A DICTIONARY OF APL[2]

Kenneth E. Iverson

ACKNOWLEDGMENTS: *I am indebted to numerous colleagues for minor corrections and major ideas arising from their reviews of various drafts of the mss., chief among them being R. Pesch, E.E. McDonnell, R. Hui, D.L. Forkes, J.A. Cramer, K.B. Iverson, P. Cooper, R.H. Chamberlain, L.J. Dickey, and J. Sansom.*

APL is a formal, imperative language. Because it is imperative, a sentence may be called an *instruction*, and may be *executed* to produce a *result*. In illustrations used here, an instruction will be indented, and the result of its execution will be shown on the following line without indentation. Thus:

```
      3+4
7
      2×(3+4)
14
```

Because it is formal and unambiguous, APL can be executed mechanically by a machine called a *computer* or, strictly speaking, an *APL computer*, or *APL system*. For this reason it is also called a *programming language*, a phrase from which its name derives. Because it shares the analytic properties of mathematical notation, it is also called an analytical programming language.

Like any living language, APL has dialects, resulting in part from evolution that has rendered certain constructs obsolete, but also from the limitations and special characteristics of the many different computers on which APL systems are implemented. Although this introduction includes a brief discussion of dialects, the reader may find it necessary to consult manuals for specific systems, or the APL Standard promulgated by the International Standards Organization [1].

APL originated in an attempt to provide consistent notation for the teaching and analysis of topics related to the application of computers, and developed through its application in a variety of topics, and through its implementation in computer systems. Extensive discussions of the history of the language may be found in *The Design of APL* [2] and *The Evolution of APL* [3], and in the publications cited therein.

A dictionary should not be read as an introduction to a language, but should rather be consulted in conjunction with other material that uses the language in some context of interest to the reader. Even the general section on grammar, which may be intelligible even to the beginner, should perhaps be studied only after a certain amount of other exposure to the language.

On the other hand, a dictionary should not be used only to find the meanings of individual words, but should also be studied to gain an overall view of the language. In particular, the grammar may be profitably reviewed again and again in the light of increased knowledge of the language, and the study of groups of related verbs and adverbs can reveal important relationships otherwise easily overlooked.

I: LETTERS AND WORDS

In addition to both cases of the letters of some *native* alphabet such as English (used for names) and the ten decimal digits (used in numbers and in names), the APL alphabet includes six Greek letters, fifty-four graphic symbols, and about twenty composite symbols formed by superposing a pair of graphic symbols. Nearly all of the graphic symbols occur (in at least one orientation) in some table of commonly-used symbols in English dictionaries, such as the Table of Symbols and Signs in *The American Heritage Dictionary* [4]. At least half are familiar from their use in English and in arithmetic.

On most APL systems, the name ⎕av denotes a list whose elements include the alphabet. Many systems print the two English alphabets in other than lowercase and uppercase italics, often using uppercase and uppercase underscored.

Every "primitive" verb and adverb (i.e., those defined in the dictionary) is formed from a single graphic, composite symbol, or Greek letter. Consequently, symbols do not conflict with the choice of names for pronouns, as do the *reserved words* (names of primitives represented by letters of the English alphabet) commonly used in programming languages.

In any use of symbols it is convenient to have pronounceable names for them. Because of the many different uses commonly made of the same symbol in different endeavours, the best-known names usually reflect a particular use of the symbol rather than a name for the symbol itself. For example, in the cited table from [4], "resistivity", and "multiplied by" are given for ⍴ and ×, rather than "Greek rho" and "St. Andrew's cross".

This paper was originally published at APL Quote Quad, Vol. 18:1.

In this dictionary, we attempt to provide both a name for the symbol (in Table 1: Alphabet) and for its general use in APL (Sections IV-VI). Table 4 provides synonyms for possible use in particular contexts. The three columns of Table 1 provide the APL symbol, a name for the symbol and a transliteration scheme for use with the ASCII character set [5] widely provided on computer terminal devices. The last twenty-five APL symbols (beginning with ⍀) are called composite symbols because they are commonly entered by superposing two simpler symbols (as in ∩ with ∘ for ⍀), and are often named by the pair composing them (as in circle-bar for ⊖).

A student of any language should be alert to, and exploit, clues provided by knowledge of other languages, as well as clues provided by the appearance of the symbols themselves. For example:

- The appearance of the symbols ⌊ and ⌈ suggests their use for *minimum* and *maximum*, and the similarity of ⌊ to the letter L suggests "lesser of"; the pair ↑ and ↓ are used for *take* and *drop*.
- Greek-English correspondences are also significant, as in ⍳ for integers (⍳4), and in ⍴ for repetition (3⍴4) or reshape (2 3 ⍴⍳6).
- ⊥ is used for *base-value* (as in 10⊥1 8 6 7 ↔ 1867) because its shape suggests a base, and ⊤ is used for the inverse function.
- ∨ is used for *or* because v is the initial letter of the corresponding Latin *vel*, and ^ is used for *and* because of its graphic relationship to ∨.

Words are formed by scanning a sequence of letters of the APL alphabet from left to right according to the following rules:

1. Quotes are treated much as in English: everything between an opening quote and a cIosing quote forms a single word. However, a pair of adjacent quotes following an opening quote represents the quote character itself, and does not act as a closing quote; *'can''t'* is the five-character word commonly used as an abbreviation of the six-letter word *'cannot'*. The remaining rules are used *after* applying the rule for quotes.
2. A *delimiter* terminates (and is not included in) any word that precedes it.
3. 3. Every letter that is neither a macron (¯), dot (.), quad (⍞), space ( ), digit, nor letter of the native alphabet, is a one-letter word, and also serves as a delimiter.
4. The space is a delimiter unless it falls between two *numbers*, that is, words that begin with a digit or a macron, or with a dot followed by a digit.

Word-formation is defined formally by the function ⊥ in Section IV.

Any of the words ⍺, ⍵, →, and $, or a word that begins with the symbol ⍞ or a letter of the native alphabet and continues with further letters, digits, macrons or dots is a *name*, and may be assigned a value by means of a copula (←) as discussed in the section on grammar.

A word that is not a name is called a *token*. A token is either meaningless or has a fixed referent. For example, the tokens 3.14 and 314e¯2 and 2j3 are meaningful, but 3..14 and ¯¯2 and 'pi and 2k3 are not.

A meaningful numeric token may be a list formed of two or more numbers separated by spaces, where a number may be a *real number* or a *complex number* formed by two real numbers separated by a *j*.

A *real number* is either *simple* or an *exponential* number formed from two simple numbers separated by an *e*. A simple number may be formed of one or two macrons (representing infinity and minus infinity), or contain at least one digit and at most one dot and one macron (which denotes a negative number and must be in the leading position).

II: G R A M M A R

APL has six grammatical elements:

1. **Nouns**, such as the *numbers* 3 and 5.6, *alphabetic characters* or *literals* such as '+' and *'A'* and *'a'*.
2. **Verbs**, such as + (addition) and | (magnitude) that act upon nouns to produce results that are also nouns.

   The nouns to which a particular verb applies are called its *arguments* (a word adopted in its sense of *theme*, or *subject*). A verb may have two distinct (but usually related) meanings according to whether it is applied to one argument (to its right) or two arguments (to its left and right). For example:

```
      2÷5                    ÷4
.4                    .25
```

3. **Adverbs**, such as /, that act upon verbs or nouns. For example, +/ is a (*derived*) verb called *plus across* that sums any list of numbers to which it is applied, and ×/ is a verb that yields the product across all elements of a list.

   When an adverb applies to a noun rather than to a verb (as in 1 0 2 / to produce a 'selection' verb so that 1 0 2 / 5 8 7 yields 5 7 7) it might well be called an *adjective*. However, since such production of verbs from nouns is (in English) more commonly performed by a suffix rather than by an adjective (as in deify, reify, and nitrify), we will use the term *adverb* for all cases.
4. **Conjunctions**, such as ⍤ and ¨, which apply to two arguments, as in ⌽⍤⍉ (reverse *and* transpose). The arguments of a conjunction may be either verbs (as in the examples above) or nouns (as in 10¨⍟, meaning base-ten logarithm).

5. A **copula**, denoted by the arrow ←, and used to assign a name to a noun, verb, adverb, or conjunction. For example:

```
      area←4×8
      3+area
35
      sum←+/
      sum 2 3 4
9
```

   The arrow plays the same role as the copulas "is" and "are" in English; it is usually read as "is", as in "area is four times eight" for `area←4×8`, and "(the verb) sum is plus across" for `sum←+/`.

   A name assigned to a noun will be called a *pronoun*, and one assigned to a verb may be called a *proverb* (pronounced with a long o as in "pronoun" to distinguish it from the existing English word).

6. **Punctuation**, provided by paired parentheses that specify the sequence of execution just as they do in elementary algebra. Sentences must appear on separate lines, and no punctuation is used to separate them.

A. NOUNS

Nouns may be classified in four independent ways: numeric or literal; token or pronoun; open or boxed; arrays of various ranks. Arrays of ranks `0`, `1`, and `2` have the special names, *item*, *list*, and *table*, or, alternatively (in mathematics) *scalar*, *vector*, and *matrix*. The four classifications are elaborated below.

Numeric and literal. Numbers are written as `2` and `2.3` and `23e2` (for `2300`) and `2j5` (for a complex number), and a negative number has a leading macron, as in `¯3.2`; literals are enclosed in quotes, as `'A'` and `'a'` and `'+'`.

Tokens and pronouns. A name that has been assigned to a noun will be called a *pronoun*. Tokens that refer to nouns include `¯` and `¯¯` and `∘` (denoting infinity, minus infinity, and the boxed empty list `<''`), as well as other numeric items and lists (such as `3.4e2` and `2 3 4`), and literal items and lists (such as `'a'` and `'cab'`) . A phrase that produces a noun (such as `3×4` or `⌽'abcd'`) may be called a *noun phrase*.

Arrays. A single entity such as `2.3` or `2.3j5` or `'A'` or `'+'` is called an *item*. The verb denoted by a comma (and called *catenate* or *before*) chains its arguments to form a *list* whose *shape* (given by the verb `⍴`) is equal to the number of items combined. For example:

```
      date←1,7,7,6
      ⍴date
4
      word←'s','a','w'
J
      ⍴word
3
```

   `⌽word` (`⌽` is the verb *reverse*)

```
was
```

   `⌽date`

```
6 7 7 1
```

The expression `s⍴l` produces an array of shape `s` from the list `l`. For example:

```
      (3,4)⍴date,1,8,6,7,1,9,1,7
1 7 7 6
1 8 6 7
1 9 1 7
      table←2 3⍴word,'b','a','t'
      ⍴table          table
2 3             saw
                bat
```

The number of elements in the shape of a noun is called the *rank* of the noun; thus a noun of rank `2` is a table, of rank `1` is a list, of rank `0` is an item, of rank `3` is a `3`-*dimensional* or *rank*-`3` array, etc. Moreover, each position of the shape is called an *axis* of the array, and axes are referred to by indices `0`, `1`, `2`, etc. For example, axis 0 of table has length `2` and axis `1` has length `3`.

The last `k` axes of an array `a` determine *rank-k cells* or *k-cells* of `a`. For example, if:

```
      a←2 3 4⍴'abcdefghijklmnopqrstuvwx'
      a
abcd
efgh
ijkl

mnop
qrst
uvwx
```

then the list `'abcd'` is a `1`-cell of `a`, the two separate `3` by `4` tables are `2`-cells of `a`, and the individual letters are each `0`-cells of `a`.

The rest of the shape vector is called the *outer shape* or *frame* of the array relative to the cells of rank `k`. For example, if `⍴b` is `2 3 4 5`, then `b` has the frame `2 3` relative to the cells of rank `2` (and therefore of shape `4 5`), has a frame of `2 3 4` relative to `5`-element rank-`1` cells, a frame of `2 3 4 5` relative to cells of rank `0` (that is, items), and an empty frame (signifying an item) of cells of rank `4`.

The number of cells in a frame is the product over its shape, and if one or more of the elements in the shape is zero, the number of cells is zero; the frame is then said to be a *zero frame*. Since the product across an empty list is `1`, an empty frame is *not* a zero frame.

A cell of rank one less than the rank of `a` is called a *major* cell of `a`, and major cells play an important role in the discussion of nouns and the application of verbs to

them. For example, the verb *from* (denoted by {) selects major cells from its argument, as in:

```
      0{a
 abcd
 efgh
 ijkl

      1{a
mnop
qrst
uvwx

      0{0{a
 abcd

      2 1{0{a
ijkl
efgh

      1{2{0{a
 j
```

Moreover, the verb *grade* (denoted by ⍋) provides indices to { that bring major cells to "lexical" or "row-major" order. For example:

```
      n←4 3⍴3 1 4 2 7 9 3 2 0 3 1 4
      n
3 1 4
2 7 9
3 2 0
3 1 4

      g←⍋n
      g
1 0 3 2

      g{n
2 7 9
3 1 4
3 1 4
3 2 0

      'abcdefg' ⍋ 'cafe'
1 0 3 2
```

Negative numbers (as in ¯1-cell and ¯2-cell) are also used to refer to cells whose corresponding frames are of the rank indicated by the magnitude of the number. For example, the list 'abcd' may be referred to either as a ¯2-cell or as a 1-cell of *a*, and each of the two separate 3 by 4 tables are called either ¯1-cells or 2-cells of *a*. The ¯1-cells of an array are its major cells, and an item has a single major cell, itself.

Open and boxed. The nouns discussed thus far are called *open*, to distinguish them from *boxed* nouns produced by the verb *box* (denoted by <). The result of box is an item and boxed nouns are commonly (but not necessarily) displayed in boxes. For example:

```
      <'here'
┌────┐
│here│
└────┘
```

Box allows one to treat any array (such as the list of letters that represent a word) as a single entity. For example:

```
      letters←'I was here'
      ⍴letters
10

      ⊖letters
ereh saw I

      words←(<'I'),(<'was'), (<'here')
      ⍴words
3

      ⊖words          (2,3)⍴words, ⊖words
┌─┬───┬────┐    ┌─┬─┐   ┌───┬────┐
│I│was│here│    │I│ │   │was│here│
└─┴───┴────┘    └─┴─┘   └───┴────┘
                ┌────┬───┬─┐
                │here│was│I│
                └────┴───┴─┘
```

B. VERBS

Most verbs are limited in their normal application, and the class of nouns to which a verb normally applies is called its *domain*. For example, the verb *minus* (in the expression -*n*) applies only to a numeric argument *n*, and the expression -'*n*' is meaningless; the domain of *minus* is limited to numeric arguments.

Monads and Dyads. Most verbs have two definitions, one for the *monadic* case (one argument), and one for the *dyadic* case (two arguments). If one of these definitions is omitted, the corresponding case of the verb has an empty domain. In any sentence, the dyadic definition of a verb applies if it is preceded by a suitable left argument, that is, any noun that is not itself an argument of a conjunction. Otherwise the monadic definition applies.

The monadic case of a verb is also called a *monad*, and we may speak of "the monad ÷" used in the expression ÷⍵, and of "the *dyad* ÷" used in the expression ⍺÷⍵.

Ranks of verbs. The notion of verb rank is closely related to that of noun rank: a verb of rank *k* applies to each of the *k*-cells of its argument. This notion will be introduced by an example using the verb *ravel* (denoted by a comma), which ravels its entire argument to produce a list of the elements. Thus:

```
      a←2 3 4⍴'abcdefghijklmnopqrstuvwx'
      a
abcd
efgh
ijkl

mnop
qrst
uvwx

      ,a
```

```
abcdefghijklmnopqrstuvwx
```

Since ravel applies to its entire argument, its rank is said to be *unbounded*, or *infinite*; it can also be applied to cells of a specified rank *r* by using the rank conjunction ⍤ in the expression ,⍤*r*. Thus:

```
      ,⍤2 a
abcdefghijkl
mnopqrstuvwx
      ⍴,⍤2 a
2 12
```

The last result illustrates a general rule: the shape of a result is the frame of the argument (relative to the cells to which the verb applies) catenated with the shape produced by applying the verb to the individual cells. Commonly these individual shapes agree, but if not, they are first brought to a common shape as follows:

1. If the ranks differ, they are brought to a common maximum rank *mr* by reshaping each argument to introduce leading unit lengths. Formally, an individual result *a* is replaced by `(u,⍴a)⍴a`, where `u←(mr-⍴⍴a)⍴1`.
2. If the individual shapes differ (after being brought to a common rank), each is brought to a common shape by the expression *ms*↑*a* , where *ms* is the maximum over the shapes. Thus, if the shapes are *s*1 and *s*2 and *s*3, then *ms*←*s*1⌈*s*2⌈*s*3. For example, the individual results of `⍳⍤0(2 3 4)` are `0 1` and `0 1 2` and `0 1 2 3`; the overall result is the table:

```
0 1 0 0
0 1 2 0
0 1 2 3
```

The case of a zero frame (which has no cells to which the function may be applied) is normally treated as follows: The shape of the individual result is determined by applying the function to a surrogate argument having the shape required for the argument cell. For example, if ⍴*m* is `0 5 4`, then the individual result shape for the case ⌹*m* is `4 5`, and the shape of the entire result is `0 4 5`. If the individual result shape can differ for different suitable surrogate arguments, it is taken to be the smallest shape over all surrogate arguments. For example, if ⍴*m* is `0 3`, the individual result shape of ⍳*m* has rank `1` and is of shape `0`. The shape of the overall result is `0 0`.

The dyadic case of a verb has two ranks, a *left* rank that governs the rank of the cells of its left argument, and a *right* rank that governs the rank of cells of its right argument. For example:

```
      p←'abc'
      q←3 4⍴'wakereadlamp'
      q
wake
read
lamp
      p,⍤0 1 q
awake
bread
clamp
```

Finally, each verb has three intrinsic ranks, a monadic rank, a left rank, and a right rank. This fact often simplifies the definition of a verb. For example, the monadic case of ⍎ is defined to have rank `1`, and it therefore suffices to define its behaviour on lists, perhaps by example, as in `⍎'abc' ↔ 'cba'` and `⍎1 2 3 ↔ 3 2 1`. The application of ⍎ to an argument of higher rank is therefore completely defined. For example:

```
      ⍎a
dcba
hgfe
lkji

ponm
tsrq
xwvu
```

Degenerate cases. The rank of a verb merely places an upper limit on the ranks of the cells to which it applies, and its domain may include arguments of rank lower than its nominal rank. For example, ⍎ has rank `1`, but its domain also includes an item *s* as follows: *s*≡⍎*s*. Similarly, ⌹ (matrix inverse) has rank `2`, but is extended to list and item arguments as follows: `(⌹a)≡⌹⍪a`, where the *table* function ⍪ forms a one-column table from a list or item argument.

Agreement. The two arguments of the dyadic case of a verb must *agree* in the following sense: the left frame and the right frame (relative to the particular verb) must be identical, except that if one frame is an empty list, the single corresponding cell is used as argument together with each cell of the other argument.

For example, if `q←3 4⍴'wakereadlamp'` and `p←'abc'` (as in the earlier example using the derived verb ,⍤`0 1`), then in the expression

```
p ,⍤0 1 q
```

the shapes of the arguments are `3` and `3 4`, the shapes of the cells (of ranks `0` and `1`) are empty (the list `''`) and `4`, and the frames are `3` and `3`. The result (as shown in the earlier example) has shape `3 5`, the `3` being contributed by the common frame, and the `5` by the shape of the catenation of individual cells.

The same arguments in the expression

```
p,⍤1 1 q
abcwake
abcread
abclamp
```

illustrate the exceptional case; the left frame is empty, the single cell `'abc'` is *extended* to apply to each of three cells of the right argument.

If one of the ranks is unbounded, the extension of the corresponding argument will always occur. For example, since the indexing verb *from*, `{`, has zero left rank and unbounded right rank, we have:

```
      p←3 4⍴'abcdefghijkl'
      p
abcd
efgh
ijkl
      2 0{p
ijkl
abcd
```

The *tie* conjunction (denoted by a period) can be used to relax the normal agreement constraints by specifying the number of leading axes of the frames that must agree, and "freeing" the remaining frame axes to interact independently. For example:

```
      p←1 2 3
      g←1 2 3 4
      m←p 0 .+ q
      m
2 3 4 5
3 4 5 6
4 5 6 7

      ⍴m 1 .+ m
3 4 4
      ⍴m 0 .+ m
3 4 3 4
```

C. ADVERBS AND CONJUNCTIONS

Unlike verbs, adverbs and conjunctions have a *fixed* valence; an adverb is monadic (applying to a single argument to its *left*), and a conjunction is dyadic (applying two arguments, one on each side).

Each argument of a conjunction may be either a noun or a verb, and it may therefore produce as many as four distinct classes of results. For example, `v⍤n` produces a rank `n` function that applies the verb `v` to each cell of rank `n`; `n⍤v` produces a *cut* that applies the verb `v` to each of a set of segments cut along the first axis of the argument in a manner determined by the noun `n`; `v1⍤v2` produces the *composition* of the verbs `v1` and `v2`; and `n1⍤n2` yields a *constant verb* of rank `n2`, whose result for each cell is `n1`.

The conjunction `⍤` may therefore be referred to variously as the *rank*, *cut*, *composition*, or *constant* according to its use, or may be referred to by the single term on, which is vague enough to roughly cover all of the cases, as in "ravel on reverse" for `,z⍤⌽`, and "ravel on `2`" for `,⍤2`.

The most common result of an adverb or conjunction is a verb, but it may also be a noun, adverb, or conjunction.

D. COPULA

In addition to certain *distinguished* names (discussed in the following section), the names that may be assigned by the copula are `⍺` and `⍵` and `→` and `$` and those that begin with a letter of the native alphabet and (may) continue with letters or digits, as in `ABc←3` and `a2←3`, and `A2b3←3`, or with a macron or dot, as in `A¯b¯2` and `a.b`. This restriction prohibits the redefinition of tokens, as in `3←2` or `+←2` or `+←x`.

In the expression `(1{'abc')←3+4`, the parentheses force evaluation of the expression `1{'abc'` before the assignment, and the name represented by the result is assigned the result of `3+4`. Thus the name `b` is assigned the value `7`. Name assignments of the form `(n)←x` are called *indirect*.

More generally, if the shape of `n` agrees with the outer shape of `x`, then `(n)←x` assigns the name represented by the open of each element of `n` to the corresponding cell of `x`. For example, if `x←3 4⍴⍳12` and `n1←'abc'` and `n2←<⍤1(3 2⍴'p0p1p2')`, then:

```
   n1         n2              x
abc       |¯¯||¯¯||¯¯|    0  1  2  3
          |p0||p1||p2|    4  5  6  7
          |__||__|1__|    8  9 10 11

   (n1)←x    (n2)←x          'abc'←3 4 5
   b         p0              b
4 5 6 7    0 1 2 3         4
```

In other words, `(n)←x` implies that `(>i{n)←>i{x` for each item `i{n` of `n`, and corresponding cell `i{x` of `x`. However, if `>i{n` is itself boxed (and therefore not a proper name), it and the corresponding cell are both opened, and assignment is re-attempted. Thus, if `n←(<'p'),(<<'q'),<<<'r'` then `(n)←3⍴<<'cat'` assigns `<<'cat'` to `p` and `<'cat'` to `q`, and `'cat'` to `r`.

An alternate form of the copula (denoted by ↵ or the overstrike of characters ← and |) is discussed in Section J.

E. DIALOGUE

Any user of a language will normally wish to control his own pronouns, in the sense that the value assigned by him to any name will not be affected by assignments made by others to the same name. On the other hand, one partner in a *dialogue* may use a pronoun assignment made by the other, as in:

Walter is coming.

Oh, when will *he* arrive?

In general, a mutual agreement by two users to *share* a specific name can be used to provide communication

between them, and such communication (via one or more shared names) can provide the basis for arbitrarily complex collaboration.

If one partner wishes to ensure that the other partner has actually consulted the current value assigned to a shared name before he assigns a new value, the partners can achieve this by sharing one or more further names, and using them in a strict protocol to signal and acknowledge dispatches (assignments) and receipts (uses) of the values of the primary shared name.

APL computer systems that provide for sharing names also provide protocols that control delivery and receipt of values assigned to shared names, and manuals for specific systems should be consulted. A good basic statement is provided by [7]. Most systems limit sharing to names of *nouns*, and prohibit the sharing of names of verbs and adverbs.

A set of *distinguished* names (beginning with ⎕ or ⍞) is reserved for communicating with the APL system that executes APL sentences. Three such names, used uniformly in dialects, merit consideration here.

As may have been apparent from earlier examples of the execution of APL sentences, the result of a sentence such as `⍳3` is automatically displayed, but the result of a sentence whose execution terminates in an assignment (such as `a←⍳3`) is not. Display can be forced by the prefix `⎕←`, as in:

```
      ⎕←⍳3
3 2 1
```

the name ⎕ denoting, in effect, the display mechanism of the system.

The name ⍞ is shared with the *terminal driver* (that controls the input-output device). When referred to (as in `b←⍞`) it, in effect, denotes the keyboard, and assumes the literal value of the list of symbols next entered on it. For example:

```
      a←⍳⍞
eva can i stab evil live bats in a cave
      a
evac a ni stab evil live bats i nac ave
      ⍴a
39
```

When assigned a value (as in `⍞←d`), ⍞ behaves like ⎕, except that it signals the terminal driver to suppress the final "carriage return", that is, it leaves the cursor at the end of the output displayed. The detailed behaviour of `⍞←d` varies considerably between different APL systems.

Finally, `⎕io` (called *index origin*) is not used here, but merits comment because it occurs in all dialects, affecting the behaviour of the verbs ⍋, ⍒, ⌽, ?, and ⍳. If `⎕io` is assigned the value `0`, the behaviour in any dialect agrees with that described here. If `⎕io←1`, then each element of `⍳⍵` is increased by `1`, and the other functions (which yield or use indices drawn from `⍳⍵`) are affected accordingly.

F. COMPARATIVES

In the everyday use of comparisons, a reasonable relative tolerance is implied. For example, a statement that two three-foot shelf boards are equal in length would normally imply that they agree to a fraction of an inch, whereas a statement that two cities are equidistant from a third would normally imply that the distances agree to within a mile or so.

In APL the comparison `⍺=⍵` is treated similarly, yielding `1` (for true) not only if ⍺ and ⍵ are identical, but also the difference `⍺-⍵` falls relatively close to zero. The relative tolerance used in comparisons is specified by the system variable `⎕ct` (called comparison tolerance). If the magnitude of the difference `⍺-⍵` does not exceed `⎕ct` times the larger of the magnitudes of ⍺ and ⍵, then `⍺=⍵` yields `1`.

The application of tolerance in other comparisons (such as <, ≤, and ≠) is detailed in the discussion of these verbs. Tolerance also applies to the verbs ⌊ (*floor* or *integer part*) and ⌈ (*ceiling*), which yield integer results, effectively by comparing the argument with neighbouring integers. Exact comparisons may be obtained by setting `⎕ct` to `0`.

G. TERMINOLOGY

Standard names and synonyms. Because of the rather large number of verbs, adverbs, and conjunctions in APL, it is important to choose names for them that are both distinctive and suggestive. *Standard* names are used in Sections IV-VI, and Table 4 lists synonyms that may be appropriate in narrower contexts. For example, the verb ⍴ (called *shape*) yields the number of elements of a list to which it is applied and is therefore often called "length", although such a term would be inappropriate in geometry, where the length of a list `b` (better called a *vector* in this context) is defined as `(+/b*2)*.5`.

Although many mathematical terms provide standard names (such as *plus*, *minus*, and *times*), others are unsuitable for various reasons:

- They have different meanings in different branches of mathematics, and therefore conflict; *inverse* is an example.
- They denote special cases only. For example, *determinant* and *permanent* (`-.×` and `+.×`) are special cases of the more general term *dot product*, and *transpose* is a special case of *cant* applicable to matrices only.
- They are rather awkward phrases, such as "divided by" for ÷, and "integer part of" for *floor*

(with no term for the companion concept of *ceiling*).

- They suggest at most part of a concept, as in "sign" or "signum" instead of *trend*. Not only does "sign" fail to suggest the meaning for a complex argument, but even for a real argument suggests only the way in which the result is represented, and not a "direction".

Distinct names for the monadic and dyadic cases of a verb are desirable, but not essential. Thus, *a*⋆*b* and *a*×⋆*b* are clear when read as "*a* power *b*" and "*a* times power *b*", and the phrases "the dyad power" and "the monad power" are as convenient as "the power function" and "the exponential function".

When working in a narrow and familiar context, one might find the suggested standard names bizarre and unnecessary, but in wider contexts come to accept them as reasonable compromises.

Other considerations. In mathematics the terms *scalar*, *vector*, and *matrix* are used for what we have here called *item*, *list*, and *table*; *function* and *operator* are used for what we have called *verb* and *adverb*. We will use these synonyms wherever they seem appropriate.

In verbalizing a written APL sentence, clarity may be gained by observing that the simple verbs ⊢ and ⊣ (called *right* and *left*) have the effect of coordinating or subordinating conjunctions. Thus, *a*←*n*÷*x*⊣*n*←⌽*x*←*l* 2 3 would be read as "*a* is *n* divided by *x*, *where n* is the reversal of *x*, and where *x* is the list 1 2 3", and ,⍤2⊢3 4⍴⍳12 would be read as "the rank 2 ravel *of* the 3 by 4 table of integers".

The occurrence of a copula without ⊣ may also be read as "where", as in "*a* is *x* times *x*, where *x* is 1 2 3", for *a*←*x*×*x*←1 2 3. Because of the common use of "and" for the verb ∧ in logic, it should probably *not* be used instead of "where", even though it may seem appropriate.

H. IDENTITIES AND PROOFS

Formal identities between sentences can play an important role in the use of formal languages. In the body of the dictionary we will adopt a rather widely used scheme for expressing identities: writing one sentence immediately below another will imply that the second is equivalent to the first. For example:

```
+/⍳n
+/⌽⍳n
.5×(+/⍳n)+(+/⌽⍳n)
.5×+/((⍳n)+(⌽⍳n))
.5×+/n⍴(n-1)
.5×n×(n-1)
```

The foregoing six sentences state five identities, all of which may be tested by executing them after assigning some value to *n*. Moreover, the five identities together imply an identity between the first and the last sentences, and therefore provide the well-known efficient calculation (.5×*n*×*n*-1) for the sum of a sequence of n successive integers beginning with zero.

The foregoing example can be elaborated to provide a *proof* of the identity between the first and last sentences by writing beside each sentence the basis for asserting its identity with the preceding sentence. For example:

```
.5×+/n⍴(n-1)
.5×n×(n-1)          Definition of × (that is,
                    p×q is the sum over p
                    repetitions of q)
```

Identities are also expressed by placing ↔ between sentences, as in +/⍳*n* ↔ .5×*n*×*n*-1, or by using the verb ≡, as in (+/⍳*n*)≡.5×*n*×*n*-1.

I. PARSING AND EXECUTION

A sentence is executed by executing its parts in a sequence determined by the parsing rules of the language. For example, the sentence 10÷3+2 is executed by *first* executing 3+2 to obtain a result that is then used to divide 10.

The parsing rules can be summarized as follows:

1. Execution proceeds from right to left, except that when a right parenthesis is encountered, the segment enclosed by it and its *matching* left parenthesis is executed, and its result is used to replace that entire segment and its enclosing parentheses.
2. Adverbs and conjunctions are executed before verbs; the phrase ,⍤2-*a* is equivalent to (,⍤2)-*a*, not to ,⍤(2-*a*). Moreover, the left argument of an adverb or conjunction is the entire verb phrase that precedes it. For example, in the phrase *a*+.×/*b*, the adverb / applies to the derived verb resulting from the verb phrase +.×, not to the verb ×.
3. Verbs are applied dyadically if possible. For example, *a*-*b* signifies a subtraction, but *a*×-*b* signifies multiplication of *a* with the negative of *b*.

One important consequence of these rules is that in an unparenthesized sentence the right argument of any verb is the result of the entire phrase to the right of it. A sentence such as 3×*p*⌈*q*⋆|*r*-5 can therefore be *read* from left to right: the overall result is three times the result of the remaining phrase, which is the maximum of *p* and the part following the ⌈, and so on.

It is also instructive to examine the explicit parsing process. Parsing proceeds by moving successive elements (or their *values* in the case of pronouns) from the tail end of a *left stack* (originally the given sentence prefixed by a marker ⍝) to the front of a *right stack*, and eventually

"executing" some *eligible* portion of the right stack and replacing it by the single result of the execution.

For example, if `a←1 2 3`, and if ◇ is used to separate the stacks, then the sentence `b←+/2×a` would be parsed and executed as follows:

```
⍝ b ← + / 2 × a ◇
⍝ b ← + / 2 × ◇ 1 2 3
⍝ b ← + / 2 ◇ × 1 2 3
⍝ b ← + / ◇ 2 × 1 2 3
⍝ b ← + ◇ / 2 × 1 2 3
⍝ b ← + ◇ / 2 4 6
⍝ b ← ◇ + / 2 4 6
⍝ b ◇ ← + / 2 4 6
⍝ b ◇ ← 12
⍝ ◇ b ← 12
⍝ ◇ 12
◇ ⍝ 12
```

The foregoing illustrates two important points: 1) Execution of the phrase `2×1 2 3` is deferred until the next element, the `/`, is transferred; had it been a conjunction, the `2` would have been *its* argument, and the monad `×` would have applied to `1 2 3`; and `2`) Whereas the *value* of the name `a` is moved to the right stack, the name `b` (because it precedes a copula) is moved unchanged. Moreover, a covert effect of the execution is that the pronoun `b` is assigned the value `12`.

The executions in the right stack are confined to the *first four elements only*, and eligibility for execution is determined only by the *class* of each of these elements (noun, verb, adverb, conjunction, copula, parenthesis, name, and left marker). Consequently, the parsing process can be made clearer by replacing each element in a sentence by a single chosen member of its class: `1`, `+`, `/`, and period for noun, verb, adverb, and conjunction; and the letters `a`, `b`, `c`, `d`, and `e` for names. The earlier example would then begin as follows:

```
⍝ b ← + / 1 + 1 ◇
⍝ b ← + / 1 + ◇ 1
⍝ b ← + / 1 ◇ + 1
⍝ b ← + / ◇ 1 + 1
⍝ b ← + ◇ / 1 + 1
⍝ b ← + ◇ / 1
```

Thus:

```
parse '+/a+a←1'
⍝+/a+a←1 ◇
    →
⍝+/a+a←◇1
   →
⍝+/a+a◇←1
    ∘
⍝+/a+◇a←1
    ↓↓↓
⍝+/a+◇1
 →
⍝+/a◇+1
  →
⍝+/◇1+1
   →
⍝+◇/1+1
    ↓↓↓
⍝+◇/1
 →
⍝◇+/1
→
◇⍝+/1
  ↓↓
◇⍝+1
  ↓↓
◇⍝1
```

The *parse* function illustrated at left is defined in Table 2; it is a complete formal statement of the parsing procedure that may be consulted for a deeper understanding of the process. It may be read by anyone conversant with APL, or entered on an APL system for experimentation of the kind illustrated here. In the accompanying example, the ◇ indicates the division between the "left (input) stack" and the "right (execution) stack"; the arrow → indicates that the element above it is evaluated and moved to the execution stack, the symbol ∘ indicates that the element is moved without evaluation (i.e., as a *name* to be assigned as a pronoun); and vertical arrows indicate the phrase to be evaluated and replaced by a single result, as where `1+1` (noun plus noun) is to be replaced by a `1` (a noun).

J. VERB DEFINITION

The conjunction ∇ provides a general means for defining a new verb, as discussed in Section VI. A simpler informal scheme called *direct definition* [6] will be adopted for definitions used for exposition in this dictionary. It defines a function by either one or three sentences, as illustrated below:

```
      sqrt: ⍵*.5
      sqrt 4 5 6
2 2.2361 2.4495

      root: ⍵*÷⍺
      2 root 64                3 root 64
8                          4
      f:  ⍵:⍺=1:~⍵
      0 f 0 1                  1 f 0 1
0 1                        1 0
```

The symbols ⍺ and ⍵ denote the left and right arguments. In a three-sentence definition, the middle sentence is a *conditional* which is executed first; the first or last sentence is then executed according to whether the result of the conditional is `0` or `1`.

Since direct definition is informal, its use on any APL system requires a translation provided by the functions *translate* and ⍙, themselves defined in the canonical form discussed in Section III. They appear in Table 3, together with examples of use.

Local and global names. A name may be *local* to a function in the sense that its use in the execution of the function has no relation to its use outside the function. For

example, in direct definition, the argument names ⍺ and ⍵ are local, as are any names that occur immediately to the left of a copula. For example:

```
    f:a×a ←1+b←2× ⍵
    a←b←⍵←3
    f 2
25
    a,b, ⍵
5 3 3
```

A name that is not local to a function is said to be *global* to it; a name not local to any function is said to be a *global* name.

In functions produced by the conjunction ⍛, the names ⍺, ⍵, ←, and $ are local, and other localizations may be produced *dynamically*: when any name *a* is to be assigned by an expression of the form *a*←*b* or (*a*)←*b*, then *a* is first made local if it is not already so. Expressions of the form *a*↵*b* and (*a*)↵*b* do not produce localization.

III. DIALECTS

The vexing question of what to include in a dictionary as standard, and what to relegate to dialects is settled here as follows: a construct is excluded if a) it is anomalous, and itself requires special rules, and b) is obsolescent, in the sense that its use can be avoided by the use of other (usually newer) constructs that are at least as convenient. Variations in word-formation were discussed in Section I.

Anyone beginning to write for a particular APL system should consult the manual for it, and should probably do so rather early so as to avoid the use of constructs that it does not include. On the other hand, the use of such alien constructs may prove beneficial, since they may lead to an improved style of programming, and may, in effect, be partially or fully incorporated into the dialect by designing functions to simulate them.

A sentence may fail to execute either because it is ill-formed (for example, 2+ or *a*← or *a*←←3) or because a verb is applied to arguments not in its domain. Most dialects provide a set of *error reports* which are used to indicate the type of failure in a sentence. The error report is normally followed by a display of the sentence, with a caret marking the point at which execution stopped.

Dialectal definitions are included in Sections IV-VI, together with explicit references to the relevant manuals. Certain dialectal constructions are excluded from such discussion because they depart too strongly from the grammar defined here. These excluded topics include *strands* (which, in effect, allow elision of the verb *link*), *ambiguous symbols* (which allow certain symbols such as / to denote either a verb or an adverb), and *selective specification* (such as (*c*/[*a*]*v*)←*x*).

Certain important constructs excluded from standard APL occur in nearly all dialects and are therefore discussed here, even though such discussion cannot completely obviate the consultation of other manuals. In each case the major reasons for exclusion are presented, as are alternative phrasings in the standard language.

A. BRACKET – SEMICOLON INDEXING

Brackets and semicolons are commonly used for *indexing* in dialects, and a good definition may be found in [7]. For example:

```
     ⊢a←2 3 4⍴⍳24
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23

     a[1;2;1]     a[1;1 2;1]     a[1;;1]
21           17 21          13 17 21
```

Deficiencies of bracket-semicolon indexing include:

a) Anomalous syntax due to the use of two separated symbols ([and]) for the indexing function. In particular, this leads to different interpretations of phrases such as a[*i*;][*j*] , and *a*[] in different dialects.

b) The phrase enclosed in brackets does not obey normal rules. For example, it has no explicit result to which a name can be assigned (as in *a*[*i*←(*j*;*k*)]), and "implied" parentheses surround the portions separated by semicolons (that is, *a*[2×*i*;*j*]↔*a*[(2×*i*);*j*]). This form of indexing can easily be expressed in terms of {. For example:

```
    a[i;j;k] ↔ (<i⊃j⊃k){ a
    a[i;j;]  ↔ (<i⊃j){ a
    a[i; ;k] ↔ (<i⊃⍬⊃k){ a
```

On the other hand, an expression such as (0 1⊃1 2⊃2 3){*a* (that provides "scattered indexing", to yield an array whose major cells are *a*[0;1;] and *a*[1;2;] and *a*[2;3;]) is *not* easily written in the bracket-semicolon form.

B. INDEXED ASSIGNMENT

The effect on the value of the name *a* produced by an expression of the form *a*[*i*]←*b*, is to assign to it a *merge* of the values of *a* and *b* controlled by the index *i*. The same effect can be obtained by applying the *merge* adverb } to the selection function *i*⍤{, that is, *a*←*b* *i*⍤{}*a*.

Indexed assignment shares the deficiencies already noted for the related bracket-semicolon indexing. Moreover, it applies only to the particular form of

indexing provided by the brackets, rather than to any function, as in `l ⌽[0]⍉t` to merge the list *l* in as the diagonal elements of *t*. Finally, the result of the expression `a[i]←b` (as assigned to the name `c` in the expression `c←a[i]←b`) is *not* the merged result, which can be obtained only by a separate reference to *a*.

C. BRACKET AXIS NOTATION

Expressions such as `⌽[i]a` and `+/[i]a` and `+\[i]a` apply the (possibly derived) function preceding the brackets "along axis i" of the argument *a*. For example:

```
      ⊢a←2 3⍴⍳6
0 1 2
3 4 5

      ⌽[0]a          +/[0]a          +\[0]a
3 4 5          3 5 7           0 1 2
0 1 2                          3 5 7
```

Phrases of the form `f[i]a` can be re-expressed in terms of the rank conjunction. For example, `⌽[i]a` and `+/[i]a` and `+\[i]a` can be expressed as `⊖⍤k a` and `+⌿⍤k a` and `+⍀⍤k a`, respectively, where `k←(⍴⍴a)-i`.

Similar remarks apply to a dyad, that is, `a,[i]b` is equivalent to `a ⍪⍤k b ⊣k←(⍴⍴a)-i`.

A fractional value of *i* in the expression `a,[i]b` provides *lamination*, inserting a new axis of length 2 between axes `⌊i` and `⌈i`. For example:

```
    a←3+b←⍳3
    a,[.5]b              a,[¯.5]b
3 0                  3 4 5
4 1                  0 1 2
5 2
```

The verb `,⍤<` is equivalent to `,[¯.5]`, and `,⍤<⍤k ↔ ,[¯.5+(⍴⍴a)-k]`.

D. CANONICAL FUNCTION DEFINITION

In most dialects, the system function `⎕fx` applies to a character matrix argument that represents a function in canonical form, and establishes the definition of the function. The first row of the matrix is called a *header*; it is a paradigm of the use of the function (such as `'z←a plus b'`) followed by a list of those names that are to be made local to the function, each prefaced by a semicolon. For example:

```
      m                        ⎕fx m
z←a root b;c           root
c←÷a
z←b*c
      2 root 64                3 root 64
8                      4
```

The major anomaly to be noted in canonical definition is that although `⎕fx` produces a function (verb), it is itself a function rather than an operator (adverb or conjunction). Moreover, it produces a defined function not as an explicit result to which a name may be assigned, but rather as a covert effect, a function with a specific name determined by the header of the argument of `⎕fx`.

The major deficiencies are:

a) The inability to make self-reference to the function being defined, as provided by the symbol `$` in the definition produced by the conjunction `⍛`, by any means other than the explicit name specified in the header. This makes it impossible to produce a recursive definition in which the function can be safely renamed. An attempt to make a systematic name substitution in the argument of `⎕fx` will founder on any use of execute (`⍎`) on pronouns.

b) A branch in canonical definition can specify only a beginning point *b*, the subsequent sequence being limited to the specific sequence `b+1`, `b+2`, etc., to the point of the next branch. In a definition of the form `m⍛d`, the branch `→←l` can specify an arbitrary sequence of any length, as determined by the list *l*. For example, the expression `→←(k⍴2),4` will repeat (*k* times) the single line `2`, and then finish with `4`; and if `l1` is any label, then `→←,>k⍴<l1` will repeat *k* times the entire segment whose beginning is labelled by `l1`.

Most APL systems provide special editing facilities to make convenient the revision of functions defined in canonial form. Corresponding editing facilities can be provided for direct definition by writing editing functions in APL using `⎕fx edit ⎕cr 'f'`, where `⎕cr` yields the matrix that represents the function named *f*. However, the general editing facilities provided on modern computing systems (particularly those with screen terminals) often make such special editing unnecessary.

APL systems also provide facilities for monitoring and controlling the execution (*trace* and *stop*) of canonically defined functions. Editing to insert expressions such as `⎕←` or *m* (where *m* is any desired monitoring function whose explicit result is its argument) can provide equivalent facilities for functions produced by the conjunction `⍛`.

Any function definition of the form `⎕fx t` can be mimicked by an expression of the form `f←m⍛d`, and the appropriate values of *m* or *d* can be obtained from *t* rather simply. For example, in any assignment to a global variable, the arrow `←` must be replaced by `↵`, and a branch of the form `→u/l1,l2,l3` may be replaced by `→←>0{u/l1⊃l2⊃l3`.

Niladic functions. The function *f* defined by `⎕fx 2 4⍴'z←f z←0a'` is said to be *niladic* because the header dicates no explicit argument; it behaves syntactically as

noun, although its result may vary because of its dependence on the pronoun *a*.

The behaviour of a niladic function is therefore the same as that of a normal function provided with an argument. If the result of a function does not depend on the explicit argument, then any argument may be used. For example, if *g*←'0*a*'∇'', then the phrase *g*∘ is equivalent to the niladic function *f* defined above.

Adverbs and Conjunctions. Some dialects provide for the definition of adverbs and conjunctions, using the canonical scheme used for defining functions, but allowing the use of parentheses in the header as follows:

```
Z←A(F ADV G) B      Conjunction → Conjunction
Z← (F ADV G) B      Conjunction → Adverb

Z←A(F ADV) B        Adverb → Conjunction
Z← (F ADV) B        Adverb → Adverb
```

E. COMMENTS AND STATEMENT SEPARATORS

Anything following a comment symbol (⍝) in an expression is ignored in its execution. Comment can be similarly added to the end of a line by appending ⍝' *This is a comment*', but can also be inserted anywhere in a line. Thus:

```
a←⍵× ⊢'count to n'⊢ ⍳n← ⊢'length of r'
     ⊢ ⍴r.
```

Expressions using the statement separator (⋄) can be mimicked by expressions using the verb *left*. The primary difference is that the separation imposed by ⍝ follows the normal rules for order of execution. For example, either of the following expressions will determine the coefficients *c* of a polynomial equivalent to a polynomial with roots *r*:

```
n←⍴r⋄b←n⍴2⋄t←b⊤⍳×/b⋄s←(⍳1+n)∘.=+⌿t⋄p←r×
      .*t⋄c←s+.×p

c←s+.×p⍝p←r×.*t⍝s←(⍳1+n)∘.=+⌿t⍝t←b⊤⍳×/b
      ⍝b←n⍴2⍝n←⍴r
```

Since ⍝ is a normal verb with simple properties, the phrase *b*⍝*b* can be simplified to *b*, and (since *p* is used nowhere else) the phrase *p*⍝*p*← can be omitted entirely, allowing the second expression to be simplified to:

```
c←s+.×r×.*t⍝s←(⍳1+n)∘.=+⌿t←b⊤⍳×/b←n⍴2⍝n←⍴r
```

The difficulty with the seemingly-simple statement separator is that it raises questions about many issues, such as the behaviour of a branch between separators, and the interaction between separators and comments. An indication of the complexity is given by the index entry for "diamond" in Berry [8]; references are made to eight distinct sections of the manual.

F. PERMISSIVETREATMENT OF ONE-ELEMENT ARRAYS

Most dialects are permissive in allowing one-element lists (and sometimes one-element arrays of any rank) to be treated exactly as the corresponding scalar. For example, the shapes of (,2)⍴⍳5 and 2⍴⍳5 are both 5, although the shape of the former should be 1 5.

In dialects which assign ranks to primitive verbs, agreement with the established permissive definition is obtained by making the rank unbounded, thus permitting any desired behaviour. In such cases, standard behaviour for a primitive *p* can be obtained by imposing the proper rank *r*, as in *p*⍤*r*.

For example, the fundamental definition of antibase (⊤) is on a list left argument and a scalar right argument, and it therefore has rank 1 0. However, because the extra result axis was traditionally placed *first* rather than last, dialects assign it unbounded right rank. Standard behaviour can be obtained by using ⊤⍤1 0. For example, most dialects would yield:

```
      ⊢m←2 2 2⊤⍳2*3
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
```

whereas 2 2 2⊤⍤1 0 ⍳2*3 would yield ⍉*m*.

IV: VERBS

The ranks specified for a function are *very important* reading its definition, since the basic definition is given only for cells of the indicated rank, and extension to higher-rank arrays follows the general rules for verbs stated in Section II.B. For example, the ranks of the verb ⍴ are 1 0 1 (monadic, left dyadic, and right dyadic, respectively), and ⍴⍵ need therefore be explicitly defined only for the simple case of a list. Similarly, the dyadic case need be defined only for a scalar left argument and a list right argument. For example:

```
      2⍴⍳5                    1 3⍴2 5⍴⍳10
2 3 4 0 1               1 2 3 4 0
                        8 9 5 6 7
```

Unless otherwise clear from the definition of a verb, the rank of the result it produces is the same as the rank of its right argument. For example, the functions +, -, ×, ⌊, and *, all produce item results, but ⍺?⍵ (as stated in its definition) does not.

The first line of each main entry contains:

- symbols of the verbs treated
- **Rank**: followed by the ranks in the order *monadic*, *left*, *right*, each indicated by a digit, the symbol ¯ (for unbounded), or * for an undefined case

- the heading `v≠:` followed by the identity element (or `⋆` if none exists) for each function, as defined in the discussion of `≠` in Section V
- the names of the verbs (as in Floor/Ceiling; Minimum/Maximum for `⌊ ⌈`). Although these names are suggested for general use, they are often replaced by words suited to a given context. See Section II.G and Table 4.

It should not be surprising that many of the names do not have the *form* of English verbs; English phrases such as `3` *and* `4`, `3` *plus* `4`, `3` *times* `4`, and `p` *or* `q` denote the action by prepositions or conjunctions rather than by verb, and even where a verb is available (as in *deny* `p` for `~p`) it may be more appropriate to use an adjective (as in *not* `p`).

`+ - × ÷ Rank: 0 0 0  v≠: 0 0 1 1`
Mate/Minus/Trend/Per; Plus/Minus/Times/Per

Monad. The following definitions and examples apply:

Mate	`+⍵ ↔ (∣⍵×⍵)÷⍵`	`+3j5 ↔ 3j¯5`
Minus	`-⍵ ↔ 0-⍵`	`-7 ↔ ¯7`
Trend	`×⍵ ↔ ⍵÷∣⍵`	`×3 0 ¯5 ↔ 1 0 ¯1`
Per	`÷⍵ ↔ 1÷⍵`	`÷4 ↔ .25`

Dyad. These functions are defined as in elementary arithmetic, except that `0 ÷ 0` is defined as `0`, for reasons presented by McDonnell [9]. Most dialects define `0÷0` as `1`, and most restrict the domains to real numbers.

`⋆ ⍟ Rank:0 0 0  v≠:1⋆` Power/Log; Power/Log

Monad. The *exponential* [10] denoted by `⋆⍵` is equivalent to `e⋆⍵`, where `e` is the base of the natural logarithms, given approximately by:

```
      ⋆1
2.718281828
```

The natural logarithm `⍟` is inverse to `⋆`, that is, `⍵ ↔ ⍟⋆⍵ ↔ ⋆⍟⍵`. Moreover, `⍟⍵ ↔ e⍟⍵`.

Dyad. `a⋆2` and `a⋆3` and `a⋆.5` are the square, cube, and square root of `a`. The general definition of `⍺⋆⍵` is `⋆⍵×⍟⍺`, and applies for complex numbers as well as real. For the simple case of an integer right argument it is equivalent to `×/⍵⍴⍺`; in particular, `×/` applied to an empty list yields `1`, and `⍺⋆0` is `1` for any `⍺`, including the case where `⍺` is zero. The expression `⍺⋆⍵` is often read as "`⍺` *to the power* `⍵`".

The base-`b` logarithm `b⍟⍵` is inverse to power, in the sense that `⍵ ↔ b⍟b⋆⍵ ↔ b⋆b⍟⍵`.

`< Rank: ¯ 0 0  v≠: 0` Box; Before

Monad. The result of `<⍵` is a *scalar encoding* of `⍵` in the sense that it has rank `0` and can be decoded (by `>`). Moreover `<⍵` differs from `⍵`. See the discussion of boxed nouns in Section II, and of their display under `⍕`.

Dyad. The result of `⍺<⍵` is `1` if `⍺` is less than `⍵`, and is `0` otherwise. Also see `≤`.

`≤ Rank: 1 0 0  v≠: 1` Cycle; Fore

Monad. If `s` is a mix (as defined under `⌷`), and if `s{a` replaces `i{a` by `j{a`, and `j{a` by `k{a`, and `k{a` by `i{a`, then the list `i,j,k` is said to be a *cycle* of the mix `s`. Every mix can be conceived as a collection of disjoint cycles of various lengths, including length one for those elements that do not move. For example, if `s←4 5 2 1 0 3`, then `s` has the cycles `5 3 1` and `4 0` and `2`.

A mix `c` is said to be a *cycle* representation of a mix `s` if the cycles of `s` are the boxed elements of the vector `b←(a=⌈\a)1⌸<a←(⍴c)⊤c`. For example:

```
      c←2 4 0 5 3 1
      (c=⌈\c) 1⌸<c
┌─┬┬───┬┬─────┐
│2│⊂4 0⊃│⊂5 3 1⊃│
└─┴┴───┴┴─────┘
```

and `c` is therefore the cycle representation of the mix `s←4 5 2 1 0 3` used above.

If `s` is a mix, then `≤s` yields its cycle representation; `≥` is the inverse function, and `≥≤s` is the standard form of `s`. The results of both `≥` and `≤` are mixes in standard form.

More generally, if the elements of `⍵` are distinct nonnegative integers, then `(≤⍵)≡≤⍵,(⍳⌈/⍵)~⍵`. For example, `(≤3 5 1)≡≤3 5 1 0 2 4`. In other words, the missing elements of `⍳1+⌈/⍵` are first appended to produce a complete permutation.

Dyad. The result of `⍺≤⍵`, is `1` if `⍺` is less than or equal to `⍵` and `0` otherwise. However, the comparison is made with a *tolerance* specified by the system variable `⎕ct` as follows: `⍺≤⍵` is `1` if `∣⍺-⍵` does not exceed `⎕ct` multiplied by the larger of their magnitudes, that is, if `∣⍺-⍵` is less than or equal to `⎕ct×(∣⍺)⌈∣⍵`.

Similar comparisons apply to the other relations. For example, `⍺≥⍵` is `1` if `∣⍺-⍵`, is greater than or equal to `-⎕ct×⌈/∣⍺,⍵`, and `⍺>⍵` is `1` if `(⍺≥⍵)∧~⍺=⍵`. The *custom* conjunction (`:`, which see) may be used with any of the relations, that is, `⍺<:t ⍵` provides comparison with a tolerance `t`, regardless of the value of `⎕ct`.

The relations are commonly applied to boolean arguments. For example, `⍺≠⍵`, is the *exclusive-or* of boolean `⍺` and `⍵`, and `⍺≤⍵`, is *implication*.

≥ Rank: 1 0 0 v≠: 1 Mix; Aft

Monad. The function ≥ is the inverse of ≤, and ≥⍵ produces the standard representation of the mix whose cycle representation is ⍵. See ≤ and].

More generally, if the elements of ⍵ are distinct non-negative integers, then ≥⍵, is the mix whose elements are determined by ⍵, in the normal manner, but whose missing elements (⍳⌈/⍵)~⍵, are treated as cycles of length 1, and therefore stay fixed. For example, (≥5 3 1)≡0 5 2 1 4 3, and the corresponding standard cycle representation ≤≥5 3 1 is 0 2 4 5 3 1.

Dyad. The result of ⍺≥⍵, is 1 if ⍺ is greater than or equal to ⍵, and is 0 otherwise. Also see ≤.

> Rank: 0 0 0 v≠: 0 Open; After

Monad. Open (>) is the inverse of box (<), that is, ⍵≡><⍵. When applied to an open array (that contains no boxed elements), open has no effect. For example:

```
      ><123
  1 2 3
      >123
1 2 3
```

The opened elements are brought to a common shape as discussed in Section II.B.

Dyad. The result of ⍺>⍵ is 1 if ⍺ is greater than ⍵, and is 0 otherwise. Also see ≤.

= Rank: ¯ 0 0 v≠: 1 Nub in; Equal

Monad. The function = classifies the major cells of the nub of ⍵ according to equality with the major cells of ⍵, producing an *m* by *n* boolean table, where *m* and *n* are the number of major cells of the nub of *a* (that is, ↑*a*) and of *a*, respectively. For example:

```
    a          ↑a          =a
allah       alh         1 0 0 1 0
                        0 1 1 0 0
                        0 0 0 0 1
    b          ↑b          =b
abc         abc         1 0 1
def         def         0 1 0
abc
```

Formally, =⍵↔(*MC*↑⍵)∘.=*MC*⍵, where *MC*:,<⍤¯1 ⍵ boxes the major cells of its argument. Hence, =*s* ↔ =,*s* for any scalar argument *s*.

Dyad. The function = is a rank 0 form of match (≡); formally, ⍺=⍵↔⍺≡⍤0 ⍵. See ≡ and ≤. For example:

```
      ⊢a←3 5⍴'abcdefghijklmno'
abcde
fghij
klmno
    a=⌽a                    ⊢b←<⍤1 a
0 0 1 0 0       ┌─────┬┬─────┬┬─────┐
0 0 1 0 0       │abcde│⋄│fghij│⋄│klmno│
0 0 1 0 0       └─────┴┴─────┴┴─────┘

    b=⌽b
0 1 0
```

Dialects. Some define = differently on boxed arrays; it may therefore be necessary to use ⍺≡⍤0 ⍵ instead of ⍺=⍵.

≠ Rank: ¯ 0 0 v≠: 0 Nubsieve; Unequal

Monad. Nubsieve (≠⍵) provides a boolean list that selects the nub of ⍵, that is, ↑⍵ ↔ (≠⍵)/⍵. For example, ≠'abacus' ↔ 1 1 0 1 1 1, and ≠3 ↔ 1⍴1. Formally, ≠⍵↔(c⍳c)=⍳0{⍵c←,<⍤¯1 ⍵.

Dyad. The result of ⍺≠⍵ is 1 if ⍺ is unequal to ⍵, and is 0 otherwise. Also see ≤.

≡ Rank: ∗ ¯ ¯ ; Match

Dyad. The expression ⍺≡⍵ yields a scalar boolean result; 1 if the arguments match completely, in shapes, in boxing structure, and in elements. The comparison of numeric elements is made under the normal rules of comparison tolerance described under ≤. In particular, the custom conjunction applies to ≡.

~ Rank: 0 ¯ ¯ Not; Less

Monad. ~ applies only to boolean arguments, and negates them: ~0 1 ↔ 1 0.

Dyad. The result of ⍺~⍵ is the array whose major cells are the major cells of ⍺ *less* the major cells of ⍵. For example:

```
    (⍳6)~7 2 4
0 1 3 5
    ⍺←3 4⍴'abcdefghijkl'
    ⍵←2 4⍴'mnopabcd'
    ⍺~⍵
efgh
ijkl
```

Formally, ⍺~⍵ ↔ (~(*MC* ⍺)∈ *MC* ⍵)/⍺, where *MC*: ,<⍤¯1⍵. The result of *a*~*a*~*b* is called the *intersection* of *a* and *b*. For example, if *a*←⍳6 and *b*←7 4 2, then *a*~*a*~*b* is 2 4. The result for any scalar argument *s* is the same as for ,*s*.

∧ ∨ ⍲ ⍱ Rank: ∗ 0 0 v≠: 1 0 ∗ ∗ ; And (LCM)/Or (GCD)/Nand/Nor

Dyad. *p*∧*q* is the least common multiple of *p* and *q*, and *p*∨*q* is the greatest common divisor. For example:

```
      12∨30
      12∧30
```

```
6                              60

      12 30⊤12∨30              ∨/12 30⊤12∨30
2 5                    1

      3.6∨4                    3.6∧4
0.4                    36
```

For boolean arguments (`0` and `1`) the `∨` is equivalent to the logical *or*, and the `∧` to logical *and*. Thus:

```
      p←0 0 1 1
      q←0 1 0 1
      p∧q                      p∨q
0 0 0 1                0 1 1 1
```

Finally, `⍲` and `⍱` (called *nand* and *nor*) are defined only on boolean arguments, and are negations of `∧` and `∨`, that is:

```
      (α⍲ω)≡(~α∧ω)             (α⍱ω)≡(~α∨ω)
```

`⊢ ⊣` Rank: `¯ ¯ ¯` Right/Left; Right/Left

Monad. Both `⊢` and `⊣` are *identity* functions, that is, `⊢ω↔ ω`, and `⊣ω ↔ ω`.

Dyad. Left yields the left argument, and Right the right, that is, `α ↔ α⊣ω` and `ω ↔ α⊢ω`.

`⊥` Rank: `1 1 1` Words; Base

Monad. `⊥` provides the word formation described in Section I. For example:

```
      ⊥ '3 4.5×ABC DEρQ'
|¯¯¯¯¯||¯||¯¯¯||¯¯||¯||¯|
|3 4.5||×||ABC||DE||ρ||Q|
|_____||_||___||__||_||_|
```

It is equivalent to the function WORDS defined below:

```
words:f(c g(q∧¯1↓0,q←∨≠3↑c)⍱q∧¯1↓0,q)
      1⍥<,ω⊣q←q∨≠\q←,¯1↑c←(<,ω)∊⍥>p
   f:(0≠,ρ⍥>c)/c←(r⊥⍥>r←⌽⍥>a)↓⍥>(-a⊥
     ⍥>a←ω=⍥>' ')↓⍥> ω
   g:ω\q⍲¯1↓0,q←ω/(∨≠α[1 3;])∨
     (α[2;])∧1↓(∨≠α[1 2;]),0
```

The parameter p is a 5-element list of boxed lists consisting of the quad (`⎕`) and the native alphabet, the macron (`¯`) and the decimal digits, the period (`.`), the space, and the quote (`'`).

Dyad. `α⊥ω` is the item `+/W×ω`, where `W` is the list of *weights* such that `k{W` is `×/(k+1)↓(ρω)ρα`. For example if `α←24 60 60` and `ω←1 2 3`, then `10⊥ω` is `123`, and:

```
      α⊥ω          ⊢W←1↓⌽×\⌽α,1          +/W×ω
3723          3600 60 1              3723
```

Dialects. Most dialects differ in their treatment of higher rank arguments, for both `⊥` and `⊤`.

`⊤` Rank: `⋆ 1 0` ;Antibase

Dyad. For simple cases, `⊤` is inverse to the base `⊥`. For example, if `α←24 60 60`, then `α⊥1 2 3` is `3723` and `α⊤3723` is `1 2 3`. More generally (since the largest number representable in base `α` is `α⊥α-1`), `α⊥α⊤ω` equals `(×/α)|ω` rather than `ω`. For example, `(3ρ10)⊤3247` is `2 4 7`, and `(3ρ10)⊥2 4 7` is `247`, that is, `(×/3ρ10)|3247`.

`⍎` Rank: `1 1 1` Execute; Execute

Monad. The result of `⍎ω` is the result of executing the character list `ω`. For example, `⍎'a←3+4'` assigns the value `7` to the name `a`. The result of `⍎''` is `⍬`.

Dyad. The result of `α⍎ω` is the result of `⍎ω` if `ω` is a valid expression (that is, its execution completes properly). Otherwise, the result of `α⍎ω` is the result of `⍎α`, which may itself invoke an error report. Effects may also be produced by any partial execution of `ω` that has occurred.

`⍕` Rank: `¯ 2 2` Format; Format

Briefly stated, the format function produces a character array that represents numeric results in familiar forms. However, the exact rules for rounding the numbers, for inserting spaces to align columns, and for replacing leading and trailing zeros by spaces, become rather intricate.

Monad. If `ω` is an open character array, then `ω≡⍕ω`. Although the monadic rank of `⍕` is unbounded, we will first restrict attention to arguments of rank `2`.

If `ω` is an open numeric, then `⍕ω` is, with the exceptions stated below, equivalent to `((m+1),⎕pp)⍕ω`, where `m` is the greatest width required for any of the fields, and where `⎕pp` is the system variable that controls printing precision:

> Leading zeros preceding the decimal point and trailing zeros following it are replaced by spaces, except that if all digits are zero, the one before the decimal point is retained, and the decimal point itself is replaced by a space.

In extending the definition of `⍕` to arguments of rank greater than `2`, it is necessary to choose a value of `m` for use in the expression `((m+1),⎕pp)⍕ω` that is large enough for all of the individual rank `2` cells. We therefore define a dyadic function `F` of rank `0 2` such that `n F ω` is equivalent to `⍕ω` if `2=ρρω` and if `n` is the maximum field width required. Moreover, if `om` is the overall maximum width required for all rank `2` cells of an argument `ω`, then `om F ω` provides a useful extension to arguments of higher rank. We adopt this extension as a definition of `⍕`, with a modification that

inserts rows of spaces between the results for individual cells. Using the subsequent definition of ⍪ on boxed arrays, these insertions may be expressed as follows:

If `3=⍴⍴⍵`, and if `r←⍕⍤1 F ⍵`, then the result of `⍪⍵` matches `⍪⌿<⍤2 r`, evaluated with `⎕ps←0 0 1 0`.
If `3<n←⍴⍴⍵`, then `⍪⍵` matches `⍪⌿<⍤2 ⍪⍤(n-1) ⍵`.

The degenerate cases of vector and scalar arguments

are defined by the expression `,⍪⍀⌿,⍵`.

The normal display of any result `r` matches the display of `⍪r`. For example:

```
      1 100∘.+.1 1∘.×¯1 0 1
0.9       1       1.1
0         1       2.

99.9    100     100.1
99      100     101
```

If the argument of ⍪ is a boxed array, a width sufficient for all elements in a column of the display is allotted for each column, and sufficient height is similarly allotted to each row. The justification within the allotted space is then determined by the first two elements of the "position and spacing" system variable `⎕ps` as follows: justify top, centre, or bottom in each row if `0{⎕ps` is `¯1`, `0`, or `1`, and left, centre, or right in each column if `1{⎕ps` is `¯1`, `0`, or `1`.

Space between successive rows and columns is alloted as follows: `⌈2{⎕ps` spaces between successive rows, and `⌈3{⎕ps` spaces between successive columns. Finally, each element is bordered by horizontal lines (`¯` and `_`) immediately above and below if `2{⎕ps` is negative, and by vertical lines immediately to the left and right if `3{⎕ps` is negative. For example:

```
   b←2 2⍴(3 5⍴'ab')⊃1 2⊃'cd'⊃'efghij'
   ⎕ps←¯1 1 3 3
   b
|¯¯¯¯¯|    |¯¯¯|
|ababa|    |1 2|
|babab|    |___|
|ababa|
|_____|

|¯¯|       |¯¯¯¯¯¯|
|cd|       |efghij|
|__|       |______|

  ⎕ps←0 0 0 0
  b
ababa
babab 1 2
ababa
 cd  efghij
```

All other examples of boxed arrays in the dictionary are shown as if `⎕ps←¯1 1 2 2`.

The custom conjunction (which see) applies to ⍪ to specify the position and spacing independently of `⎕ps`, as in `⍪⍠(¯1 1 3 3)`.

Dialects. Display, and the result of ⍪, differ somewhat among dialects. However, most control display and the result of ⍪ by the system name `⎕pp` (printing precision), whose value determines the maximum space allotted to the printing of each number. They also use `⎕pw` to control the *printing width* (the point at which the line is broken to continue indented on a subsequent line); this applies to normal display, but not to the result of ⍪.

Dyad. The expression `⍺⍪⍵` yields a character array; the form of the result is controlled by ⍺ in two different ways, depending upon whether ⍺ is numeric or character.

If ⍺ and ⍵ are both numeric tables, the result of `⍺⍪⍵` is a table of `1↑⍴⍵` rows, in which groups of adjacent columns (called *fields*) are determined by each column of ⍵ together with a two-element control vector `c`. The control for field `j` given by `c←j{⍺`; the element `0{c` determines the width of the field, and `1{c` determines the number of decimal places used.

If `k←1{c` is negative, the result is represented in *exponential form*, as illustrated below:

```
      (⍀⌿8 ¯3)⍪ ⍀⌿123.45678 .00876543
1.23E2 8.77E¯3
```

To obtain the exponential form of a number `n`, it is first *normalized* to the equivalent `m×10*p`, where `p` is an integer, and `0≤⌈m`, and `10>⌈m`. The number `m` is then rounded to `⌈k` places (as described below) and is suffixed by one of the forms `e¯qr` or `e qr`, where `qr` denotes the representation of `⌈p`.

The rounding of a number `n` is determined by `k←1{c` as follows:

```
(10*-⌈k)×⌊.5+(⌈n)×10*⌈k
```

The character string that represents this result is then prefixed by enough spaces to bring it to the width determined by `0{c`, after being prefixed by the symbol `¯` if `n` is negative.

The width of a field is equal to `0{c`, unless `0{c` is zero, in which case the width is one more than the maximum width required to represent any one of the numbers in the entire column of ⍵.

The degenerate case of a scalar or vector right argument is treated as `,⍺⍪⍀⌿⍵`. The degenerate case of a (2-element) vector left argument is treated as `(((1↑⍴⍵),2)⍴⍺)⍪`, and the case of a scalar left argument ⍺ is treated as the vector `0,⍺`.

Dialects. Dialects commonly use the ravel of the table left argument described above, rather than the table itself. For example:

```
   ⊢a←3 2⍴24.34 ¯57.684 ¯0.45 ¯134.27
24.34     ¯57.684
¯0.45     ¯134.27
24.34     ¯57.684

   12 3⍕a
      24.340    ¯57.684
      ¯0.450   ¯134.270
      24.340    ¯57.684

   5 2 8 0⍕a                8 2 0 ¯2⍕a
24.34   ¯58           24.34   ¯5.8e1
¯0.45  ¯134           ¯0.45   ¯1.3e2
24.34   ¯58           24.34   ¯5.8e1
```

If ⍺ is a character array, the expression ⍺⍕⍵ is called *format by example*, since ⍺ provides a pattern for the result. In the standard case of a single cell, ⍺ is a one-row table, and the last dimension of the table ⍺⍕⍵ agrees with the last dimension of ⍺.

Since the left rank of ⍕ is 2, the standard case requires a table left argument ⍺; since that table consists of a single row, the equivalent degenerate case (,⍺)⍕⍵ is commonly used. Degenerate cases of the right argument ⍵ are treated as ,⍺⍕⍉⍪,⍵. For example:

```
  ⍺←'Balance is  $(55,510.50) on 05/55/55'

  ⍺⍕23456.714 61184
Balance is 23,456.71 on 06/11/84

   ⍺⍕456.78, 100⊥6 11 84
Balance is 456.78 on 06/11/84

   ⍺⍕¯23456.714 61184
Balance is $(23,456.71) on 06/11/84

   ⍺⍕¯456.78 61184
Balance is $(456.78) on 06/11/84

   ⊢r←'555.55'⍕1 0 10.1 100          ⍴r
1          10.1   100              24
```

A numerical field in ⍺ is bounded by blanks or (on the right) by the first non-control character following a six, and must contain at least one digit. The digits in a field are both place-holders and control characters for that field. Non-digits are *decorators*, simple, controlled, or conventional:

- A simple decorator may be embedded in a numerical field or stand alone; it is reproduced in place.
- A controlled decorator is a group of one or more characters immediately adjacent to a leading or trailing digit in a numerical field that contains a 1, 2, or 3 nearer that decorator than a 4. For example, if *y* is `1 ¯1∘×123 321`, then:

```
 'CONTROL55154NOT  NOT543215CONTROL'⍕y
           123NOT  NOT   321
    CONTROL123NOT  NOT   321CONTROL
```

- The dot and comma are conventional decorators; they specify decimal digits or group separators according to common conventions (as in `23,456.78`). A dot is a decimal point conventional decorator if it either precedes the first digit or is surrounded by digits in a numerical field, and if it is the only *such* dot in the field. A comma is a conventional decorator if it is surrounded by digits in a numerical field. For example:

```
      '5...55.55' ⍕ 12.3  123.45
... 1       ...12.3 1...23.45
     'LF,55,525,,5,RT'⍕ 12345 ¯12345 123 ¯123
LF,1,234,,5,RT   1,234,,5  LF,12,,3,RT  12,,3
```

These conventional dots and commas *print* as the elements `0{⎕fc` and `1{⎕fc` of the *format control* variable ⎕*fc*. (The normal value of `0 1{⎕fc` is `'.,'`; reversal of these values provides printing according to a common European convention.)

A decimal point not followed by a fractional value is a part of the trailing decorator, and as such is suppressed according to the rules that apply to any trailing decorator. For example:

```
      '-5125.POS'⍕ 123 ¯123
123.POS -123
```

The control characters have the following significance:

`0` **Leading/trailing zeros**: unused positions are filled with `0` from here toward the decimal point. For example:

```
      '55,550.09'⍕1 0 1000.1
1.00      0.0  1,000.10
```

`1` **For negative values**, the decorator text on this side (of the decimal point) is included and floats (to abut the digits of the number); otherwise, it is replaced by blanks.

The effects of the digits `1`, `2`, and `3` do not depend in any way on the decimal point. For example:

```
      'THIS1555.55HERE'⍕ 12 ¯12
  12             THIS12HERE
```

`2` **For nonnegative values**, the decorator text on this side is included and floats; otherwise, it is replaced by blanks.

`3` **Float**: for all values, the decorator text on this side floats.

`4` **No float**: for all values, the decorator text on this side does not float; nullifies the floating specified by a `1`, `2`, or `3` further from this side. For example:

```
      '→→55125.5←← →→54124.5←← →→54125.5←←
          55124.5←←' ⍕ 1 ¯1 ⊗.×4⍴1234
1234←←  →→ 1234  ←← →→  1234←←       1234  ←←
→→1234    ********** →→  1234     →→1234  ←←
```

5 **Normal digit**: position available for digit, or for sign decorator (when supplied and selected by 1, 2, or 3).

6 **Field delimiter**: next rightward noncontrol character (that is, any character other than 0123456789.,) starts a new field; needed where fields are not separated by blank.

7 **Exponential format**: value is represented in exponential notation; next rightward noncontrol character (for example, *e*) separates characteristic from mantissa. Engineering notation differs from scientific notation in requiring all exponents to be multiples of 3. For example:

```
  '¯1.75E5  VS  ¯155.75E5' ⍕ (10*⍳8)*.×1 1
1.00E0      VS       1.00E0
1.00E1      VS      10.00E0
1.00E2      VS     100.00E0
1.00E3      VS       1.00E3
1.00E4      VS      10.00E3
1.00E5      VS     100.00E3
1,00E6      VS       1.00E6
1.00E7      VS      10.00E6
```

8 **"Cheque protection"**: unused positions from here to the decimal point are filled with 2{⎕*fc*.

```
  '$558,555,535.50'⍕ 123 12345 1234567
$******123.00   $***12,345.00    $1,234,567.00
```

9 **Conditional zero-fill**: effect same as 0 for nonzero values, but blank when value is zero.

Notes.

a) The digit 6 provides an additional field break at the next non-control character. This allows, for example, the use of slashes desired in display of a date as field delimiters as well. For example:

```
      '06/06/05' ⍕ 7 11 84
07/11/84
```

b) The system variable ⎕*fc*, referred to under digit 8, contains six elements whose normal values are the characters: . , * * _ ¯.

c) If space in a field is insufficient to represent a large element of *⍵*, then an error is evoked if 3{⎕*fc* is '0'; otherwise the element 3{⎕*fc* is used to fill the offending field.

d) 4{⎕*fc* is a "pseudo-blank"; that is, its occurrence in a decorator displays a blank, but the character does not delimit a field as a blank would.

○ Rank: 0 0 0 — Pi; Circle

Monad. The expression ○*⍵*, is equivalent to *pi*×*⍵*, where *pi* is the ratio of the circumference of a circle to its diameter, and is given approximately by:

```
  ○1
3.141592654
```

Dyad. The expressions *k*○*⍵*, and (-*k*)○*⍵*, produce several families of related functions, *circular* or *trigonometric* (for *k*∈1 2 3), *hyperbolic* (for *k*∈5 6 7), *pythagorean* (for *k*∈0 4 8), and *complex* (for *k*∈9 10 11 12). The trigonometric functions are based on radian arguments, and the sine of *d* degrees would therefore be written as 1 0○*d*÷180. All apply to complex arguments.

Corresponding positive and negative cases are *inverse* in the sense that at least one of the identities *⍵*≡*k*○(-*k*)○*⍵* or *⍵*≡(-*k*)○*k*○*⍵* hold (at least within the principal domains, as defined in [10]). The cases 9○*⍵* and 11○*⍵* give the real and imaginary parts, and 12○*⍵*, gives the *arc* of a polar representation.

As a mnemonic aid it may be noted that (except for *k*=12) the expression *k*○*⍵* is even (that is, *k*○*⍵* ↔ *k*○-*⍵*) if *k* is *even*, and is odd (that is, *k*○*⍵* ↔ -*k*○-*⍵*) if *k* is *odd*. The definitions follow:

```
     0           1           2           3
(1-⍵*2)*.5   sin ⍵     cos ⍵       tan ⍵    +
(1-⍵*2)*.5 arcsin ⍵ arccos ⍵   arctan ⍵    -

     4           5           6           7
(1+⍵*2)*.5   sinh ⍵    cosh ⍵      tanh ⍵   +
(¯1+⍵*2)*.5 arsinh ⍵ arcosh ⍵   artanh ⍵    -

     8           9          10          11
(¯1-⍵*2)*.5 (⍵++⍵)÷2     |⍵    (⍵-+⍵)÷0j2   +
-(¯1-⍵*2)*.5     ⍵        +⍵      ⍵×0j1      -

    12          13          14          15
(⊗×⍵)÷0j1        ⍵        *⍵      *⍵×0j1     +
*⍵×0j1           ⍵        ⊗⍵     -0j1×⊗⍵     -
```

The custom conjunction (which see) can be used to specify the number of units per quadrant, as in 1○:90 *d* to determine the sine of an angle given in degrees. Formally, *k*○:*q* *⍵* ↔ *k*○.5×○*⍵*÷*q*.

⌽ Rank: 1 0 1 — Reverse; Rotate

Monad. *Reverse* reverses the order of a list. For example:

```
      ⌽'abc'                ⌽2 3⍴'abcdef'
cba                   cba
                      fed
```

Dyad. *Rotate* cycles the elements of a list as illustrated below:

```
     2⌽l←'abcdef'
cdefab
     ¯2⌽l
efabcd
```

For example:

```
   ⊢t←2 3⍴l        1 2⌽t          1⌽t
abc            bca            bca
def            fde            efd
```

Dialects. Dialects commonly depart from this definition in two significant ways:

1. If s is an item, then expressions such as `(,s)⌽ω` and `(1 1⍴s)⌽ω` are treated as equivalent to `s⌽ω` rather than contributing outer axes to the result.
2. Since the dyadic ranks are `0 1`, a right argument of rank `1` should be extended, as illustrated by:

```
   1 2 3⌽'abcd'
bcda
cdab
dabc
```

Dialects may yield an error report instead, even in those [5] which produce the foregoing result when the rank adverb is applied, as in `1 2 3⌽⍤0 1 'abcd'`.

⊖ Rank: ¯ ¯ ¯ — Upset; Rowel

Monad. *Upset* overturns its argument as illustrated below:

```
   ⊢t←3 4⍴'abcdefghijkl'          ⊖t
abcd                          ijkl
efgh                          efgh
ijkl                          abcd
```

It may be expressed in terms of reverse as follows:

```
(⊖ω)≡⍉⌽⍉⌽ω
```

Dyad. *Rowel* is equivalent to `⌽⍤⍉`. For example:

```
   0 1 2 3⊖t
afkd
ejch
ibgl
```

⍉ Rank: ¯ 1 ¯ — Cant; Cant

Monad. This function reverses the order of axes of its argument. Thus:

```
     ⊢m←3 4⍴⍳12          ⍉m          ⍴⍉m
 0  1  2  3          0  4  8       4 3
 4  5  6  7          1  5  9
 8  9 10 11          2  6 10
                     3  7 11
```

It may be defined in terms of the dyad *cant*: `(⍉ω)≡(⌽⍳0{⍴⍴ω)⍉ω`.

Dyad. If p is any permutation of the axes of array a, then $b←p⍉a$ is similar to a except that its axes are permuted; axis i of a becomes axis $i\{p$ of b. For example:

```
   ⊢b←(p←1 2 0)⍉a←2 3 4⍴⍳24
 0  4  8        ⍴b
12 16 20     4 2 3

 1  5  9        p{⍴b
13 17 21     2 3 4

 2  6 10        ⍴a
14 18 22     2 3 4

 3  7 11        ⍴1 0 4 2 3⍉2 3 4 5 6⍴9
15 19 23     3 2 5 6 4
```

More generally, $q⍉a$ is defined if `(⍴q)=⍴⍴a` and if the nub of q is a permutation (of $⍳n$, for some n). For example, each of the following values of r, s, and t are valid left arguments of `⍉` for a right argument of rank `4`:

```
  ↑r←2 0 1 2    ↑s←1 0 1 1    ↑t←0 0 0 0
2 0 1          1 0            0
```

Just as for the case $b←p⍉a$ where p is a permutation, the ith axis of a becomes the $i\{q$ axis of $b←q⍉a$. However, in this case, two or more axes of a may map into a single axis of b, providing a diagonal section of a. For example:

```
   a←3 3⍴⍳9                c←3 5⍴⍳15
0 1 2                   0  1  2  3  4
3 4 5                   5  6  7  8  9
6 7 8                  10 11 12 13 14

   0 0 ⍉a                  0 0 ⍉c
0 4 8                   0 6 12
```

Formally, if i is any complete index of $q⍉a$ (that is, `(<i){q⍉a` selects a scalar element of $q⍉a$), then `(<i){q⍉a` is equivalent to `(<q{i){a`.

Dialects. Since the elements of the left argument of `⍉` are drawn from $⍳n$, they commonly depend on the value of `⎕io`, as discussed in Section II.E.

| Rank: `0 0 0` *v*≠: `0` — Size; Residue

Monad. The definition `(ω×+ω)*.5` yields the *size* or "absolute value" not only for a real argument, but also for a complex argument, in which case the result is the length of the hypotenuse of a right-angled triangle whose side-lengths are the real and imaginary parts of the number. For example, `|3j4` is `5`, and `|¯6` is `6`.

Dyad. The most familiar use of residue is to determine the remainder resulting from dividing a non-negative integer dividend by a positive integer divisor. For example:

```
     3|0 1 2 3 4 5 6 7 8
0 1 2 0 1 2 0 1 2
```

The definition `ω-α×⌊ω÷0=α` extends this notion to a zero left argument (giving the right argument unchanged), to non-integer right arguments, and to negative and fractional left arguments. For a negative integer left argument, the result ranges between the argument and zero, as it does for a positive left argument. For example:

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

    1|2.5 3.64 2 ¯1.6
0.5 0.64 0 0.4
```

However, in order to produce a true zero (rather than a small fraction) for cases such as `(÷3)|2÷3`, the residue is made "tolerant" by the following definition:

`α|ω ↔ ω-α×⌊s if (α≠0)∧(⌈s)≠⌊s←ω÷α+α=0`
`↔ ω×α otherwise`

For example:

```
   .1|2.5 3.64 2 ¯1.6
0 0.04 0 0
```

This definition also applies to complex arguments, using the complex properties of the floor function `⌊`.

`!` Rank: `0 0 0` `v≠: 1` Factorial; Out of

Monad. For a non-negative integer argument, the factorial is defined by `!ω ↔ ×/1+⍳ω`. For example, `!4` is `×/1 2 3 4`, or `24`, and `!0` is `1`. For other arguments it is defined in terms of the *gamma* function [10] as `!ω ↔ gamma ω+1`.

Dyad. The dyadic case of `!` is defined in terms of the *beta* function [10]; subject to the interpretation given below, this definition is equivalent to `α!ω ↔ (!ω)÷(!α)÷(!ω-α)`. For non-negative integer arguments, `α!ω` yields the number of ways of selecting `α` things from `ω` things; this accounts for the name "out of", and for its use in producing binomial coefficients. For example, `(⍳n+1)! n←3 ↔ 1 3 3 1`.

For a negative integer *i*, the expression `!i` is not defined, because near a negative integer the magnitude approaches infinity. Nevertheless, the definition of `α!ω` can be understood by assuming that these infinite values occur in the expression for the dyadic definition in the following sense:

1. If `α` and `ω` are both non-negative, but `α>ω` then the term `!ω-α` is infinite, yielding `0` for the result of `α!ω`. This agrees with the notion that `α` things can be picked from a smaller collection in 0 ways.
2. If infinite values occur in both numerator and denominator of the defining expression, they are assumed to cancel. This can be seen in the values in the following function table:

```
 i  0  .!i←(⍳7)-3
 1   ¯2   1  0  0  0  0
 0   1  ¯1  0  0  0  0
 0   0   1  0  0  0  0
 1   1   1  1  1  1  1
 ¯3  ¯2  ¯1  0  1  2  3
 6   3   1  0  0  1  3
¯10  ¯4  ¯1  0  0  0  1
```

`⌊ ⌈` Rank: `0 0 0` `v≠: ¯ ¯¯` Floor/Ceiling
Minimum/Maximum

Monad. The expression `⌊ω` gives the *floor* or *integer part* of `ω`. For example, `⌊3.1 ↔ 3`. However, the implied comparison with integers is tolerant (see `≤`), and the definition of `⌊ω` is therefore the largest integer that does not exceed `ω+⎕ct`. The *ceiling* `⌈ω` is defined by `⌊¨-ω` or, equivalently, `-⌊-ω`. For example, `⌈2+10*¯14 ↔ 2` for a normal setting of `⎕ct`.

The definitions of floor and ceiling on complex numbers are rather involved, and the interested reader should experiment with them on an APL system, or consult McDonnell [11].

Dyad. `a⌊b` yields the lesser of *a* and *b*, and `a⌈b` yields the greater. For example, `3⌊4` is `3`, and `3⌊¯4` is `¯4`. Complex numbers are not in the normal domain of `⌊` and `⌈`.

`↑ ↓` Rank: `¯ 1 ¯` Nub/Raze; Take/Drop

Monad. `↑ω` selects the *nub* of `ω`, that is, all of its distinct major cells. For example:

```
   ω        ↑ω        ↑3          ⍴↑3
ABC      ABC       3           1
ABC      DEF
DEF
```

Formally, `↑ω ↔ (≠ω)≠ω`.

The *raze* function is defined by `(↓ω)≡⊃¨>≠ω`, and therefore assembles along a leading axis the opened elements of `,ω`. For example:

```
   ⊢ω←2¨⊂'Now is the time '
┌────┬───┬────┬─────┐
│Now │is │the │time │
└────┴───┴────┴─────┘

   ↓ω
┌────────────────┐
│Now is the time │
└────────────────┘
```

Dyad. If `(⍴α)=⍴⍴ω`, and if all elements of `α` are nonegative, then the shape of `r←α↑ω` is `α`; if *i* is any complete (boxed) index of both *r* and `ω`, then `(i⌷ω)≡i⌷r`; if *i* is an index of *r* only, then `i⌷r` is an array of *fill elements*, zeros if `ω` is entirely open numeric

or empty (`0=⌿⍴⍵`), spaces (`' '`) if ⍵ is entirely open character, and `<''` otherwise. For example:

```
      ⊢⍵←3 4⍴⍳12          2 2↑⍵          3 6↑⍵
0  1  2  3          0 1          0  1  2  3  0  0
4  5  6  7          4 5          4  5  6  7  0  0
8  9 10 11                       8  9 10 11  0  0
```

More generally, ⍺ may have negative items, which cause selection from the trailing end of the corresponding axes, and if (`⍴⍺)<⍴⍴⍵` then ⍺ takes the (`-⍴⍺`)-cells of ⍵ as defined by `⍺↑⍵` ↔ `(⍺,(⍴,⍺)↓⍴⍵)⍴⍵`. For example:

```
      2 ¯2↑⍵          3 ¯6↑⍵                  2↑⍵
2 3           0  0  0  1  2  3        0 1 2 3
6 7           0  0  4  5  6  7        4 5 6 7
              0  0  8  9 10 11
```

As illustrated by the case `3 ¯6↑⍵` above, the fill elements are placed in the leading positions in the case of a negative argument.

Drop is defined similarly, and `⍺↓⍵` drops elements from ⍵, from the trailing end if the element of ⍺ is negative. If (`⍴⍺)=⍴⍴⍵` then the shape of `⍺↓⍵` is `0⌈(⍴⍵)-|⍺`. For example, `¯2 2↓⍵` ↔ `1 2⍴2 3`. More generally, `(⍺↓⍵)≡((⍴⍵)↑⍴,⍺)↓⍵`.

Dialects. Because of permissive treatments of left arguments such as `(1 1,⍴V)⍴V` as a vector, dialects commonly treat the left rank of these verbs (↑ and ↓) as unbounded. Moreover, *k*`↑0⍴`*a* usually treats the empty argument `0⍴`*a* like *a* itself, "filling" with `0` or `' '`. For example, `3↑0⍴⍳5` ↔ `0 0 0`, and `3↑0⍴'abc'` ↔ `3⍴' '`.

⍋ ⍒ Rank: ¯ ¯ ¯ Grade/Downgrade; Grade/Downgrade

Monad. If ⍵ is a list of real numbers, and if *g*`←⍋⍵`, then *g* is the *grade* of ⍵ in the sense that *g*`⌷⍵` is in ascending order. For example:

```
      ⊢g←⍋⍵←3 1 4 2 1 3 3          g⌷⍵
1 4 3 0 5 6 2                1 1 2 3 3 3 4
```

Moreover, elements of *g* that select equal elements of ⍵ (such as `1 4` and `0 5 6`) occur in ascending order.

If ⍵ is a table, ⍋⍵ grades the *rows*, that is, it grades the base value of the rows, using a base larger than the magnitude of any of the elements. For example:

```
      ⍵←4 3⍴3 1 4 2 7 9 3 2 0 3 1 4
      ⍵              ⊢g←⍋⍵              g⌷⍵
3 1 4          1 0 3 2            2 7 9
2 7 9                             3 1 4
3 2 0                             3 1 4
3 1 4                             3 2 0
```

Higher rank arguments are graded as if their major cells were ravelled, that is, `⍋⍵` ↔ `⍋,⍤¯1⍵`.

The definition of downgrade (⍒) is like that of upgrade, except that (`⍒⍵`)`⌷⍵` is in descending order.

Dialects. In most dialects the results of ⍋ and ⍒ depend upon the index origin `⎕io`, as discussed under *dialogue* in Section II.E.

Dyad. If ⍺ is a list, then `⍺⍋⍵` grades ⍵ according to the "collating sequence" specified by ⍺. Formally, `⍺⍋⍵` ↔ `⍋⍺⍳⍵`. For example:

```
      ⍺←'abcde'⍪⍵←'labelled'
      ⍺⍳⍵
5 0 1 4 5 5 4 3

      ⍋⍺⍳⍵                        ⍺⍋⍵
1 2 7 3 6 0 4 5          1 2 7 3 6 0 4 5

      (⍺⍋⍵)⌷⍵
abdeelll
```

In particular, since `⍺⍳`*s* yields `⍴⍺` for any scalar *s* not in ⍺ then (`⍺⍋⍵`)`⌷⍵` places all elements of ⍵ not in ⍺ at the end, in the same relative order that they have in ⍵.

If ⍺ is a table of two fonts such as:

```
      ⍺
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

then `⍺⍋⍵` grades ⍵ in *dictionary* order, that is, using the ordering of the fonts (as specified by the columns) as a secondary ordering *within* that specified by the rows. For example:

```
      ⍵                (⍺⍋⍵)⌷⍵
cabal              baker
Baker              Baker
baker              cabal
```

Since dyadic grade is extended to higher rank arguments in the same systematic manner that monadic grade is, we will state its general behaviour in terms of a list right argument only.

The result of `⍺⍋⍵` is based upon a ranking of the complete indices of ⍺ in ⍵. Because earlier axes of ⍺ provide *secondary* ordering within the ordering imposed by later axes, the ranking is based upon the monadic grade of the table of *reversed* complete indices. For example, using the two-font alphabet ⍺ and the value `⍵←'abcABC'`, the indices, reversed indices, and ranking are:

```
      i              ⌽i              ⍋⌽i
0 0            0 0            0 3 1 4 2 5
0 1            1 0
0 2            2 0
1 0            0 1
1 1            1 1
1 2            2 1
```

For any argument not in ⍺, the complete index is taken as ⍴⍺. Formally, then,

```
      ⍺⍋⍵ ←→ ⍋⌽⍺⍳⍵
```

where `I:((⍴⍺)⊤⍤1 0(,⍺)⍳⍵)+(~⍵∈⍺)×⍤0 1⍴⍺` gives the indices of ⍵ in ⍺.

Finally, if a value *s* occurs more than once in ⍺, its index is taken to be the minimum over all of its possible indices. For example, if ⍴⍺ is `3`, and the indices of *s* are `2 3 4` and `1 5 2`, the index is taken as `2 3 4⌊1 5 2`, that is, `1 3 2`.

Alternative formal definitions of dyadic ⍋ and examples of its use may be found in [14], and in the original paper [15] cited therein.

Dyadic downgrade (⍒) is defined analogously.

Dialects. Most dialects exclude a scalar right argument from the domains of ⍋ and ⍒, and also introduce a dependence upon the index origin `⎕io`, as discussed under *dialogue* in Section II.E.

∈ Rank: `¯ 0 ¯` Raze in; In

Monad. ∈⍵ is an autoclassification that determines for each element of ⍵ whether its open contains each element of the open of the raze of ⍵, that is, ∈⍵ ←→ ⍵∈⊂⍤⊃↓⍵. The result is a boolean of shape (⍴⍵),⍴↓⍵. For example:

```
      ⍵←'abc'⊃'dc'⊃'a'
      ↓⍵                          ∈⍵
┌──────┐                  1 1 1 0 1 1
│abcdca│                  0 0 1 1 1 0
└──────┘                  1 0 0 0 0 1
```

Dyad. The result of ⍺∈⍵ is `1` if ⍺ belongs to ⍵ in the sense that ∨/⍺⍷⍵, for example:

```
      'cat'∈'abcd'
1 1 0
```

∈ Rank: `* ¯ ¯` ;In

Dyad. If *b*←⍺∈⍵, then the **ones** in *b* indicate the beginning points of occurrences of the pattern ⍺ in ⍵. For example:

```
      'co'∈'cocoa'   (0 1∘.+⍳3)∈4|i∘.+i←⍳5
1 0 1 0 0        1 0 0 0 0
                 0 0 0 0 0
                 0 0 1 0 0
                 0 1 0 0 0
                 0 0 0 0 0
```

Formally, ⍺∈⍵ ←→ ((⍳1,⍴⍺)⍷⍤<⍵)∈<⍺.

⍳ Rank: `1 1 0` Count; Index

Monad. The verb ⍳ "counts" in the sense that ⍳*n* yields a list of the first *n* integers beginning with zero. Thus, ⍳3 is the list `0 1 2`. More generally, ⍳⍵ is an array of shape ⍵,⍴⍵ such that (⍴⍵)⌷⍤>*i* ←→ *i*⌷⍳⍵ for any index *i* that selects a `1`-cell of ⍳⍵.

Dialects. Many dialects treat any one-element argument as a scalar, rather than allowing it to provide an outer shape. Thus ⍳⍴*v*←`2 3 4 5` yields the four-element list `0 1 2 3` rather than a `4` by `1` table. Moreover, all dialects provide dependence on `⎕io` for both the monadic and dyadic cases, as discussed under *dialogue* in Section II.

Dyad. If *j*←⍺⍳⍵, then *j* is the index of the first occurrence of ⍵ in ⍺ (that is, ⍵≡*j*⌷⍺), unless ⍵ does not occur in ⍺, in which case *j* equals ⍴⍺. Formally, (⍺⍳⍵)≡+/∧\⍺∘.≠⍵, except that a scalar left argument is not permitted.

⍸ Rank: `* ¯ ¯` ; Index

Dyad. The function ⍸ is defined in terms of ∈ and *i*←⍺⍸⍵ yields the index in *b*←⍵∈⍺ of the first `1`, or the value ⍴*b* if ⍵ does not occur in ⍺. Thus:

```
      A                   B          A⍸B
0 1 2 0 1 2         4 5         1 1
3 4 5 3 4 5         2 0
1 2 0 1 2 0                     A⍸⍋B
4 5 3 4 5 3                 4 6
```

⍴ Rank: `¯ 1 ¯` Shape; Reshape

The monadic shape and dyadic reshape functions are simply related; ⍺⍴⍵ produces a result of shape ⍺ from elements of ⍵, and therefore ⍺≡⍴⍺⍴⍵. For example:

```
⊢r←2 3⍴q←'abcdef'
abc                          ⍴r
def                       2 3

   q≡,r          ⍴⍴r          ⍴⍴3
1             2           0
```

The last examples (⍴⍴*r* and ⍴⍴`3`) illustrate the fact that the rank of an array is the shape of its shape; and that the rank of an item is `0`, implying that its shape is an empty list; and that (⍳0)⍴,`3` produces an item as a result.

If the number of elements in *p* equals the number of elements in an array of shape *s* (that is, ×/*s*), then (,*s*⍴*p*)≡,*p*. If (⍴,*p*)>×/*s*, then only the first ×/*s* elements of ,*p* are used; if (⍴,*p*)<×/*s*, then ,*p* is used cyclically. For example:

```
   p←2 3⍴'abcdef'
   2 2⍴p                  3 5⍴p
ab                     abcde
cd                     fabcd
                       efabc
```

? Rank: `0 0 0` Roll; Deal

Monad. The function *roll* is named from the analogy with rolling a die to choose one of a set of numbers with equal probability; the set used for ?⍵ is ⍳⍵.

Dyad. The function *deal* is named by analogy with dealing from a pack of cards. The result of ⍺?⍵ is a list

of length ⍺, all elements being distinct. The population drawn from in each case is ⍳⍵. See Berry [8] for details of the most commonly used pseudo-random number generator.

Dialects. See Section II.E concerning dependence on ⎕*io*.

, ⍪ Rank: ¯ ¯ ¯ Ravel/Table; By/Over

Monad. *Ravel* (,) ravels the elements of its arguments in "normal" order. For example:

```
    ⊢⍺←2 3 4⍴'abcdefghijklmnopqrstuvwx'
abcd
efgh
ijkl

mnop              ,⍺
qrst        bcdefghijklmnopqrstuvwx
uvwx
```

Normal order means that the results are ordered by *cells*; that is, the elements of *i*{⍵ precede those of *j*{⍵ if *i*<*j*; the ¯2-cells (0{0{⍵, 1{0{⍵, etc.) are similarly ordered within the ¯1-cells, and so on.

The result of ,⍵ is a list of all elements of ⍵; its shape is 1⍴×/⍴⍵. In particular, the ravel of an item is a one-element list.

For a non-scalar argument, the expression ⍪⍵ produces a ravel of each of the major cells of ⍵, that is, (⍪⍵)≡,⍤¯1 ⍵. For a scalar argument, the result is a table of shape 1 1; the result for any argument is therefore a table. The complete formal definition therefore involves two applications of ,⍤¯1, the second one producing an effect only in the case of a scalar. Thus: (⍪⍵)≡,⍤¯1,⍤¯1⍵. For example:

```
    ⍪0 1 2       ⍪ 2 2 2⍴⍳8          ⍴⍪2
0             0 1 2 3              1 1
1             4 5 6 7
2
```

Dyad. The expression ⍺⍪⍵ catenates the major cells of ⍺ and ⍵. For example:

```
    ⊢⍺←2 4⍴'abcdefgh'                ⍺⍪⍵
abcd                              abcd
efgh                              efgh
                                  ijkl
    ⊢⍵←3 4⍴'ijklmnopqrst'         mnop
ijkl                              qrst
mnop
qrst
```

If either argument is an item, it is first "extended" by the expression (1,1↓⍴⍺)⍴⍵ or (1,1↓⍴⍵)⍴⍺. If either argument is of rank 1 less than the other, its shape is first augmented b y a leading 1, that is, by application of the expression >,<. The resulting major cells must then agree in shape. For example:

```
    ⍺⍪'A'          'ABCD'⍪⍵
abcd          ABCD
efgh          ijkl
AAAA          mnop
              qrst
```

The related function denoted by the comma is best defined in terms of ⍪, that is:

⍺,⍵ ↔ ⍺⍪⍤¯1⍉⍵ ↔ ⍉(⍉⍺)⍪(⍉⍵)

For example:

```
    ⍺,⍺           'AB',⍺         ⍺,'A'
abcdabcd      AAabcd         abcdA
efghefgh      BBefgh         abcdA
```

Dialects. Most dialects restrict catenation to arguments of the same type, and therefore avoid the introduction of "heterogeneous" arrays that may contain both numbers and alphabetics, but some [12 13] do not.

⊃ Rank: ¯ ¯ ¯ Box open; Link

Monad. ⊃ boxes an open argument and has no effect on a boxed argument. The expression ⍵≡⊃⍵ can therefore be used to determine whether ⍵ is open or boxed.

Dyad. ⍺⊃⍵ *links* ⍺ and ⍵, by boxing ⍺ and catenating it to ⍵, first boxing ⍵ if it is open. Thus, (⍺⊃⍵)≡(<⍺),⊃⍵. For example:

```
  'now'⊃'is'⊃'the'⊃'time'
┌───┬──┬───┬────┐
│now│is│the│time│
└───┴──┴───┴────┘
```

Dialects. In dialects [12 13], ⊃⍵ is a form of open (>), and ⍺⊃⍵ is a form of indexing of ⍵ by ⍺.

] Rank: 1 0 ¯ Mix index; Mix

Monad. If ∧/(*i*=⌊*i*),(*i*<*n*),*i*≥¯*n*←1↑⍴*a*, then *i* is an index of *a*, and the major cells of (,*i*){*a* are major cells of *a*. If (*n*←⍴*i*)⍴*i* is also a reordering or *permutation* of ⍳*n*, then *i* is called a *mix* of order *n*, and *n*⍴*i* is its *standard* form. There are exactly !*n* distinct standard mixes of order *n*, and if *S* is a !*n* by *n* table of them in lexical order (such that (⍋*S*)≡⍳!*n*), then *i* is the *mix index* of *i*{*S*.

If ⍵ is a mix, then]⍵ yields the mix index of (*n*←⍴⍵)⍴⍵; if it is not, the result is !*n*. For example, 2=]0 1 3 2 4 and 120=]0 1 3 0 4 and 4=]¯1 0 1.

Dyad. If *i* is an index of ⍳*n*←''⍴⍴*a*, then *i*]*a* is *p*{*a*, where]*p* is *n*⍴*i*. For example, (2]⍳4)≡0 2 1 3, and (¯22]⍳4)≡0 2 1 3, and 2]'*abcd*' is '*acbd*', and (⍳!*n*)]⍳*n* is the table of all mixes of order *n* in lexical order.

{ Rank: 1 0 ¯ All; From

Monad. The expression {*w* forms a catalogue from the elements of its argument. For example:

```
  ⍵←'hat'⊃'ao'⊃'gtw'     (2 3⍴⍳6)⊃10 11
  {⍵                   ┌────┬┬────┐
┌───┬───┬───┐          │0 10│└0 11┘
│hag│hat│haw│          └────┴┴────┘
└───┴───┴───┘          ┌────┬┬────┐
                       │1 10│└1 11┘
                       └────┴┴────┘
```

```
|hog||hot||how|      |____||____|
|___||___||___|      |====||====|
                     |2 10||2 11|
|---||---||---|      |____||____|
|tag||tat||taw|
|___||___||___|      |----||----|
|===||===||===|      |3 10||3 11|
|tog||tot||tow|      |____||____|
|___||___||___|      |====||====|
                     |4 10||4 11|
  ⍴{⍵                |____||____|
2 2 3                |====||====|
                     |5 10||5 11|
  ⍴{(2 3⍴⍳6)⊃10 11   |____||____|
2 3 2
```

As may be seen in the foregoing examples, the shape of the catalogue `{⍵` is the catenation of the shapes of the disclosed elements of `⍵`, that is, `>↓⍴¨>⍵`, and the common shape of the disclosed elements of the catalogue is `⍴⍵`.

Expressions of the form `{⍳¨>`*s* are useful for producing a complete array of indices of an array of shape *s*. For example:

```
   ⍳¨>s←2 3
|---||-----|
|0 1||0 1 2|
|---||-----|
   {⍳¨>s
|---||---||---|
|0 0||0 1||0 2|
|---||---||---|
|===||===||===|
|1 0||1 1||1 2|
|___||___||___|
```

Portions of such an array of indices are useful in conjunction with the dyadic case of `{`; in particular, `⍵≡({⍳¨>⍴⍵){⍵`.

The case `{(<⍺),<⍵` is called the *cartesian product* of `⍺` and `⍵`.

Dyad. For an integer scalar left argument in the range from `-1+⍴⍵` to `¯1+1↑⍴⍵`, `⍺{⍵` selects the major cell of `⍵` indexed by `(1↑⍴⍵)|⍺`, as in `2{`*a*`←5 3⍴⍳15 ↔ 6 7 8`, and `¯2{`*a* `↔ 9 10 11`; in other words, negative indices select from the tail end. Since the left rank of `{` is zero, the shape of `⍺{⍵` for any array of integers a is `(⍴⍺),1↓⍴⍵`. For example:

```
   i{'abcde'¬i←2 2⍴1 ¯1 0 ¯2      i{a
be                                  3  4  5
ad                                 12 13 14

                                    0  1  2
                                    9 10 11
```

More generally, each element of `⍺` may be a boxed vector, whose successive elements are (possibly) boxed arrays of integers which specify selection along successive axes of `⍵`. For example:

```
     a←<(<i0←1 2),(<i1←2 3⍴⍳6)
     b←3 6⍴'abcdefghijklmnopqr'

   b                          a{b
abcdef                     ghi
ghijkl                     jkl
mnopqr                     mno
                           pqr

  ⊢c←i0{⍤(¯,2) b           i1{⍤(¯,1) c
ghijkl                     ghi
mnopqr                     jkl
                           mno
                           pqr
```

For the case of a single-element (i.e., scalar) left argument, this selection along successive axes can be stated formally as follows (using only the simple definition of `{` given originally for an integer left argument):

```
  FROM: (>⍺)FR ⍵¬n←1+⍴⍴⍵
    FR: (1↓⍺)FR (>0{⍺){⍤(¯,n ←n-1) ⍵:0=⍴⍺:⍵
```

It may be noted that `(<'')⍵ ↔ ⍵ ↔ ⊂⍵`, even for a scalar `⍵`.

If any of the values of `>0{⍺` occurring in the execution of *FR* is *not* an open array, then the selection is made using the complement (with respect to all indices along that axis) of the indices in its open, that is in `>>0{⍺`. In other words, the indices selected are `(⍳l)~l|>>0{⍺¬l←(-n){⍴⍵`. For example, if `b←4 5 6⍴⍳120` and `a←<(<0 3),(<<2 ¯1),(<1 0 4)`, then the selection made along the middle axis in evaluating `a{b` includes all indices *except* `2` and `¯1`, that is, it includes `0`, `1`, and `3`, in that order.

Since the middle element in the open of `i←<I⊃⊂⊃⊂K` is the boxed boxed empty vector, the expression `i{b` selects *all* along the middle axis, and is equivalent to the dialectal form `b[I;;K]` discussed in Section III.A.

⌹ Rank: `2 ¯ 2` — Inverse; Inverse

Monad. For a non-singular matrix `m`, `⌹m` is the inverse of `m`, that is, `(⌹m)+.×m` is the identity matrix `i∘.=i←⍳0{⍴m`. More generally, `⌹m` is defined in terms of the dyadic case, as `(i∘.=i←⍳0{⍴m)⌹m` or, equivalently, by the relation `⍺⌹⍵ ↔ (⌹⍵)+.×⍺`. The shape of `⌹m` is `⌽⍴m`.

The degenerate cases (vector and scalar) are defined by using the table `⍪⍵` instead of `⍵`, although most dialects differ in yielding a result of the same shape as `⍵`, rather than a matrix. For a non-zero vector `v`, the result of `,⌹v` is `v÷+/v×+v`; that is, a vector collinear with `v`. For a scalar `s`, the result of `,⌹s` is `,s×÷s`.

Dyad. If the columns of `⍵` are linearly independent, and if the first elements of `⍴⍺` and `⍴⍵` agree, then `⍺⌹⍵` is defined so as to minimize the elements of `r←+⌿d×÷d←⍺-⍵+.×⍺⌹⍵`. If `⍵` is square, it is necessarily non-singular (since its columns are linearly independent), the elements of `r` are all zero, and `⍺≡⍵+.×⍺⌹⍵`.

Geometrically, `⍵+.×⍺⌹⍵` is, for vector `⍺`, the projection of `⍺` on the column space of `⍵`, that is, the point in the space spanned by the columns of `⍵` that is nearest to `⍺`. The most common uses of `⍺⌹⍵` are in the solution of

linear equations, and in the approximation of function polynomials.

As in the monadic case, the degenerate cases of ⍵ are treated as ⍳⍵, and disagreements with most dialects arise as noted under the monadic case.

V: ADVERBS

An adverb produces two different classes of results (usually verbs), one when it is applied to a noun and the other when applied to a verb. The derived results are therefore referred to by the symbol for the adverb precededed by *n* (for noun) or *v* (for verb). For example, *n*⌿ refers to the derived verb *copy* (for which `2 0 3⌿'ABC'` yields `'AACCC'`), and *v*⌿ refers to the derived verb *over* (for which `+⌿2 3 5` yields `10`).

The ranks are given as they are in the verb table, except that a rank may depend upon the monadic, left, or right rank of the argument verb *v*, indicated by *m*v, *l*v, and *r*v. As for the verbs, the definitions show the results for individual cells, and the derived verb applies to arguments of higher rank in the standard manner discussed in Section II.B.

n⌿ Rank: ¯ ¯ ¯ Copy down; Copy down

Monad. In the expression *n*⌿⍵, the argument ⍵ is split into its major cells, and cell *i*⌷⍵ is copied *i*⌷*n* times. Thus:

```
      ⊢⍵←3 3⍴'abcdefghi'
abe
def
ghi

      2 0 2⌿⍵
abc
abc
ghi
ghi

      2 0 2⌿⍳3
0 0 2 2
```

Scalar arguments. If *n* has a single element, it is treated as `(1↑⍴⍳⍵)⍴n`, for example, `2⌿⍳3 ↔ 0 0 1 1 2 2`. A scalar argument ⍵ (which has no "leading" axis), is treated as `(1↑⍴⍳n)⍴⍵`. Thus, `2 0 2⌿3 ↔ 3 3 3 3`.

Dyad. In the dyadic case of *n*⌿, the argument *n* may contain negative elements; a negative element copies major cells from the left argument of the derived verb rather than from the right. For example:

```
  'ABCDEFG' ¯1 0 1 0 1 0 0⌿'abcdefg'
Ace
  '␣'(¯1*s='')⌿s←'now is the winter'
now␣is␣the␣winter
```

v⌿ Rank: ¯ 0 ¯ *v*-Down; ⍺-Way *v*-Down

Monad. In the expression *v*⌿⍵, the argument ⍵ is split into its major cells, and the verb *v* is applied between them. Thus if `1↑⍴⍵` is `3`, the result is `(0⌷⍵)v(1⌷⍵)v(2⌷⍵)`. Consider, for example, the following identities:

```
    ⊢⍵←3 2⍴⍳6
0 1
2 3
4 5

  ∘.+⌿⍵
 0 1 ∘.+ 2 3 ∘.+ 4 5
 0 1 ∘.+ 6 7
         7 8
6 7
7 8

7 8
8 9

      +⌿⍵
(0⌷⍵)+(1⌷⍵)+(2⌷⍵)
0 1 + 2 3+ 4 5
0 1 + 6 8
6 9

  +.×⌿⍵
 0 1 +.× 2 3 +.× 4 5
 0 1 +.× 23
 23
```

Identity elements. If the leading axis of ⍵ has zero length (that is, `0=1↑⍴⍵`), the result of *v*⌿⍵ is the *identity element* of verb *v*. The *left identity* of *v* is a noun *l* such that *l v x* yields *x* for any *x* in the right domain of *v*, the *right identity* of *v* is a noun *r* such that *x*≡*x v r*. A left identity of a commutative verb (such that ⍺ *v* ⍵ ↔ ⍵ *v* ⍺) is, of course, a right identity as well, and may be called simply an *identity*. For example:

`1` is a right identity of both ÷ and ⋆
`1` is an identity of × and ∧
`0` is a left identity of ∣
`0` is an identity of + and ∨

The definition of each verb indicates its identity element (if any), listed after the symbols *v*⌿:. An element is included even though it is strictly a right identity or a left identity, and in some cases (such as for = and ≠) if it applies only over a subdomain (boolean).

Identity elements extend relations of the form `(+⌿a)≡(+⌿k↑a)+(+⌿k↓a)` to include the cases *k*=`0` and *k*=⍴*a*. Thus:

```
    a←2 3 5 7 11
   k←3
   (+⌿k↑a)+(+⌿k↓a)
   (+⌿2 3 5)+(+⌿7 11)
       10    +    18
28

      k←0
      (+⌿k↑a)+(+⌿k↓a)
      (+⌿⍳0)+(+⌿a)
          0   +   28
   28
```

Dyad. The expression `3 +⌿ v←2 3 5 7 11` produces "running sums" over successive `3`-element groups of contiguous elements of *v*, therefore yielding `10 15 23`. More generally, the major cells of *r*←⍺ *v*⌿⍵ are the results of *v*⌿⍺↑*k*↓⍵, for *k* running from zero to *n*←`(1↑⍴⍵)-`⍺; a domain error occurs if *n*<`0`. For example:

```
    ⊢m←5 4⍴⍳20
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
16 17 18 19

    3 +⌿ m
12 15 18 21
24 27 30 33
36 39 42 45

      2 +⌿ m
 4  6  8 10
12 14 16 18
20 22 24 26
28 30 32 34
```

`n⍀` Rank: `¯ * *` Expand down;

Monad. The expression `v⍀ω` *expands* the argument `ω` along the first axis, inserting at each point corresponding to a zero of `n`, a cell of zeros if `ω` is numeric, a cell of spaces if `ω` consists of characters, and a cell of the elements `<⍳0` if `ω` is boxed or heterogeneous. For example:

```
  1 0 1 0 1⍀2 3 5
2 0 3 0 5

  1 0 1 0 1⍀'abc'
a b c
```

```
  1 0 1 0 1⍀3 4⍴⍳12
0  1  2  3
0  0  0  0
0  0  0  0
4  5  6  7
```

In other words, `n` must be a boolean list, and if `r←n⍀ω`, then `ω≡n⌿r`, and `(~n)⌿r` is an array of zeros, spaces, or boxed empty vectors, of shape `(+/~n),1↑⍴ω`.

`v⍀` Rank: `¯ * *` Scan down;

Monad. The expression `v⍀ω` assembles, along the leading axis, the `1↑⍴ω` results `(0{ω)`, `(v⌿0 1{ω)`, `(v⌿0 1 2{ω)`, etc. For example:

```
    ⊢m←3 5⍴0 1 0 0
0 1 0 0 0
1 0 0 0 1
0 0 0 1 0
```

```
    v⍀m
0 1 0 0 0
1 1 0 0 1
1 1 0 1 1
```

`n⌿ v⌿ n\ v\` Rank: As in `⌿` and `⍀` As in `⌿` and `⍀` with *across* for *down*

The expression `v⌿ω` is equivalent to `v⌿ω` except that the split is made along the last axis rather than the first. In other words `v⌿ω` is equivalent to `v⌿a` , where `a` is obtained from `ω`, by transposing the last axis to the leading position that is, `a←(1⌽⍳0{⍴⍴ω)⍉ω`, or `a←¯1⍤⊢ω`. For example:

```
    ⊢ω←3 4⍴⍳12
 0  1  2  3
 4  5  6  7
 8  9 10 11
```

```
    ⊢a←(1⌽⍳0{⍴⍴ω)⍉ω
0  4  8
1  5  9
2  6 10
3  7 11
```

```
    +⌿ω
6 22 38
```

```
    +⌿a
6 22 38
```

A similar correspondence holds for the other three cases, except that the leading axis of the result (which, in effect, results from the splitting axis) must be returned to last position. For example:

```
    2 0 1 0⌿ω
0  0  2
4  4  6
8  8 10
```

```
    2 0 1 0⌿a
0  4  8
0  4  8
2  6 10
```

```
    1 0⍉2 0 1 0⌿a
0  0  2
4  4  6
8  8 10
```

The relations can be summarized as follows:

```
(n⌿ω)   ≡ ⊢⍤0 n⌿ ¯1⍤⊢ω
(v⌿ω)   ≡     v⌿ ¯1⍤⊢ω
(n\ω)   ≡ ⊢⍤0 n⍀ ¯1⍤⊢ω
(v\ω)   ≡ ⊢⍤0 v⍀ ¯1⍤⊢ω

(⍺n⌿ω)  ≡ ⊢⍤0 ⍺n⌿ ¯1⍤⊢ω
(⍺v⌿ω)  ≡     ⍺v⌿ ¯1⍤⊢ω
```

`v}` Rank: `¯ ¯ ¯` Select; Merge

Monad. The result of `v}ω` is a selection from `ω` of the form `i{ω`, where the index `i` is obtained by applying `v` to `{⍳¨>⍴ω`, the "complete index table" of `ω`. For example:

```
    ⊢ω←2 3⍴⍳18
 0  1  2
 3  4  5
 6  7  8

 9 10 11
12 13 14
15 16 17
```

```
    t←{⍳¨>⍴ω
    ⍴t
2 3 3
```

```
    0{t
┌─────┬─────┬─────┐
│0 0 0│0 0 1│0 0 2│
├─────┼─────┼─────┤
│0 1 0│0 1 1│0 1 2│
├─────┼─────┼─────┤
│0 2 0│0 2 1│0 2 2│
└─────┴─────┴─────┘
```

```
    ⊢i←0 1 1⌽t
┌─────┬─────┬─────┐
│0 0 0│0 1 1│0 2 2│
├─────┼─────┼─────┤
│1 0 0│1 1 1│1 2 2│
└─────┴─────┴─────┘
```

```
    ⊢r←i{ω
0 4 8
```

```
    r≡0 1 1¨⌽}ω
1
```

The foregoing illustrates how the adverb `}` can apply to any selection function (in this case `0 1 1¨⌽`) to produce a selection. Since this is the same selection that could be produced directly as `v⌿ω` (that is, `0 1 1⌽⌿ω`), it is of no interest except as it is used in the dyadic case to produce a merge, as discussed under the dyadic case below.

However, verbs other than selection can serve meaningfully as arguments of `}`. For example:

```
    ⊢q←1¨⌊¨>}ω
0  1  1
3  4  4
3  4  4

 9 10 10
12 13 13
12 13 13
```

```
        ⊢j←1 ⌊¨>t
┌─────┬─────┬─────┐
│0 0 0│0 0 1│0 0 1│
├─────┼─────┼─────┤
│0 1 0│0 1 1│0 1 1│
├─────┼─────┼─────┤
│0 1 0│0 1 1│0 1 1│
└─────┴─────┴─────┘
```

```
   q≡j{⍵                 ┌─────┬─────┬─────┐
1                        │1 0 0│1 0 1│1 0 1│
                         ├─────┼─────┼─────┤
                         │1 1 0│1 1 1│1 1 1│
                         ├─────┼─────┼─────┤
                         │1 1 0│1 1 1│1 1 1│
                         └─────┴─────┴─────┘
```

Dyad. If ⍺ has the same shape as the index array *v*}⍳¨>⍴⍵, then ⍺ *v*} ⍵ produces a *merge* of ⍺ and ⍵, by inserting the elements of ⍺ in the selected positions of ⍵, more specifically, in the positions of ⍵ indexed by the array *v*{⍳¨>⍴⍵. Continuing the example used in the monadic case:

```
    ⊢⍺←2 3⍴100+⍳6       ⊢m←⍺ 0 1 1¨⍀}⍵
100 101 102          100   1   2
103 104 105            3 101   5
                       6   7 102

                     103  10  11
                      12 104  14
                      15  16 105
```

v⊂ Rank: *mv rv lv* ; Swap

Monad. *v*⊂⍵ ↔ ⍵*v*⍵
Dyad. ⍺ *v*⊂ ⍵ ↔ ⍵ *v* ⍺
Dialects. Some dialects [12 13] use ⊂ for a function called enclose; in these, ⊂⍵ is equivalent to <⍵ except that ⍵≡⊂⍵ if ⍵ is an open item.

VI: CONJUNCTIONS

A conjunction produces four main classes of results, referred to by using *m* and *n* for left and right noun arguments, and *u* and *v* for left and right verb arguments. Ranks are shown as for adverbs.

m¨*v* *u*¨*n* Rank: *rv* ⋆ ⋆ *lu* ⋆ ⋆ With/With;

The expression *m*¨*v* ⍵ is defined by *m* *v* ⍵ and *m*¨*v* is therefore a monadic verb resulting from supplying the dyadic case of *v* with a left argument *m*. Similarly, *u*¨*n* ⍵ is defined by ⍵ *u* *n*. For example:

```
    3⍒2¨⍟⍵←1 2 3 4 5 6 7 8
0.000 1.000 1.585 2.000 2.322 2.585 2.807 3.000

    ⋆¨2 ⍵
1 4 9 16 25 36 49 64
```

u¨*v* Rank: *mv mv mv* Under; Under

This function is equivalent to composition (*u*⍥*v*) except that the function inverse to *v* is applied to the result of each cell. For example, since < and > are inverses, as are ⋆ and ⍟:

```
    ⍴⍥>a←'abc'⊃1 3⊃'abcd'      ⍴¨> a
3                              ┌─┬─┬─┐
2                              │3│2│4│
4                              └─┴─┴─┘

    ⍴¨>b←1 2 3⊃''⊃2 3⍴0
┌─┬─┬───┐
│3│0│2 3│
└─┴─┴───┘
    a,¨<b
┌───┬───┐  This result is often called
│abc│1 3│  lamination of a and b,
└───┴───┘  since it combines them
┌─────┬─────┐ along a new initial axis.
│1 2 3│0 0 0│
└─────┴─────┘

    3 +⍥⍟ 4                    3 +¨⍟ 4
 2.48490665                 12
```

The function *u*¨*v* is often called "the dual of *u* with respect to *v*", but the phrase "*u under v*" is probably better, suggesting that *u* is performed after preparatory work by *v*, and before the task is sewn up by reversing the effect of *v*.
The expression *u*¨*v* is valid only if *v* possesses an inverse. The following list shows inverse pairs commonly used in dialects:

```
⋆ + - ÷ ⍟ ⍣ ⍲ ⊢ < ⌹ ~ ≤
⍟ + - ÷ ⍟ ⍣ ⍲ ⊢ > ⌹ ~ ≥
```

m⍥*n* Rank: ⍟3⍴⍟*n* Constant; Constant
See *u*⍥*n* for a discussion of the expression *n*←⍟3⍴⍟*n*.

Monad. The derived verb *m*⍥*n* has rank *n*, and produces a constant individual result *m* for each cell to which it applies. For example:

```
    m←'abc'
    ⍵←2 3⍴⍳6
    m⍥0 ⍵          m⍥1 ⍵          m⍥2 ⍵
abc            abc            abc
abc            abc
abc

abc
abc
abc
```

Dyad. The dyadic case differs only in that agreement of the outer shapes of the arguments is imposed. For example:

```
   ⍺←'PQ'
   ⍺ m⍥0 1 ⍵                ⍺ m⍥0 0 ⍵
abc                      domain error
abc
```

m⍥*v* Rank: ¯ 1/2 ¯ Cut; Cut

Monad. The expression 1⍥*v* ⍵ applies *v* to each of a set of "segments" cut along the first axis, and assembles the results along a leading axis of length equal to the number of segments.

Each segment is of the form `s←(k+⍳n){⍵`, and *k* and *n* are chosen so that each segment begins at the occurrence of the *delimiter* `0{⍵`. In other words, for each segment *s* `(j{s)≡0{⍵` for *j* = 0 and for no other value. For example:

```
   1⍤0<⍵←' worlds on worlds'
┌───────┬───┬───────┐
│ worlds│ on│ worlds│
└───────┴───┴───────┘
```

```
    ⊢m←5 4⍴⍳6
0 1 2 3
4 5 0 1
2 3 4 5
0 1 2 3
4 5 0 1

   1⍤0< m
┌───────┐┌───────┐
│0 1 2 3│|0 1 2 3|
│4 5 0 1│|4 5 0 1|
│2 3 4 5│|_______|
└───────┘

   1⍤0⍴⍵
7
3
7

   1⍤0(+⌿) m
6 9 6 9
4 6 2 4
```

The expression `¯1⍤0 ⍵` differs only in that the delimiters are excluded from the segments. For example:

```
   ¯1⍤0<⍵
┌──────┬──┬──────┐
│worlds│on│worlds│
└──────┴──┴──────┘

   ¯1⍤0⍴⍵
6
2
6

   ¯1⍤0<m
┌───────┐┌───────┐
│4 5 0 1│|4 5 0 1|
│2 3 4 5│|_______|
└───────┘

   ¯1⍤0(+⌿) m
6 8 4 6
4 5 0 1
```

The verbs `2⍤0` and `¯2⍤0` differ from the corresponding cases `1⍤0` and `¯1⍤0` only in that the delimiter is the *last* cell, and marks the end of segments rather than the beginning. For example:

```
   2⍤0<⍵
┌───────┐┌──────────┐
│ worlds ││ on worlds│
└───────┘└──────────┘

   2⍤0< m
┌───────┐┌───────┐
│0 1 2 3│|2 3 4 5|
│4 5 0 1│|0 1 2 3|
└───────┘|_______|
         └───────┘
```

The case `0⍤0 ⍵` applies *v* after reversing *⍵* along each axis, and is equivalent to `(0 1⌽.×⍴⍵) 0⍤0 ⍵`. For a matrix *m*, the expression `0⍤0 ⍵` is equivalent to *v*`⍉`*m*.

The monads `3⍤0` and `¯3⍤0` produce tesselation by "maximal cubes" using size `(⍴⍴⍵)⍴⌊/⍴⍵` and shift `(⍴⍴⍵)⍴1` in the corresponding dyads.

Dyad. For `(⍳k)∊1 2`, the dyadic cases `α k⍤0 ⍵` differ from the corresponding monadic cases only in that the delimiters are the `1`'s in the boolean list `α`. For example:

```
   α←0 1 0 0 1
   α 1⍤0< m
┌───────┐┌───────┐
│4 5 0 1│|4 5 0 1|
│2 3 4 5│|_______|
│0 1 2 3│
└───────┘

   α 2⍤0< m
┌───────┐┌───────┐
│0 1 2 3│|2 3 4 5|
│4 5 0 1│|0 1 2 3|
└───────┘|_______|
         |4 5 0 1|
         └───────┘
```

The case `0⍤0 ⍵` has left rank `2`, and `α 0⍤0 ⍵` applies *v* to a "rectangle" whose beginning point in *⍵* is determined by `0{a`, and whose size is determined by `|1{α`. For example:

```
   a←2 3⍴0 1 1 2 2 3
   b←2 3⍴0 1 1 2 2 ¯3
   ⍵←2 3 5⍴⍳30

   a            b            ⍵
0 1 1        0  1  1      0  1  2  3  4
2 2 3        2  2 ¯3      5  6  7  8  9
                         10 11 12 13 14

   a 0⍤0⊢ ⍵     b 0⍤0⊢ ⍵
 6  7  8      8  7  6     15 16 17 18 19
11 12 13     13 12 11     20 21 22 23 24
                          25 26 27 28 29
21 22 23     23 22 21
26 27 28     28 27 26
```

The beginning point is determined by `(⍴⍵)|(⍴⍴⍵)↑0{α`; in other words, negative indexing may be used, and `0{α` is extended by zeros to provide a full index to *⍵*. The size is determined by *s*`←(⍴s)↓⍴⍵⊣`*s*`←|1{α`. Finally, before application of the verb *v*, the rectangle is reversed along each axis for which `1{α` is negative, as illustrated by the example using *b* above.

The case `3⍤0` also has left rank `2`, and `α3⍤0⍵` applies *v* to each element produced by a tessellation of *⍵*, using a size `1{α`, and beginning points that are multiples of the "shift" `0{α`. For example:

```
   a
abcdef
ghijkl
mnopqr
stuvwx
yzABCD

      (>3 2⊃2 3) 3⍤0<a
┌───┐┌───┐┌──┐
│abc│|cde||ef|
│ghi│|ijk||kl|
└═══┘└═══┘└══┘
│stu||uvw||wx|
│yzA||ABC||CD|
└───┘└───┘└──┘
```

The table of beginning points of the elements of the tessellation is given by:

```
s×⍤¯1>{⍳⍤¯1>⌈(⍴⍵)÷s←0{α
```

The case `¯3⍤0` is equivalent to `3⍤0` except that shards of shape less than `1{α` are omitted.

`u⍤n` Rank: `⌽3⍴⌽n` Rank; Rank

If `n` is a `3`-element list, then `u⍤n` is a monadic function of rank `0{n`, and a dyadic function of left rank `1{n` and right rank `2{n`. In general, `n` is treated as if it were `⌽3⍴⌽n`; in other words a single element specifies all ranks, but if `2=⍴n`, the first element specifies the left rank, and the last specifies the right ranks, both dyadic and monadic.

Monad. A monadic rank of `k` implies that the function applies to `k`-cells of its argument, except that the rank of the cell will not exceed the rank of the argument, as discussed under *degenerate cases* in Section II.B. For example:

```
    ⍵←2 3 4⍴'abcdefghijklmnopqrstuvwx'
    ⍵
abcd
efgh
ijkl

mnop
qrst
uvwx

      ,⍤2 ⍵
abcdefghijkl
mnopqrstuvwx

      ,⍤¯1 ⍵
abcdefghijkl
mnopqrstuvwx

      ,⍤4 ⍵
abcdefghiffklmnopqrstuvwx
```

Dyad. In the expression `⍺ u⍤(l,r) ⍵`, the outer shapes of `⍺` and `⍵` (complementary to the shapes of the `l`-cells and `r`-cells) must agree unless one of them is empty, in which case the single corresponding cell is extended to apply to each of the cells of the other argument. For example, if `⍺←'yz'`, then:

```
    ⍺,⍤0 2 ⍵
yabcd
yefgh
yijkl

zmnop
zqrst
zuvwx

    ⍺,⍤1 1 ⍵
yzabcd
yzefgh
yzijkl

yzmnop
yzqrst
yzuvwx
```

`u⍤v` Rank: `mv mv mv` On; On

Monad. In the simplest case `u⍤v ⍵`, is equivalent to `u v ⍵`. For example:

```
 ⊢y←⍳⍤⌽ ⍵←4 3 2⍴'abcdefghijklmnopqrstuvwx'
bhnt
djpv
flrx

agms
ciou
ekqw

      y≡⍳⌽ ⍵
1
```

However, this relation holds only because the monadic rank of `⌽` is unbounded; more generally, the rank of the derived function `u⍤v` is the rank of `v`; that is, the expression `u v` is applied to each of the *cells* of `⍵` relative to `v`. For example:

```
   ⊢z←⍳⍤2 ⌽⍤2 ⍵
bdf
ace

hjl
gik

npr
moq

tvx
suw

      z≡⍳⌽⍤2 ⍵
0

      z≡⍳⍤(⌽⍤2) ⍵
1
```

Dyad. The left and right ranks of `u⍤v` are both the monadic rank of `v`. Therefore `⍺ u⍤v ⍵` is equivalent to `(v⍺) u v ⍵`.

`m⍤v u⍤n` Rank: `¯ ¯ ¯` Prefer/Defer; Prefer/Defer

The verbs `m⍤v` and `u⍤n` apply the verbs `u` and `v` to their argument or arguments after transpositions to *defer* the axes specified by `n` (in the case of `u⍤n`) to the tail end, or to *prefer* the axes specified by `m` (for `m⍤v`) to the front. For example:

```
    a←2 3 4⍴'abcdefghijklmnopqrstuvwx'
    ⊢b←2 1⍤⊢a
am
eq
iu

bn
fr
jv

co
gs
kw

dp
ht
lx

    2 1⍤,a
ameqiubnfrjvcogskwdphtlx
    (⊢⍤0 a)≡(1 2⍤⊢a)
1
```

These axis movements are prescribed by:

```
PR: (⍋⍺,(⍳0{⍴⍴⍵)~⍺←(⍴⍴⍵)|⍺)⌽⍵
DE: (⍋(⍳0{⍴⍴⍵)~⍺),⍺←(⍴⍴⍵)|⍺)⌽⍵
```

in the following definitions:

```
  m⍤v⍵ ↔ v(>0{a)PR⍵⍪a←⌽3⍴⌽⊃m
  u⍤n⍵ ↔ u(>0{a)DE⍵⍪a←⌽3⍴⌽⊃n
 ⍺m⍤v⍵ ↔ ((>1{a)PR⍵)v(>2{a)PR⍵⍪a←⌽3⍴⌽⊃m
 ⍺u⍤n⍵ ↔ ((>1{a)DE⍵)u(>2{a)DE⍵⍪a←⌽3⍴⌽⊃n
```

Thus, the axes moved (for the cases of a monadic argument, left argument, and right argument, respectively) by `v⍤(1⊃2⊃3)` are `1`, `2`, and `3`, by `v⍤(2⊃3)` are `3`, `2`, and `3`, and by `v⍤1 2 3` or `v⍤(<1`

`2 3)` are `1 2 3`. Compare with the rank adverb ⍤ for use of the phrase `⍵3⍴⍵`.

`u⍥v` Rank: `mv lv rv` Upon; Upon

The monad `u` is applied to the result of `v`, that is:

```
    u⍥v ⍵ ←→ u v ⍵ ←→ u⍤v ⍵
  ⍺u⍥v ⍵ ←→ u ⍺ v ⍵
```

For example:

```
      7 5 3 |⍥- 3 5 7
4 0 4
```

`m.v` Rank: `⋆ ¯ ¯` ; Tie

Dyad. The left argument of the tie adverb specifies the number of outer axes of the arguments that must agree, leaving any remaining axes free to contribute independently to the overall shape of the result, as illustrated by the examples in the discussion of agreement in Section II.B.

If the ranks of `v` are `l` and `r`, and if `z←a t.v b`, then the outer shapes are `osa←(-l)↓⍴a` and `osb←(-r)↓⍴b`. The argument `t` determines a split of these outer shapes into tied and free shapes as follows:

```
ta←t↑osa                    fa←t↓osa
tb←t↑osb                    fb←t↓osb
```

The tied shapes must agree, that is, `ta≡tb`; the overall outer shape of the result `z` is given by `ta,fa,fb`.

If `i` is an index that selects a single cell of `z`, then `⍴>i` must be `⍴ta,fa,fb`. Moreover, each cell of `z` is defined by:

```
(i{z)≡((<(⍴ta,fa)↑>i){a) v
   (<(-⍴ta)⍋(⍴fa)↓(⍴ta)⍋>i){b
```

Dialects. Dialects commonly permit `∘.v` for `0 .v`.

`u.n` Rank: `mu ⋆ ⋆` Ply;

Monad. The function `u` is applied `n` times. For example:

```
      ⍟. 2 ⍵ ←→ ⍟⍟ ⍵
    -⍱. 4 ⍵ ←→ -⍱-⍱-⍱-⍱ ⍵
  1⍤○. 3 ⍵ ←→ 1○1○1○ ⍵
```

The function `u.¯` is the *limit* of the application of `u`; that is, `u.¯⍵` is equivalent to `u.k ⍵`, where `u.k ⍵` matches `u.(k-1)⍵`.
Finally, a negative value of `n` denotes `|n` applications of the inverse function; that is, `u.¯1` is the function inverse to `u`, and `u.(-n)` is inverse to `u.n`, and `u.¯¯` is inverse to `u.¯`.

`u.v` Rank: `2 ¯ ¯` ; Dot product

Monad. The expressions `-.×⍵`, and `+.×⍵` are, for square matrix arguments `⍵`, the *determinant* and the *permanent* of mathematics. More generally, `u.v` is defined in terms of a recursive "expansion by minors", given by the following expression from [17]:

```
(0{⍉⍵) u.v $ (<⍤¯>⍤¯>{0⊃⊂⍳0{⍴⍵){
      ⍵ : 1=¯1{⍴ ⍵ : u≠,⍵
```

Dyad. The expression `⍺+.×⍵` is equivalent to the *dot*, *inner*, or *matrix* product as defined in mathematics for vectors `(+/⍺×⍵)` and matrices (where the element in row `i` of `⍺` and column `j` of `⍺+.×⍵` is the dot product of row `i` of `⍺` and column `j` of `⍵`). For example:

```
   1 2 3 +.× 3 4 5
26

   a←2 3⍴⍳6⋄b←3 4⍴12
   a+.×b
20 23 26 29
56 68 80 92
   a+.×1 2 3
8  26
```

For verbs other than `+` and `×` that share the property of applying to items (i.e., of rank `0`) and producing items, the same definition holds for argument ranks not greater than `2`. For example:

```
   ⍺ ⌊.+ ⍵                    ⍺ +.⌊ ⍵
0 1 2 3                    3  3  3  3
3 4 5 6                    9 10 11 12
```

The general definition for arbitrary functions and arguments of arbitrary rank is:

```
  ⍺ u.v ⍵ ←→ u≠(¯1⍥⊢⍺) 1 .v ⍵
```

In other words, the result is obtained by `u` *over* the result of applying `0 .v` to the major cells of `¯1⍥⊢⍺` and `⍵`. For example:

```
   ⊢c←¯1⍥⊢a                  b
0 3                      0  1  2  3
1 4                      4  5  6  7
2 5                      8  9 10 11

   (0{c)∘.× 0{b          (1{c) ∘.× 1{b
0 0 0 0                  4  5  6  7
0 3 6 9                 16 20 24 28

   (2{c) ∘.× 2{b
16 18 20 22
40 45 50 55
```

The final result of `a+.×b` is the sum over these tables, which agrees with the example of `a+.×b` given earlier. Since `¯1⍥⊢⍺` moves the last axis of `⍺` to the leading position, its overall effect in the definition of `u.v` is to "split `⍺` along the last axis", just as `⍵` is split along the

leading axis. This asymmetric treatment of the arguments rests on the desire to make the simple case of `+.×` on matrix arguments agree with the matrix product of mathematics, whose definition exhibits the same sort of asymmetry.

`u∪n` Rank: Same as `u` Combining rank

The function `u∪n` is equivalent to `u`, except that it has a combining rank of `n` (relevant to the application of the conjunctions `u` and `n` to verbs). The combining rank of all primitive verbs is zero.

`u∪v` Rank: Max of ranks of `u` and `v` Union

Monad. The result of `u∪v` is a "catenation" of `u` and `v` determined by the combining ranks of `u` and `v` (denoted here by `cu` and `cv`) as follows:

```
u∪v ω ↔ (uω),¨<(vω) if cu = cv
      ↔ (uω),(vω)   if cu ≠ cv
```

The combining rank of `u∪v` is `(cu⌈cv)+cu=cv`. For example:

```
   -∪|∪× ¯2 ¯1 0 1
 2 2  ¯1
 1 1  ¯1
 0 0   0
¯1 1   1
```

`u∩v` Rank: Min of ranks of `u` and `v` Intersection

Monad. The combining rank (defined under ∪) is `cu-1`, where `cu` denotes the combining rank of u. Moreover:

```
u∩vω ↔ v≠⍥cu uω
```

For example, `f∪g∩+` and `f∪g∩×` denote what is commonly denoted in mathematics by `f+g` and `f×g`.

`m∇n` Rank: ¯ ¯ ¯ Define; Define

Verb definition. If `m` and `n` are (possibly boxed) character nouns, then `m∇d` yields a verb of unbounded rank whose monadic and dyadic cases are determined by `m` and `d` respectively. For example:

```
   ⊢m←,⊃'ω×⍎(ω≤2)↓''$ω-1'''
┌────────────────┐
│ω×⍎(ω≤2)↓'$ω-1'│
└────────────────┘

   ⊢d←'s←⍴ω'⊃'c↵⍺,ω'⊃'(2,s)⍴c'
┌────┐ ┌─────┐ ┌───────┐
│s←⍴ω│ │c↵⍺,ω│ │(2,s)⍴c│
└────┘ └─────┘ └───────┘

   s←c←ω←2
   r←'abc' m∇d 'def'
   r
abc
def

   s
2

   c
abcdef

   ω
2
```

The major aspects of ∇ can be inferred from this example: the derived function executes by first assigning to ⍺ and ω the values of the arguments, then executing the boxed sentences in `d` in sequence, and then providing as the explicit result of the function the result of the last sentence executed.

The names ⍺, ω, →, and `$` are *local* to the function,and other names may be localized *dynamically*, as discussed in Section II.J.

The monadic case of `m∇d` illustrates the use of the symbol `$` for *self-reference*, that is, reference to the derived function being produced. For example:

```
   m∇d 2          m∇d 3          m∇d 4
2              6              24
```

The sequence of execution of the sentences in evaluating `m∇d ω`, is controlled by a system variable → (local to a function) as follows: → is first assigned the value of `⍳0{⍴m`; sentence `>(0{→){m` is selected for execution, → is respecified by `→←1↓→`, the selected sentence is executed, and the sequence is repeated until → is exhausted, or until a value of `0{→` occurs that is not an index to `m`. Similar remarks apply to the dyadic case. Since → may be respecified within any sentence, any sequence of execution can be achieved. For example, if: `p←'b←1'⊃'→←>(ω≠0){2⊃3 1'⊃'b'⊃'b←(0,b)+b,0'`, then,

```
   p
┌──┐ ┌──────────────┐ ┌─┐ ┌──────────┐
│b←1│ │→←>(ω≠0){2⊃3 1│ │b│ │b←(0,b)+b,0│
└──┘ └──────────────┘ └─┘ └──────────┘


   f←p∇⍥
   f 0            f 1            f 3
1              1 1            1 3 3 1
```

If sentence `k` (that is, `>k{a`) begins with a name followed by a right parenthesis, that name (called a *label*) is localized and assigned the value `k↓⍳⍴a`. Labels are useful in *branching*, that is, in expressions of the form `→←>j{l1⊃l2⊃l3` or `→←l1~l2`.

Finally, if `a` is any argument of ∇, it is treated as `,⊃a`. Consequently, open arguments can be used, as in `'ω*÷2'∇'ω*÷⍺'`.

Adverb definition. The conjunction ∇ accepts certain integer left arguments: in particular `3∇(m⊃d)` produces a function, and is equivalent to `m∇d`. More, generally, `1∇n` yields an adverb, `2∇n` a conjunction, and `3∇n` a verb, provided that `n` is a suitable representation of the corresponing result.

The representation of a conjunction differs from that for a function only in that it may include the names ⍺ and ⍵, which refer to the left and right arguments of the resulting conjunction. Similar remarks apply to an adverb.

For example, `COM←1∇('⍺_⍵'⊃'⍵⍺_⍺')` is a "commute" adverb such that `a F COM b` is equivalent to `b F a`, and `F COM b` is equivalent to `F b`.

Similarly, `TIL←2∇('(⍵⍵)⍺⍵'⊃'(⍵⍵)⍺⍺')` is a conjunction such that if `H←F TIL G`, then a `H b` is eqivalent to `(G b)F a`, and `H b` is equivalent to `(G b)F b`.

Representations. The expression `0∇a` yields a noun which is the *representation* of the argument `a`. If `a` is a verb, adverb, or conjunction defined by `a←k∇(m⊃d)` (for any `k∈1 2 3`), then `0∇a` yields the "standard" representation `(,⊃m),(,⊃n)`.

More generally, the representation of a verb, adverb, or conjunction may comprise more than two elements. For example, if `H←F.G`, then the representation of `H` must incorporate the complete representations of the verbs (or nouns) `F` and `G`. This is necessary to ensure that subsequent reassignments of the names `F` and `G` will not affect the definion of `H`. This is analogous to the fact that the value of the noun `c←a×b` is not affected by subsequent reassignments of the values of `a` and `b`.

For `k∈⍳4`, the expression `k∇r` applies to any suitable representation `r`. Consequently `a` is equivalent to `k∇(0∇a)` for any `a`.

`v:n` Rank: Custom

Most adverbs and conjunctions apply to all functions in a single manner; the *custom* conjunction `:` applies to particular functions in ways defined for the specific function, and its effects are discussed under each function affected. For example, `1 ○:90 a` evaluates the sine of `a` in degrees, that is, in a system having 90 units per quadrant; `⍕:b x` formats `x` with the "position and spacing" determined by the value of `b`.
For each of the relations (`<≤=≥>≠≡`), the custom conjunction specifies the tolerance. For example, `<:1E¯6` is equivalent to `<` used with `⎕ct` set to `1E¯6`.

`u>¨v`[3] Rank: `mv lu mv` Withe

Withe (`u>¨v`) is similar to (`u⍤v`), but applies `v` only to the right argument:

```
⍺ u>¨v ⍵ ←→ ⍺ u v ⍵
  u>¨v ⍵ ←→ ⍵ u v ⍵
```

REFERENCES

[1] International Standards Organization. *Standard for Programming Language APL*, ISO TC97/SC5 WG 6 N33, 1986.

[2] Falkoff, A.D., and K.E. Iverson, "The Design of APL", *IBM Journal of Research and Development*, Vol.17, No. 4, July 1973. (Republished in [16]).

[3] Falkoff, A.D., and K.E. Iverson, *The Evolution of APL*, SIGPLAN Notices 13, ACM, August 1978. (Republished in [16]).

[4] *The American Heritage Dictionary o/'the English Language*, Houghton Mifflin Company.

[5] Berry, P.C., *SHARP APL Pocket Reference*, Toronto, I.P. Sharp Associates, 1984. Publication 0199 8409 E3 R1.

[6] Iverson, K.E., *Elementary Analysis*, APL Press, 1976.

[7] Falkoff, A.D. and K.E. Iverson, *APL Language* (IBM GC26-3847).

[8] Berry, P. C., *SHARP APL Reference Manual*, (with Additions and Corrections). Toronto, l.E Sharp Associates, 1981. ISBN 0 86493 0011.

[9] McDonnell E. E., "Zero Divided by Zero", New York: Association for Computing Machinery, *Proceedings of APL76*, pp. 295-302.

[10] Gellert, W., et al, *The VNR Concise Encyclopedia of Mathematics*, Van Nostrand Reinhold.

[11] McDonnell, E. E., "Complex Floor", *APL Congress 73*, North Holland Publishing Company.

[12] Rabenhorst, D.A., *APL2 IUP Manual*, IBM Corporation.

[13] Cheney, C., *NARS Manual*, STSC Inc.

[14] Wooster, P.K., *Extended Upgrade and Downgrade*, SHARP APL Technical Note 41, I.R Sharp Associates.

[15] Smith, H. J., "Sorting - a New/Old Problem", *APL79 Conference Proceedings*, APL Quote Quad, ACM.

[16] McDonnell, E. E., Editor, *A Source Book in APL*, APL Press, 1981.

[17] Hui, Roger, "Some uses of { and }", APL87.

[3] The two symbols `>¨` actually represent a single overstrike character.

↑	*Pike*	@I^	(	*Left Paren*	(
→		@->	)	*Right Paren*	)
↓	*Spike*	@Iv	{	*Left Brace*	{
←		@<-	}	*Right Brace*	}
⊤		@T	◇	*Diamond*	@<>
⊣		@-I	∘	*Jot*	@o
⊥	*Base*	@@T	○	*Circle*	@O
⊢		@I-	÷		@-:
∪	*Cup*	@u	α	*Alpha*	@a
⊂		@c	∈	*Epsilon*	@e
∩	*Cap*	@n	⍳	*Iota*	@i
⊃		@@c	⍴	*Rho*	@r
∨		@v	⍵	*Omega*	@w
<		<	∆	*Delta*	@D
∧	*Caret*	^	∇	*Del*	@@D
>		>	⍞	*Quad*	@[]
≥		@>_	⍝	*Lamp*	@no
≤		@<_	⌸	*Domino*	@[-:]
[	*Left Bracket*	[	⍤	*Paw*	@o"
]	*Right Bracket*	]	⍥	*Hoof*	@O"
⌊	*Downstile*	@L	⍋		@OI
⌈	*Upstile*	@@L	⍟		@0-
|	*Stile*	@I	⍀		@O\
/	*Slash*	/	⍣		@O*
-	*Bar*	-	⍒		@V~
\		\	⍲		@^~
+	*Greek Cross*	+	⍎	*Tack*	@@To
×	*Cross*	@x	⍕	*Thorn*	@To
=		=	⍏	*Pine*	@DI
≠		@=/	⍖	*Spine*	@@DI
*	*Star*	*	⍁		@/-
≡		@=_	⍂		@\-
,	*Comma*	,	⍗		@@D~
;	*Semicolon*	;	⍞		@[']
.	*Period*	.	⍨		@,-
:	*Colon*	:	↵		@<-|
?	*Query*	?	⍺		@a_
'	*Quote*	'	⍷		@e_
¨	*Dieresis*	"	⍳		@i_
!	*Exclamation*	!	⍵		@w_
¯	*Macron*	@-	>¨	*Withe*	@>"
~	*Tilde*	~			
$	*Dollar*	$			
_	*Underscore*	_			

The ASCII [5] transliteration scheme in the last column is based upon *similarity*, English-Greek *correspondences*, and *variants*, denoted by an extra delimiter (@) and varying by rotation about a horizontal or vertical axis. Each transliteration which begins with a delimiter must end with a space.

Table 1: APL ALPHABET AND ASCII TRANSLITERATION

```
parse l;names;r;u
names←'ABCDE'¬⍷'st',(0=⎕nc 'st')/'←2 8⍴''⍵1+/.←()'''
r←4⍴' '¬l←'⍵',(l≠' ')/l
l1:l,'⋄',r
u←∧/(cases∈<'')∨((⍴ cases)⍴4↑r)∈ >cases
→l1×⍷,(<\u)⌿actions

move:s¬l ←¯1↓l¬r ←(⍵eval(-1⌊s←0≠⍴l)↑l),r→⎕←(-⍴l)↑⍵
eval:⍵:⍺='→':st[1;st[0;]⍳,⍵]

noun:1¬r ←'1' f ⍵
verb:1¬r ←'+' f ⍵
punc:1¬r ←r[1] f ⍵
is:1→r ←t f⍵¬st ←((1↑s⌽q,¯1↑(s←'1'=q←r[0])⌽names),t←r[2]),st
f:(k↑r),⍺,((k←u⍳1)↓(~u←⍵='↓')/4↑r)4↓r¬⎕←((1+⍴l)⍴' '),⍵
```

The *parse* function is controlled by a table of cases, and a table of actions:

```
      cases                                    actions
⍵/+←(         +         1                    noun ' ↓↓ '
.             1         +        1           noun '  ↓↓'
⍵1/+←(        1         +        1           noun ' ↓↓↓'
⍵1/+←(        1+        /                    verb ' ↓↓ '
⍵1/+←(        1+        .        1+          verb ' ↓↓↓'
1abcde        ←         1+/.                 is   '↓↓↓ '
(             1/.+      )                    punc '↓↓↓ '
←                                            move '⋄'
                                             move '→'
```

For example, when the right stack is `(+1)+1`, the case in the leading row is satisfied because `(` belongs to the (boxed) list in the leading element, `+` to the next, `1` to the next, and `)` to the last (since an empty list indicates that anything is accepted). The corresponding action (noun `' ↓↓ '`) is therefore executed. It replaces the elements indicated by the arrows by a single `1`, which represents the noun that results from the execution of `+1`.

It should be noted that the *earliest* eligible case is chosen; the moves (from left stack to right stack) in the last two cases therefore occur only if nothing else is possible.

The noun *names* contains the capital letters *A*, *B*, *C*, *D*, *E* used as pronouns in assignments of the form `(1)←1`, a case recognized because of the `1` included in the first element of the seventh row of *cases* (that is, `6 { cases`). Name assignments made by previous uses of *parse* can be expunged by expunging the symbol table *st*, that is, by entering `⎕ex 'st'`.

A version of *parse* that uses only facilities available in all dialects may be obtained by replacing its last two lines by a single line:

```
→l1×⍷actions[(∧/ 0 1 1 ⍉cases ∨.∧'abcde⍵ ()←./+1'∘.=4↑r)⍳1;]
```

and replacing *cases* by the boolean array *cases*←⍉(14⍴2)⊤*n*, where *n* is:

```
  342       8     343       343      343    15873        64       16    16383
    2       1       1         3        3       16        15    16383    16383
    1       2       2         4        8       15        32    16383    16383
16383       1       1     16383        3    16383     16383    16383    16383
```

Table 2: PARSING PROCESS

```
z←translate a;b;d;l;m;nq;t
z← 0 0 ρd←'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ⎕0i23456789'
nq←t\b←(t←=\''''≠a)/a
t←('←'=1⊖b)>b∈']⎕⍞ αω'
d←(¯1⊖l∈t/l←+\~b∈d)/b
d[t/⍳ρt←'←'=d]←' '
d←,';',(∨≠<\d∧.=⍉d)≠d←(+/∧\d=' ')⊖d←(-+\t)⊖((+/t),ρd)↑(1,ρd)ρd
b[t/⍳ρt←':'=t\(t←=\''''≠b)/b←a]←' '
→(∧/ 3 5 ≠m←1+l←1++/t)/0
b←(-+\t)⊖(1,ρb)↑(1,ρb)ρb
b←b,[⎕io](1+ρb)↑a
→(3=m)/l1
b←b[⎕io+ 0 2 1 3 4 ;]
l1:z←(1⊖template[(⎕io+2⌊2⊥'αω'∈nq),1+m+⍳m-1;],b),(m,ρd)↑(1,ρd)ρd

    template
 ω←
ωω←
ωω←α


 ω←

   ⍝
 →2+0≠
 →0⍱ω←
 →0⍱ω←
   ⍝

Δ;α;ω
(∨/α≠(ρα)↑⎕fx translate ω)/'not done',0ρ⎕ex α←(+/∧\ω≠':')↑ω←⍞
```

For example:

```
    Δ                        Δ
f: α+÷ω                  fib: z,+/¯2↑z←fib ω-1 : ω=1 : 1
    3 f 4                    fib 10
3.25                     1 1 2 3 5 8 13 21  34 55
```

The function Δ can be modified so that if a function name alone is entered it will first display the definition of the function (in direct definition form), then allow revision using the editing facilities of the particular computer in use, and then fix the definition of the revised function.

Table 3: TRANSLATION TO CANONICAL FORM

⊤	Antibase	Representation, Encode	≥	Mix	Permutation from cycles
≥	Aft	Is greater than or equal to, Not less than	⌈	Mix	Permute
>	After	Is greater than	∪	Mix index	Permutation index
{	All	Catalogue, Generalized cartesian product	⍲	Nand	Not and
∧	And	Least common multiple, And (on booleans)	⍱	Nor	Not or
⊥	Base	Base value, Decode	~	Not	
<	Before	Is less than	↑	Nub	Distinct elements of
<	Box	Enclose	=	Nub in	
⊃	Box open	Conditional enclose	≠	Nubsieve	
,	By	Catenate along last axis	>	Open	Disclose
⍉	Cant	Transpose	∨	Or	Greatest common divisor, Or (on booleans)
⌈	Ceiling		!	Out or	Binomial coefficients; No. of combinations
○	Circle	See definitions	⍥	Over	Catenate along leading axis
⍳	Count	Integers to	÷	Per	Reciprocal; Divided by, Over
≤	Cycle		○	Pi	Pi times
?	Deal	Pseudo-random draws without replacement	+	Plus	Add, Added to
⍒	Downgrade		*	Power	Exponential, e^{ω}, Antilog; To the power, Antilog
↓	Drop		,	Ravel	
=	Equal	Is equal to, Equals	↓	Raze	
⍎	Execute		∈	Raze in	
!	Factorial		⍴	Reshape	
⌊	Floor	Integer part of	|	Residue	Remainder, Modulo
≤	Fore	Is less than or equal to, Not greater than	⌽	Reverse	
⍕	Format		⊢	Right	Identity; Dex
{	From		?	Roll	Pseudo-random choice
⍋	Grade		⌽	Rotate	Rotate along last axis
∈	In	Is a member of, Belongs to	⊖	Rowel	Rotate along leading axis
⍷	In	String search	⍴	Shape	
⍳	Index	Index of	|	Size	Absolute value, Magnitude
⍸	Index	String search index	⍥	Table	
⌹	Inverse	Matrix inverse; Matrix division	↑	Take	
⊣	Left	Lev	×	Times	Multiplied by
~	Less	Set difference	×	Trend	Sign, Signum, Direction, Direction cosines
⊃	Link		≠	Unequal	Is unequal to
⍟	Log	Natural logarithm; Base-e log	⊖	Upset	Reverse on leading axis
≡	Match				
+	Mate	Conjugate			
⌈	Maximum				
⌊	Minimum				
-	Minus	Negative, subtract, less			

Table 4: STANDARD NAMES AND SYNONYMS