Each row of the matrix B constitutes the coefficients of the variables for an equation as before. A is now a matrix, each column of which is a set of constant terms for one equation set. The result is a matrix, each column of which is a solution vector for the variables for the equation set whose constant terms appear in the corresponding column of A.

Solve:

$$2x + 3y + z = 23 \quad (=23)$$
$$x + 4y + 2z = 27 \quad (=32)$$
$$3x + y - 2z = 10 \quad (=-4)$$

```
      B←3 3⍴2 3 1 1 4 2 3 1 ¯2
      A←3 2⍴23 23 27 32 10 ¯4
      A⌹B
3 2
5 4
2 7
```

(The solution for the first set is x = 3, y = 5, z = 2
The solution for the second set is x = 2, y = 4, z = 7)

LINEAR PARAMETRIC EQUATIONS

Any equation that is linear with respect to its parameters, or which can be made so by a suitable transformation, can be used to form a linear parametric equation by substituting the coordinates of a point known to satisfy the equation. If as many independent points are known as there are parameters, then a set of parametric equations, each linear with respect to the parameters, can be solved to yield a set of parameter values which, when substituted into the original equation, results in an equation satisfying all the given points.

The general parametric equation for a line is:

$$y = ax + b$$

The parametric equation of a line passing through the point (2, 5) is thus:

$$5 = 2a + b$$

Solving a set of two such equations determines the parameters (slope and intercept) of a line passing through both given points.

Find the line passing through (2, 13) and (4, 19):

$$13 = 2a + b$$
$$19 = 4a + b$$

```
      13 19⊟2 2ρ2 1 4 1
3 7
```

The required line is $y = 3x + 7$

Find the parabola symetrical about the positive x axis which passes through (2, 2) and (12, 7):

The general parametric equation is $y^2 = ax + b$

$$4 = 2a + b$$
$$49 = 12a + b$$

```
       4 49⊟2 2ρ2 1 12 1
4.5 ¯5
```

The required parabola is $y^2 = 4.5x-5$

Find the plane passing through (2, 3, 23), (.5, 2, 16), and (¯1.5, ¯.5, 2)

(2 3 23), (.5 2 16), and (¯1.5 ¯.5 2)

parametic equation for a plane: $z = ax + by + c$

$$23 = 2a + 3b + c$$
$$16 = .5a + 2b + c$$
$$2 = {}^{-}1.5a + {}^{-}.5b + c$$

```
      23 16 2⌹3 3⍴2 3 1 .5Z2 1 ¯1.5 ¯.5 1
2 4 7
```

The required plane is $z = 2x + 4y + 7$

Note that 2 points are required to determine a line, 3 points to determine a plane, etc.

Supplying the required number of points does not guarantee a solution since, for example, two coincident points do not determine a line nor do 3 colinear points determine a plane. Also, if the required line, plane, etc. is parallel to the independent variable axis, the parameters are indeterminant. Such cases can be shown to be the result of nonmutual independence of all the linear equations in the set.

LEAST SQUARES FIT

If more points are supplied than required, then the corresponding set of parametric linear equations is said to be overdetermined and the solution obtained by matrix division is called a least squares fit of the given points. That is, the solution is a set of parameters for a parametric equation such that the sum of the squares of all projections of the points along the independent variable axis to the curve, surface, etc. is a minimum.

Obtain the equation of the straight line which is the least squares fit to the following points.

(1, 2), (2, 2.4), (4, 5.1), (5, 7.3), (6, 9.4), (8, 18.3).

parametric equation: $y = ax + b$

```
      X←1 2 4 5 6 8
      Y←2 2.4 5.1 7.3 9.4 18.3
      Y⌹(6 1⍴X),1
2.21 ¯2.16
```

The required line is $y = 2.21x - 2.16$

Obtain the equation of an exponential curve which is the least squares fit to the same points

exponential curve equation $y = ab^x$

take logarithms to obtain a linear parametric equation

$$\ln y = \ln a + x \ln b$$

```
      *(⊛Y)⌹1,⍟6 1⍴X
1.370744829 1.383563134
```

The required exponential curve is approximately

$$y = (1.37)(1.38)^x$$

By taking the sum of the squares of the projections of the points to the curve for each case,

for the straight line:

```
      +/(Y-¯2.16+2.21×X)*2
19.465
```

for the exponential curve:

```
      +/(Y-1.37×1.38*X)*2
0.3517337408
```

The exponential curve can be seen to provide a better fit to the given points.

SPECIAL CASES

When B is a vector, scalar or empty matrix and a conformable A argument is supplied, a result is obtained as follows:

examples:

```
      1 2 3 4 5⌸1 1 1 1 1
3
```
(mean of elements of vector left argument)

```
      1⌸5
0.2
```
(as per division)

```
      ⍴3⌸1 1⍴6
1
```
(one element vector result)

```
      ⍴1 2 3 4 5⌸5 0⍴⍳0
0
```
(empty vector result)

```
      ⍴(⍳0)⌸0 0⍴⍳0
0
```
(empty vector result)

```
      ⍴(0 2⍴⍳0)⌸0 0⍴⍳0
0 2
```
(empty matrix result)

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.
- 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
• SYNTAX ERROR	improper number of arguments supplied.
• VALUE ERROR	variable not established (could be misspelled).
• RANK ERROR	argument rank conformability requirement not met.
• LENGTH ERROR	other conformability requirement not met.
• DOMAIN ERROR	supplied argument not in the domain of definition, or result not in the range of definition of the function.
• 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]
SYNTAX ERROR
     Y[0.5+0 1×X+$: ↑Y]
      Y[0.5+0 1×X+1↑Y]
VALUE ERROR
     Y[0.5+0 1×$: X+1↑Y]
      X←2 3ρ1 2 3 4 3 2
      Y[0.5+0 1×X+1↑Y]
RANK ERROR
     Y[0.5+$: 0 1×X+1↑Y]
      Y[0.5+(3 2ρ0 1)×X+1↑Y]
LENGTH ERROR
     Y[0.5+$: (3 2ρ0 1)×X+1↑Y]
      Y[0.5+(2 3ρ0 1)×X+1↑Y]
DOMAIN ERROR
     $: Y[0.5+(2 3ρ0 1)×X+1↑Y]
      Y[⌊0.5+(2 3ρ0 1)×X+1↑Y]
INDEX ERROR
     $: Y[0.5+(2 3ρ0 1)×X+1↑Y]
      )ORIGIN 0
1
      Y[⌊0.5+(2 3ρ0 1)×X+1↑Y]
1 2 1
9 1 2
```

(Continued on next page.)

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
```

ADDITIONAL ERRORS

Message	Cause
• NONCE ERROR	operation not yet implemented.
• WS FULL	workspace storage capacity exceeded (see SIZE command).
• DEPTH ERROR	a function is pendent or suspended more than 16,383 times.
• SYMBOL TABLE FULL	more than 65,535 symbols have been used in this workspace.
• REF ERROR	an object or list array element has more than 16,383 active references.

NOTE: The maximum array size currently permitted is 65,535 elements. A NONCE ERROR is issued if an attempt is made to create an array larger than this size.

ERROR RECOVERY

Whenever an error is detected, the system attempts to recover to the state it was in before the line which caused the error was executed. The following rules are used in recovering from errors:

- An error in a line submitted for immediate execution causes execution of the line to be aborted, and a message indicating the error type to be issued, along with the offending line and an error marker at the point at which execution in the line was aborted. The user is then requested to submit another line for immediate execution.

- An error in a line submitted in response to a QUAD prompt (see QUAD INPUT) behaves as above, and the QUAD prompt is re-issued.

- An error in a line of a user-defined function behaves as above, except that in addition to aborting execution of the line, the function is suspended on the line in question, and the function name and line number is issued preceding the display of the line.

- An error in a line executed via the EVALUATE function behaves as above, except that the EVALUATE function is not suspended. Control is returned to the line which contained the call to EVALUATE, the line is displayed with an error marker at the offending evaluate, and suspension and display is attempted according to the above rules.

- An error in a line of a locked user defined function causes execution of the line to be aborted, and control is returned to the line which contained the call to the locked function. If this line is not itself part of a locked function, the message <function name> ERROR is issued, along with the line and an error marker at offending locked function call, and suspension is attempted according to the above rules. If the calling line is part of a locked function, control is restored to the first line which is not part of a locked function, and suspension is attempted as above.

- An error in a System Command executed via I-beam behaves as an error in an EVALUATE line.

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, (omitting the error marker), 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.

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 signalling ATTENTION.)

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.

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.

If the existing active workspace is saved while awaiting input, such as by)SAVE, the workspace is saved with the input request status intact. In this case, when the saved workspace is subsequently loaded, the prompt □: will appear and again the system awaits input.

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)
0 1                        (display from ⍞← )
1 2 3                      (result)
      1+⍞                  (immediate execution input)
⍞:                         (prompt issued)
      →                    (exit from last execution)
      )SI
(blank)                    (nothing pendant or suspended)
(blank)                    (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)
      ⍴⎕←⎕                 (immediate execution input)
⎕:                         (QUAD prompt issued)
      'X',⍞,'Y'            (response to QUAD)
ABC                        (response to QUAD-PRIME)
XABCY                      (display from ⎕← )
5                          (result)
      ⍴⍞                   (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)
0                          (exit from last execution)
                           (system again awaits input for immediate
                                                        execution)
```

QUAD -PRIME PROMPT

Normally, no prompt other than a bell or keyboard unlock occurs when the system requests input from the user. However, the user program may specify a prompt to be issued with the input request. This is done by specifying the desired prompt to ⍞ as follows:

syntax: $R\leftarrow⍞\leftarrow B$

The underlined portion of the above line is the specification proper.

domain: B must be character.

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

definition: As for specification, the result is B. The character vector B is issued as a prompt at the next request for ⍞ input.

NOTE that visual fidelity requires that if the user's input is entered immediately following the prompt, on the same line, the prompt becomes an integral part of the returned input.

examples:

```
      ⍞←'ANSWER YES OR NO:'
      ANS←⍞
ANSWER YES OR NO:YES                    (type YES)
      ANS
ANSWER YES OR NO:YES
      R←⍞
⍬                                       (note no prompt)
```

VISUAL FIDELITY

The underlying 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.
- Signal ATTENTION.

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

CORRECTING AN INPUT LINE PRIOR TO SUBMISSION

1. Position via any combination using the backspace key and/or space bar to the left-most character to be erased.
2. Signal ATTENTION.
3. Key in appended text (if any).
4. Submit the corrected line for execution.

INPUT SUBMISSION PROCEDURE

All input is submitted in the form of a line. Normally the line consists of the line entered at a terminal or the card image presented to the card reader. Submission is achieved by keying RETURN (usually) at a terminal, or on acceptance of a card by the card reader.

CONTINUATION CHARACTER

It is possible to submit a line in parts, each part consisting of a line entered and submitted as above. All line parts, except the last, must contain the continuation character $CO as the right-most non-space character. The continuation character in this position serves to indicate that the line as submitted is incomplete and is to be continued on the next submitted line part. The last part of a submitted line must not contain a terminating continuation character since its absence conveys that submission of the entire line is complete.

The APL system constructs a contiguous input line by catenating left to right all consecutively submitted line parts. The continuation characters, when placed as indicated above, are not included in the constructed input line. Any other placement of the continuation character will cause the character to be subsequently transformed to the canonical bad character and not cause action as stated above.

When a line submitted from a terminal contains a terminating continuation character, a prompt for a continuation line is issued in the form: $CO at the left of a line. This is followed on the next line by indentation six spaces from the left margin. At this point, the system awaits input of a continuation line to be submitted.

examples:

```
      3↑'ABCDEF$CO                    (first part)
$CO                                   (continue line prompt)
      GHIJK'                          (last part)
ABC                                   (result)
      +'ABCDEF$CO                     (first part)
$CO                                   (continue line prompt)
      GHIJK'                          (last part)
DOMAIN ERROR
   $: +'ABCDEFGHIJK'                  (note catenation of input)
    3↑$CO'ABCDEFG'
SYNTAX ERROR
                                      ($CO transformed to $BC if not
   3↑$: $BC'ABCDEFG'                  right-most non-space character)
```

COMMENTS

Any executable line of APL may be appended on the right with a comment. The special symbol ⍝ (verbalized 'lamp') delimits the executable portion from the comment.

An executable line is any of the following:

(1) a line submitted for immediate execution

(2) a user-defined function body line (See User-Defined Functions)

(3) a QUAD input line

(4) an EVALUATE argument

examples:

```
      M←(+/A)÷(⍴,A)⍴1 ⍝ MEAN VALUE OF A'S ELEMENTS

      [3] NEXT:→0×⍳0=⍴A←1↓,A ⍝ EXIT IF A EMPTY

      25 36 49*⎕
⎕:
      0.5 ⍝ SQUARE ROOT
5 6 7

      ⍝ THIS ENTIRE LINE IS A COMMENT

      ∊B/'J←⍴⍴K ⍝ IF B TRUE'
```

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 or consist solely of 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: At least one non-blank character must be present.

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

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: $\rightarrow B$

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 $I\leftarrow 1\uparrow B$. I must be integer.
2. If $I\in 0\ 1[1]+\iota N$, 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 I (if it exists) is always the first line of the function, and $\rightarrow 0$ 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:

$\rightarrow$

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 calling function, or be the right argument of NULL, 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*STAR 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↑ρC)⌈1↑ρA          |  [1] C←(ρA),ρB
[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≤ρρA),ρA)ρA
[6] [6]
[6] →0⌈ι(ρρA)≠ρρB
[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,ρ⎕←'DOMAIN ERROR'
[2.4] [2.1]
[2.1] →L1×ι(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<⍴⍴A),⍴A)⍴A
[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]
```

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 Contiguous Lines of a Function Definition

directive: [N□M]

action: All existing lines from N through M are displayed followed by a prompt for line 1+⌊L, where L is the last line. If M < N the error report EDIT ERROR is issued followed by reissuing of the previous prompt.

example: [2□5]

Display lines 2 through 5.

directive: [□M]

action: All existing lines from M to L are displayed followed by a prompt for line 1+⌊L.
If M > L no lines are displayed.

example: [□4]

Display all lines from 4 to last.

directive: `[□]`

action: The entire function definition is displayed, followed by a prompt for line `1+⌊L`.

example: `[□]`

directive: `[N□]`

action: Line N is displayed if it exists, followed by a prompt for line N.

example: `[0□]`

Display the header line and issue a prompt for line 0.

(B) Displaying Lines Containing a Specified String

Each of the directives discussed in (A) above can be qualified by suffixing a text string enclosed by a delimiter character. Any character not appearing in the text string other than ▽ and ▽ may be used as the delimiter character.

Lines displayed are restricted to the range defined for each case in (A), but within this range only those lines containing the specified string are displayed.

For the purpose of search, all lines are regarded as ending in ten spaces. This facilitates locating lines with a specific last character.

The ensuing prompt is as per the corresponding directive in (A) above.

examples: `[3□7]/B←/`

Display all lines from 3 through 7 in which the text string B ← occurs.

`[□3]/B /`

Display all lines from 3 to the last in which the last text character is B, or which contain the enclosed string as part of a character literal or comment.

`[□]/⍝/`

Display all lines of the function definition which have a comment.

`[7□]/:/`

Display line 7 if it exists and contains the character:

EDITING DIRECTIVES

For each of the display directives (B) above there exists a corresponding edit directive formed by suffixing a second text string using the same delimiter character.

```
[N□M]/< string1 >/<  string2 >/
```

For the range of function lines implied by the expression in brackets, all occurrences of string 1 in all such lines are replaced by string 2. The two strings may be of independent length including zero.

If string 1 is empty, an entire line is matched.

```
[3□7]/A[1]/A[1;1]/
```

Replace all occurrences of `A[1]` with `A[1;1]` from line 3 to 7 inclusively.

```
[□]/AX/AY/
```

Replace all occurrences of AX with AY in the entire function definition.

```
[3□]//R←⍳0/
```

Replace line 3 with R ← 20

```
[□6]///
```

Delete all lines from 6 to last inclusively.

EDITING ACTIVE FUNCTIONS

Any editing of a pendant or suspended function which would change the number or position of parameters, locals or labels results in either a DEFN ERROR with the change ignored, or SI DAMAGE when the function is closed.

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.

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×⍳∨/⍵[⍺]≠0 1 1⌈99 12 31[⍵;⍺]
```

(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.

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 possible input combinations for invoking and using the function editor are listed below. Note that the character # may be used in place of □ in function editor directives.

To invoke the editor (new function):	▽ < function header > { ▽ }
To invoke the editor (existing function):	▽ < function name > { < line entry > / < display directive > / < edit directive > }
To enter (or replace) a line:	{ [< line number >] } < line text >
To display line N:	[N□] { ▽ }
To display all lines from N to last:	[□ N] { ▽ }
To display the entire function definition:	[□] { ▽ }
To display all lines from N to M containing string:	[N□M] / < string > /
To edit a line:	[N□] / < string1 > / < string2 > / { ▽ }
To edit all lines from N to M:	[N□M] / < string1 > / < string2 > / { ▽ }
To delete a line:	[N□] /// { ▽ }
To delete all lines from N to M:	[N□ M] /// { ▽ }
To exit from the function 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).

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.

Although upper case letters are used to form the names of the commands, lower case letters are accepted as equivalent to upper case in the command name.

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.

If the first N letters of the command name are required to uniquely distinguish the command, the first N or more letters of the name may be used in lieu of the complete name.

CATEGORIES OF SYSTEM COMMANDS

- Listing the active workspace objects.
- Defining and listing functions.
- Erasing global objects.
- Debugging aids.
- Determining and altering workspace environment parameters.
- Altering workspace size.
- Saved workspace 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*STAR, the size of a workspace is user specifiable. For APL*STAR, the maximum size of an active workspace is installation dependent, but is in the order of 256 pages. Any attempt to exceed the current capacity of the workspace results in the error report WS FULL.

The active workspace has provision for an identification (ID) in the same format as saved 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 1E¯10
  6. DIGITS 8
  7. SEED 48131768981101
  8. input mode: awaiting input for immediate execution
  9. SIZE 52,736 (bytes).
- successful completion of the command is indicated by the report CLEAR WS.

example:

```
      )CLEAR
CLEAR WS
```

ACTIVE WORKSPACE INVENTORY

A system command listing global object names exists for each kind of workspace object.

VARS COMMAND

syntax:)VARS [<NAME 1> [<NAME 2>]]

action: lists the names of global variables currently defined in the active workspace in alphabetic order; in the range NAME 1 to NAME 2 inclusive. If NAME 2 is omitted, all variables following NAME 1 (inclusive) are listed. If no parameters are given, all variables are listed.

note: Alphabetic sequence is as follows:

0-9

A-Z

_

a-z

Δ

∫

note: NAME 1 and/or NAME 2 need not be defined, referenceable, or of the same object type as those names being listed.

examples:

```
TΔb←T_b←b←Ac←rΔ←A3←∫2←AJ←Δ5←8

)VARS
A3  AJ  Ac  T_b  TΔb  b  rΔ  Δ5  ∫2

)VARS TΔb
TΔb  b  rΔ  Δ5  ∫2

)VARS Ab  r
Ac  T_b  TΔb  b  rΔ
```

FNS COMMAND

syntax:)FNS [<NAME 1 [<NAME 2 >]]

action: lists the names of user-defined functions currently existing in the active workspace in alphabetic order (see)VARS).

OBS COMMAND

syntax:)OBS [<NAME 1 >[<NAME 2 >]]

action: Lists the names of all types of global objects currently defined in the active workspace in alphabetic order, (see)VARS).

LVARS COMMAND

syntax:)LVARS [<NAME 1 >[<NAME 2 >]]

action: Lists the names of active workspace variables currently having a value, in alphabetic order (see)VARS).

GRPS COMMAND

syntax:)GRPS [<NAME1 > [<NAME2 >]]

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

example:

```
      X←Y←Z←T←'DATA'
      )GROUP G1 X Y G2
      )GROUP G2 Z T
      )SAVE WS
74/09/03 13:47:06
      )CLEAR
CLEAR WS
      )COPY WS G1
SAVED 74/09/03 13:47:06
```

```
)GRPS
G1      G2
)VARS
X       Y
)GRP G2
Z       T
```

(note group definition G2 is copied as it is a referent of G1, but Z, T, the referents of G2 are not copied)

ALTERING A GROUP DEFINITION

(a) Any general change in a group definition.
Merely issue a new)GROUP command using the same group name with an appropriate identifier list reflecting the new grouping desired.

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

(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.

DISPLAYING A GROUP DEFINITION

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

GRP Command

syntax:)GRP <group name > { < NAME1 > { <NAME2 > } }

action: the identifier list of the group definition is displayed in alphabetic order (see)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.

example:

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

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.

ENVIRONMENTAL PARAMETERS

In all environmental parameter commands, if no parameter value is supplied, the parameter is left unchanged. The previous value is always reported.

ORIGIN COMMAND

syntax:)ORIGIN [0 1]

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

note:) CLEAR sets ORIGIN to 1.

example:

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

DIGITS COMMAND

syntax:)DIGITS [<integer>] 1 ≤Integer≤13

action:
- DIGITS is set to the value specified.
- The previous 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 8.

example:

```
      )DIGITS
8
      )DIGITS 12
8
      )DIGITS
12
```

SEED COMMAND

syntax:)SEED [<integer>] 1 < integer < 2*47

action:
- SEED is set to the integer specified
- The previous value of SEED is reported.

examples:

```
      )SEED
4.8131769E13
      )SEED 129653
4.8131769E13
      )SEED
129653
```

note:)CLEAR sets SEED to 48131768981101

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.

FUZZ COMMAND

syntax:)FUZZ [N] $0 \leq N < 1$

action: FUZZ is set to the value supplied.

The previous value of FUZZ is reported.

note:)CLEAR sets FUZZ to $1E^{-}10$

ALTERING WORKSPACE SIZE

Although the workspace at all times has a fixed capacity for storage, the user is able to alter this size at any time that he feels the current size is inappropriate. He may reduce it to exactly match the amount he has in use, or expand it to the maximum size for which he is authorized.

SIZE COMMAND

syntax:)SIZE [N]

where N is the number of bytes of capacity desired

action: The workspace size is set to $16\times\lceil N\div 16$ bytes.

The previous size is reported.

error reports: WS FULL - N is less than the amount of storage currently in use.

DOMAIN ERROR

N is greater than the maximum amount for which this user is authorized.

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 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, it is dispersed.

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

SI DAMAGE

Active user-defined function was erased.

example:

```
      )VARS
V1      V2      V3
      )FNS
F1      F2      F3
      )ERASE V1 F2
      )VARS
V2
      )FNS
F1
```

DEFINING AND LISTING FUNCTIONS

In addition to using the function editor, functions can be created using the DEFINE command and can subsequently be displayed via the DISPLAY command.

DEFINE COMMAND

syntax:)DEFINE <VARIABLE NAME>

Let the variable name be B. Then the following constraints must be satisfied:

domain: B must be list.

All elements of B must be imbedded character vectors.

conformability:

$(\rho\rho B)=1$

$(\rho B)=N+1$

$N\geq 0,\ N\leq 65534$

} where N is the number of body lines in the function represented

$(\rho\rho B[I])=1$

$(\rho B[I])\leq 65535$

} for all $I\in\iota\rho B$

definition: The first element of B must represent a legal function header, starting with a '∇'. Subsequent elements of B must represent legal function body lines.

If B represents a legal function, the function is defined as a global object, replacing any existing function of the same name.

If the function is suspended or pendant, definition proceeds only if the local environments of all suspended and pendant calls are still valid under the new definition. An exception to this rule concerns label values. If an existing labeled line occupies a different position in the new definition, definition still proceeds even though the value of the label is not updated in the local environments of suspended or pendant calls.

error reports: DEFN ERROR

a) The named variable did not represent a legal function.

b) The function represented by the named variable has a different local environment from that of a suspended or pendant function of the same name.

INCORRECT COMMAND

B is not the name of a defined variable.

DISPLAY COMMAND

syntax:)DISPLAY <FUNCTION NAME> [<VARIABLE NAME>]

Let the name of the variable specified be R. Then the following rules apply to the result R.

range: R is a list
All elements of R are imbedded character vectors.

result shape: $(\rho\rho R)=1$

$(\rho R)=N+1$ where N is the number of body lines in the function represented

$(\rho\rho R[I])=1$ for all $I\epsilon\iota\rho R$

definition: The first element of R is a character representation of the function header in canonical form, starting with a '∇'. Subsequent elements of R are a character representation of the body lines of the function in canonical form. Lines with labels are preceded by one blank. Lines without labels are preceded by two blanks. Comments are preceded by four blanks.

If no variable name is given, the display of the function is reported.

error reports: DEFN ERROR

The function name is not an unlocked function

INCORRECT COMMAND

The variable name is neither undefined nor the name of a currently defined variable.

example:

```
      LIST1←(⊂'∇R←A FUN3 B'),(⊂'R←A+2×B'),⊂'R←R⌈B*2'
      )DEFINE LIST1
      )DISPLAY FUN3
∇R←A FUN3 B
  R←A+2×B
  R←R⌈B*2
      )VARS
      LIST1
      )FNS
      FUN3
```

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:

```
      LIST1←(⊂'∇R←A FUN1 B,Z'),⊂'R←A+FUN2 B'
      )DEFINE LIST1
      )DISPLAY FUN1
∇R←A FUN1 B;Z
  R←A+FUN2 B
      LIST2←(⊂'∇R←FUN2 C'),⊂'+  ⍝THIS LINE IS WRONG'
      )DEFINE LIST2
      )DISPLAY FUN2
∇R←FUN2 C
  +⍝THIS LINE IS WRONG
      2 FUN1 3
SYNTAX ERROR
FUN2[1] $: +  ⍝THIS LINE IS WRONG
      )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 IS WRONG
      )SI
FUN2[1] *          }
FUN1[1]            }     Second suspended calling sequence.
FUN2[1] *          }
FUN1[1]            }     First suspended calling sequence.
      →
      )SI
FUN2[1] *          }
FUN1[1]            }     First suspended calling sequence (environment
                         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
FUN2[1]  $: +  ⍝THIS LINE IS WRONG
      )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 line 0 appears in the stop list, suspension occurs immediately prior to exit of the function. The environment of the function is still in effect at this point. Execution is resumed by →0.
- Issuing a new STOP command for the same function causes the new stop list to supersede the old one.
- 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: INCORRECT COMMAND

Named object is not an unlocked function.

examples:

```
      LIST1←(⊂'∇R←A FUN3 B'),(⊂'R←A+2×B'),⊂'R←R⌈B*2'
      )DEFINE LIST1
      )DISPLAY FUN3
∇R←A FUN3 B
  R←A+2×B
  R←R⌈B*2

      )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
```

SAVED WORKSPACES

Each APL user is provided with facilities for preserving his user environment (the active workspace) at any point in a session as a saved 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. Each stored workspace has a workspace identification (ID) by which it can be referenced. Facilities exist for updating or deleting individual workspaces and for incorporating specified objects or groups from saved workspaces 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> := <workspace name> [:<password>]

Every workspace has an identification (ID) consisting of:

- a workspace name.
- an optional password.

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.

In this system, the name can be up to 7 characters long and cannot include the characters Δ, _ and ∫ . The password is formed according to the same rules as apply to an identifier.

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 password is omitted, it defaults to no password.

No default is allowed for COPY, PCOPY or DROP.

Reports

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

- The password (if any) is never reported.
- If there is no workspace ID, it is reported as CLEAR WS.
- The timestamp (if any) is reported as:

  ∅∅YY/MM/DD∅HH:MM:SS∅

- If the workspace has no timestamp, a blank timestamp field is reported.

If a workspace ID was not supplied with the command, the active workspace ID is reported following the timestamp.

SAVE COMMAND

syntax:)SAVE [<workspace>]

action:

- A saved workspace identical to the currently active workspace is created.
- The active and saved workspaces are designated with the workspace ID supplied and with a current timestamp.
- A defaulted password causes the password on the active workspace to be used.
- Any previous saved workspace bearing the ID of the newly saved one is dropped, if this is allowed.
- Upon succesful completion of the command; the following is reported:

  < TIMESTAMP> [<WSID>]

error reports:

- WS LOAD ONLY

  An attempt was made to save this 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 or a currently existing library workspace while the active workspace ID was different. (This protects one from inadvertently overwriting a saved 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 WS

)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
74/03/09 15:37:04 XIS3               and set WSID to same)
      )SAVE                         (resave under same name)
74/03/09 15:37:29
      )SAVE XIS3:Y                  (resave with password)
74/03/09 15:37:43
      )CLEAR
CLEAR WS
      Y←1+X←3
      )SAVE
NOT SAVED - THIS WS IS CLEAR WS     (cannot save as CLEAR WS)
      )SAVE XIS3
NOT SAVED - THIS WS IS CLEAR WS     (XIS3 already exists)
      )WSID XIS3                    (declare WS 'XIS3')
CLEAR WS
      )SAVE
74/03/09 15:38:23
```

LOAD COMMAND

syntax:)LOAD {<workspace ID>}

action:

- 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.
- 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:

  SAVED <TIMESTAMP> {WSID>}

error reports:

- WS NOT FOUND {<WSID>}

  no workspace by that name.
- WS LOCKED

  password does not match.
- WS NOT LOCKED

  password given for unlocked workspace.

STATE OF SAVED WORKSPACES

A workspace is always saved in the state which exists at the time of the save. When the workspace is subsequently loaded, it is loaded in that state.

examples:

```
      X←3
      )SAVE XIS3
  74/03/09 15:44:59
      )CLEAR
CLEAR WS
      )VARS

      )LOAD XIS3
SAVED  74/03/09 15:44:59
      )VARS
      X
      X
3
      X←2
      )LOAD
SAVED  74/03/09 15:44:59 XIS3
      X
3
      )LOAD XIS3:PQR
WS NOT LOCKED
      )SAVE XIS3:ABC
    74/03/09 15:47:04
      )LOAD XIS4
WS NOT FOUND
      )LOAD XIS3
WS LOCKED
      )LOAD XIS3:PQR
WS LOCKED
```

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 no object list is provided, all global objects in the referenced workspace are copied.
- Successful completion of the command results in the report:

    SAVED < TIMESTAMP > { < WSID > }

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: WS NOT FOUND WS LOCKED, WS NOT LOCKED
as for LOAD

OBJECTS NOT FOUND
< identifier list >

the objects reported in < identifier list > could not be found in the referenced workspace.

SI DAMAGE

an active user-defined function was erased.

examples:

```
      X←3
      Y←7
      )SAVE XIS3
74/09/03 15:43:27
      X←2
      Y←5
      )COPY XIS3                    (copy all global objects)
SAVED 74/09/03 15:43:27
      X,Y
3 7                                 (X, Y restored)

      X←2
      Y←5
      )COPY XIS3 Y                  (copy Y only)
SAVED 74/09/03 15:43:27
      X,Y
2 7                                 (Y only restored)

      )GROUP GRP1 X Y A             (create group GRP1)
      )SAVE
74/09/03 15:44:03  XIS3
      )CLEAR
CLEAR WS
      )COPY XIS3 GRP1               (copy GRP1)
SAVED 74/09/03 15:44:03
      )VARS
      X       Y                     (copied as existing referent of GRP1)
      )GRPS
      GRP1
      )GRP GRP1
      A       X       Y             (A is a referent but does not exist)

      )COPY XIS3 A
OBJECT NOT FOUND
A
SAVED 74/09/03 15:44:03
```

PCOPY COMMAND

syntax:)PCOPY < workspace ID > {< object list >}

action: Action is identical to COPY except 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: WS NOT FOUND - as for LOAD

OBJECTS NOT FOUND - as for COPY

< identifier list >

note that objects which would have been prevented from being copied if found, are nonetheless reported if not found.

OBJECTS NOT COPIED

< identifier list >

the objects in < identifier list > had the same names as existing global objects in the active workspace.

examples:

```
      X←3
      Y←4
      )SAVE XIS3
74/09/03 15:51:17
      X←2
      )ERASE Y
      )PCOPY XIS3 X Y Z          (protect copy all global
                                  objects in XIS3)
OBJECTS NOT FOUND
Z                                 (Z not in saved XIS3)
OBJECTS NOT COPIED                (X is not copied since
X                                  it exists in active WS)
SAVED 74/09/03 15:51:17
      )VARS
      X       Y
      X,Y                         (Y is copied,
                                  X is preserved)
2 4
```

DROP COMMAND

syntax:)DROP < workspace ID >

action:

- A search is made for a workspace with the specified name, as for)LOAD.
- If found, the workspace is removed, if this is allowed.
- The date and time when dropped are displayed to indicate successful execution of this command.

error reports: as for)LOAD

- WS LOAD ONLY

  An attempt was made to drop this workspace.

```
      )SAVE XIS3                          (create it)
74/03/09 16:01:14                         (time stamp)
      )LOAD                               (load it)
SAVED  74/03/09 16:01:14 XIS3             (time saved)
      )DROP XIS3:PQR                      (drop it)
WS NOT LOCKED
      )DROP XIS3
73/03/09 16:01:37                         (time dropped)
      )LOAD XIS3
WS NOT FOUND                              (XIS3 no longer exists)
      )SAVE XIS3:ABC
74/03/09 16:01:58
      )DROP XIS3
WS LOCKED
      )DROP XIS3:ABC
74/03/09 16:02:11
```

WSID COMMAND

syntax:)WSID {<workspace ID>}

action: (a) No parameters provided. The active workspace timestamp and ID are reported.

note: an empty ID is reported as CLEAR WS; this does not necessarily mean a CLEAR workspace.

(b) If a workspace ID is provided, it becomes the ID of the active workspace. The active workspace timestamp and previous ID are reported.

example:

```
)WSID
                    CLEAR WS          (note no timestamp)
X←3
)WSID
                    CLEAR WS          (but not CLEAR !)
)SAVE XIS3
74/03/09 16:12:42                     (XIS3 saved)
)WSID
74/03/09 16:12:42 XIS3
X←4
)WSID XIS4
74/03/09 16:12:42 XIS3
)SAVE XIS3
NOT SAVED - THIS WS IS XIS4           (protects previous XIS3)
)WSID XIS3
74/03/09 16:12:42 XIS4
)WSID                                 (now XIS3 can be updated)
74/03/09 16:12:42 XIS3
)SAVE
74/03/09 16:13:23 XIS3                (update XIS3)
)WSID
74/03/09 16:13:23 XIS3                (new timestamp)
```

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 are not contained in saved workspaces.

WIDTH COMMAND

)WIDTH [<integer>] 30 ≤integer ≤65535

action:

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

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 D for default values for specific terminals.

LINES COMMAND

)LINES [<integer>] 0 ≤integer ≤¯1+2*47

action:

- LINES is set to the value supplied.
- The previous 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 'MORE??' at the left margin. Any input other than '$A.' 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.

note: The remainder of the display is aborted by signalling ATTENTION.

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.

See appendix D for default values.

EXTERNAL FILE INTERFACE

At the moment, only two commands are available to interface to files external to APL. One command performs input, and the other performs output.

INPUT COMMAND

syntax:)INPUT < FILENAME>

action: All input is subsequently taken from the named file. The file is assumed to be a standard ASCII file. When the end of the file is reached, input is again accepted from the terminal.

OUTPUT COMMAND

syntax:)OUTPUT <FILENAME>

action: A log of the entire session is placed on the named file in standard ASCII format, in a form suitable for printing.

Each OUTPUT command purges the previous contents of the named file.

The log will be formatted according to the WIDTH and LINES values for the output device and cannot be altered. Responses to)WIDTH and)LINES in the log will however be those values in force at the time issued.

TERMINATING AN APL SESSION

SYSTEM COMMAND

syntax:)SYSTEM

action: The session is terminated, the active workspace is destroyed, and the user is returned to the system from which APL was called.

ACCESS TO APL*STAR ON STAR OS A

1. Establish user identity by a LOGON line.
2. Initiate execution of APL*STAR by entering the name of the file containing the APL interpreter. The system responds with:

```
APL*STAR                V1.1
```

At this point the user is in direct communication with the APL*STAR 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*STAR 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.

For the convenience of users on terminals in which lower case is the normal alphabetic mode, lower case letters are accepted as equivalent to upper case in the mnemonic letter pair.

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	≤	$LE	Less than or Equal
*	--	asterisk	=	--	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(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)
	$:	(error marker)
	$A.	(output escape)
	$CO	(continuation character)

NUMERIC REPRESENTATION ON STAR COMPUTERS C

- An exact representation for zero exists.
- The sum of any selection from any 47 consecutive terms of the power series of 2
  2*(¯28672 ... ¯1 0 1 ... 28717)
  in which at least one term is greater than or equal to 2*¯28626 can be represented exactly.
- The negation of any such number except 2*¯28626 can be represented exactly. In addition the negation of 2*28718 can be represented exactly.
- Any number outside this range cannot be represented and is not in the domain of definition or result range of any numeric APL function.
- Any number within the range
  (-((2*¯28626)+2*¯28672)),(2*¯28626)
  is approximated by the representation for zero by all numeric APL functions.
- 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.

TERMINAL CAPABILITIES OF APL*STAR D

DISPLAY-EDIT STATION

- WIDTH 63
- LINES 15
- Local line editing facilities:

  backspace

  forward space

  clear line

  destructive overstrike

  deletion of everything above and to right of cursor.

- Quad prime input does not cause input to be formed by the catenation of the previous quad prime output and the input typed by the user. (Device does not support output followed by input on the same line.)
- Signalling ATTENTION

  input - not needed (see local line editing facilities).

  output/execution - send (F1) 0 5 I

  Note: The actual control character used is installation dependent.

  in response to MORE?? message - send $A.

CARD READER FILE (accessed through INPUT command)

- Quad prime input does not cause input to be formed by the catenation of the previous quad prime output and the input typed by the user. (Device is not output device.)
- Local line editing facilities:

  APL does not support use of ESC control character to replace three or more contiguous blank characters.
- No overstrike capability.
- Signalling ATTENTION not possible.

PRINTER FILE (accessed through OUTPUT command)

- WIDTH 135
- LINES 0
- No overstrike capability.

INDEX

COMMENT SHEET

MANUAL TITLE CONTROL DATA® APL*STAR Reference Manual

PUBLICATION NO. 19980800 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

PRINTED IN U.S.A.

419 REV. 11/69

NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

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
Documentation Department
215 Moffett Park Drive
Sunnyvale, California 94086

CUT ALONG LINE

FOLD

FOLD

► ►CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

CONTROL DATA
CORPORATION

CORPORATE HEADQUARTERS, 8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

Pub. No. 19980800

Litho in U.S.A.