Volume 1. Language Reference

Contents

1. Overview of A+

Summary of the A+ Programming Language

A+ is an array-oriented programming language that provides

- a rich set of primitive functions that act efficiently on arrays
- general operators that control the ways functions are applied to arrays.

In A+, the ordinary concept of arrays has been enhanced to provide

- a means by which files can be treated as ordinary arrays
- a variety of simple, straightforward ways of displaying and editing data on a screen, with automatic synchronization between displayed data and the contents of variables
- generalized, spreadsheet-like interactions among variables.

These features are realized in A+ by furnishing global variables with

- the attribute of being specific to one A+ process or more generally accessible
- visual attributes such as font and color, analogous to the ordinary attributes of shape and type
- asynchronous execution of functions that have been associated with variables and events
- definitions describing the spreadsheet-like relations among their values.

Global variables with associated definitions involving other global variables are called *dependencies*. The values for an interrelated set of dependencies are automatically kept as current as needed: if an object changes, any variable that is dependent upon it is recalculated just before the next use of that dependent variable. Although these spreadsheet-like relations are not new in principle, the definitions on which they are based can employ the full A+ programming language. In particular, the spreadsheet concept of a cell is not restricted to a scalar in A+, but can be any array, so that much more data can be managed by these relations than is usual for spreadsheets, and more efficiently. Similarly, the spreadsheet paradigm is not limited to numeric relations. For example, the concept of a view in a relational database can be realized as a dependency on the source data.

Other A+ features are

- conventional control structures
- contexts, or separate namespaces, in a single workspace
- dynamic linking of C functions, which can be called like ordinary A+ defined functions
- a Unix operating system environment
- an Emacs-based application development environment
- asynchronous communication between processes, based on A+ arrays.

Some Features of the A+ Language

The primitive functions of A+, a variant of APL, can be classified as scalar, structural, or specialized. A scalar primitive is applied independently to the individual elements of its array arguments, but syntactically it can be applied to an entire array, providing a very efficient implicit control structure. The scalar primitives include the ordinary arithmetic functions, comparison functions, logical functions, and certain other mathematical functions. A structural primitive is one that can be defined completely in terms of the indices of its right argument: it rearranges or selects the elements of that argument but otherwise leaves them unmodified. The specialized primitive functions include, for example, ones for sorting arrays and inverting matrices.

Leading Axis Operations

Most A+ structural primitive functions, and functions derived from the operators called Reduce and Scan, apply to the leading axis of the right argument (cf. "The Structure of Data"). These structural A+ primitives are Catenate, Take, Drop, Reverse, Rotate, Replicate, and Expand. The subarrays obtained by

- specifying only the leading axis index, and
- specifying it to be a single, scalar value

are called the *items* of the array. Another way to say that a structural function applies to the leading axis is to say that it rearranges the items, but not the elements within the items.

Rank Operator

The concepts of leading axis and item are generalized by treating an array as a *frame* defined by the array's leading m axes holding *cells* of rank n defined by the array's trailing n axes, where m+n is the rank of the array. A function `f` is applied to all cells of rank n with the expression `f@n`. The rank operator (`@`) applies uniformly to all functions of one or two arguments: primitive, derived, or defined, except Assignment and (because of its syntax) Bracket Indexing (`@` does apply to Choose, which is semantically equivalent).

Mapped Files

Mapped files are files accessed as simple arrays. These files can be very large (currently of the order of a gigabyte). Only the parts of a file that are actually referenced are brought into real memory, and therefore operations that deal only with parts of files are particularly efficient. Unless the files are extremely large, the transition from application prototypes dealing with ordinary arrays to production applications using mapped files requires only minimal code modification.

Screens and Workspaces

Screens show views of arrays. A workspace is where computation takes place and where all immediately available A+ objects reside - variables, functions, operators, and so on. An array can be displayed on a screen with the *show* function. The array in the workspace and the screen view share the same storage. Changes to the array in the workspace are immediately reflected on the screen. Changes can be made to the screen view of the array, and they immediately change the array in the workspace. (The word workspace is also used in a different sense in screen management, to denote the leading top-level window.)

Callbacks

Callbacks are automatic invocations of functions that have been associated with variables and events. Specification of a variable or selection of a row in its screen display, for example, can trigger the execution of a callback function. Callbacks provide a complete means of responding to asynchronous events.

Dependencies

Dependencies are global variables with associated definitions. When a dependent variable is referenced its definition will be evaluated if, generally speaking, any objects referenced in the definition have changed since its last evaluation. Otherwise, its stored value is returned. Thus dependencies, like functions, always use the current values of the objects they reference, but are more efficient than functions because they do not reevaluate their definitions until at least one referenced object has changed.

Contexts

Utilities and toolkits can be included in applications without name conflicts, by using contexts, which allow utility packages and toolkits to have their own private sets of names. Outside a context, names within are referred to by qualifying them with the context name. System commands and system functions provide facilities for working with contexts, such as changing the current context and listing all contexts in the workspace.

Some Features of the A+ System

Linux, Solaris, and AIX Environments

A+ operates under various forms of Unix (Linux, Solaris, AIX, ...). A+ processes can be started from a shell or an Emacs or XEmacs session. Hereafter, "Emacs" is used here in a general sense, to mean Emacs or Xemacs. Emacs provides the application development environment for A+. It is possible to work in desk calculator mode in an A+ process started under Emacs. In this mode the user's view of the A+ process is an interactive session, where expressions can be entered for evaluation, and results are displayed. A session log is maintained, and can be referenced during the A+ session and saved at the end for future reference. Desk calculator mode is also the default for an A+ session started in a shell, but it is more common to use these A+ sessions for running applications with desk calculator mode turned off. In case an application fails, the appropriate entries to a log file can be written, or the A+ process can be permitted to return to desk calculator mode for debugging. An A+ process can communicate with other processes, A+ or not A+, through a communications interface called *adap* (see "Interprocess Communication: adap").

Emacs Programming Development Environment

The A+ mode in Emacs provides programmers with very effective ways of testing and debugging applications. Programmers usually work with two visible buffers, one containing an A+ process and the other the source script of an application. Function keys provide the means to move either a single line from the script to the A+ process, where it is automatically executed, or an entire function definition. It is also possible to scroll back in the session log to bring expressions and function definitions forward for editing and reevaluation.

Applications

Programmers are concerned with three things when writing A+ applications: data, analytics (i.e., computations), and the user interface. The data of interest either reside in files accessible to the application or are maintained by another process. The analytics are the computations run on the data, and the user interface is the means for presenting the data or various aspects of the analytics to users. A+ has been designed for efficient programming of all three aspects of application production, and for efficient execution as well.

Data in files are usually maintained in A+ as so-called *mapped files* (see "Files in A+"), which are simple (i.e., not enclosed, or nested) arrays. Once an A+ application has opened a mapped file, it deals with it much as it would an ordinary array of its own creation. Mapped files can be shared, although shared updates across the network are problematical, unless mediated by a single process. Unix text files can also be copied into and written out of A+ processes.

Real-time data, which is of the utmost importance to many A+ applications, is accessed through an interprocess communication toolkit called *adap*. This toolkit provides a small number of functions for establishing and maintaining communication between an A+ process and other processes, such as a set of real-time data managers that read and write A+ arrays.

As an array-oriented language with a set of primitive functions that apply directly to entire arrays, A+ provides very efficient processing of functions that apply to large collections of data, often with less code than more conventional programming languages. Less code generally means fewer chances for failure; moreover, the A+ language processor is interpretive, which makes debugging relatively easy. Unless you take advantage of array calculations, however, being in an interpretive environment is likely to hurt you in performance. Thinking in terms of array algorithms is both a requirement and an opportunity, and it differentiates development in APL-derived environments from development in most other environments.

Application user interfaces are built with the A+ screen management system, a toolkit that relies on a small number of functions to create and interact with a variety of screen objects, such as buttons, tables, and layouts. See the chapters on screen management, display classes, and display attributes.

Script Files

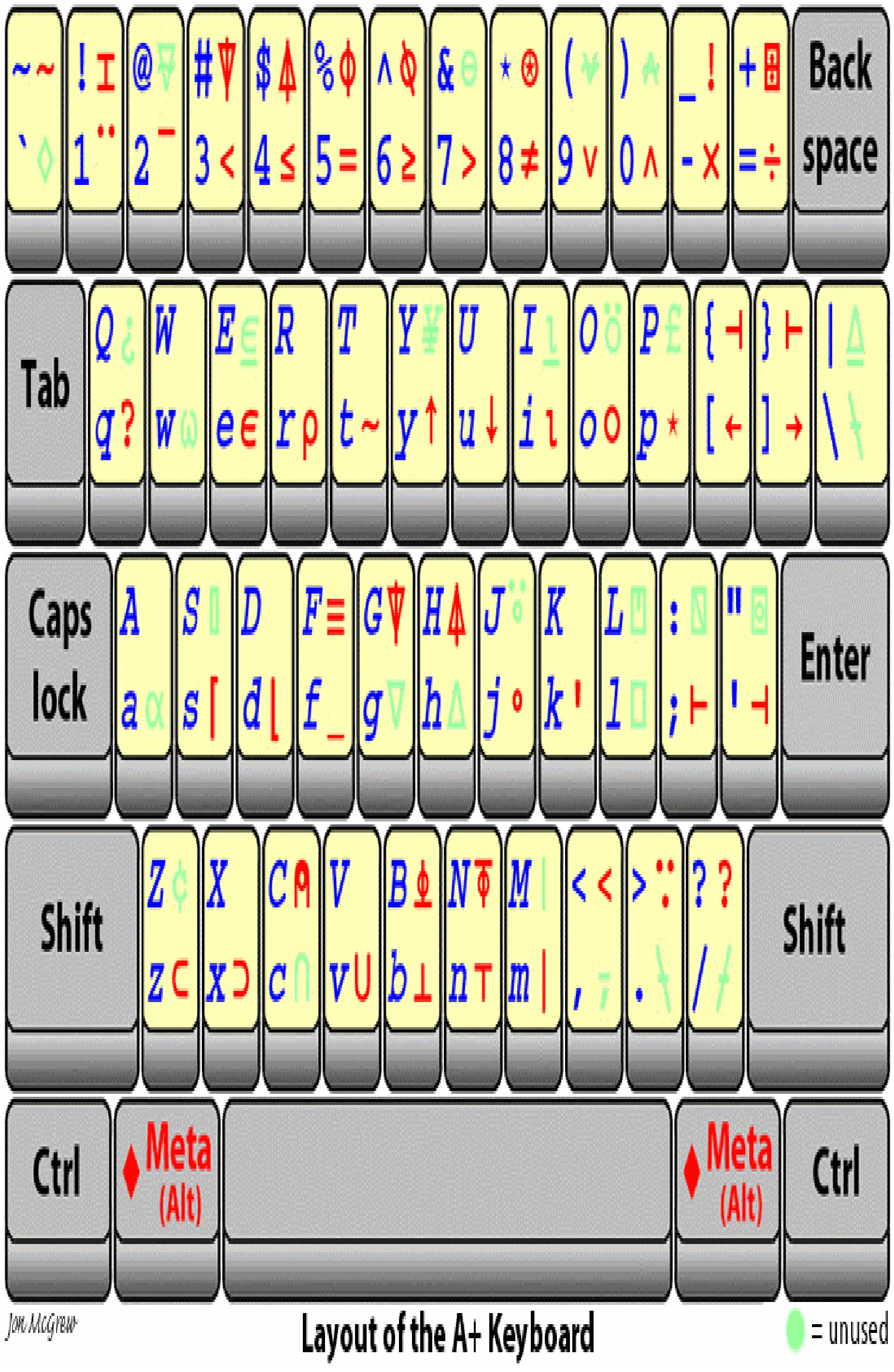
Applications are maintained in text files called scripts. Scripts contain function and data definitions and executable expressions. A+ has specific facilities for loading scripts. Loading a script has much the same effect as entering the lines of the script one at a time in desk calculator mode, starting at the top. Scripts can contain the A+ expressions for loading other scripts. Consequently application scripts do not have to contain copies of utilities and toolkits, and A+ applications tend to be very modular.

The A+ Keyboard

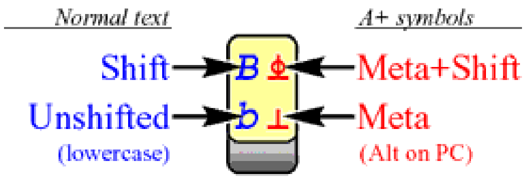
A+ uses the APL Union Keyboard. The special APL characters are entered by pressing a key while pressing either the **Meta** key, or both the **Meta** and **Shift** keys. The **Meta** keys (on Sun keyboards) are on either side of the **space** bar and are marked with diamonds. IBM keyboards have no **Meta** keys; use the **Alt** key, similarly situated, instead of **Meta**.

Note: **Meta-Shift-m** looks like **Meta-m** and **Shift-** but does not represent an A+ function.

An Interactive Keyboard Chart is also available, providing additional information about each of the A+ symbols and functions.



Layout of the A+ Keyboard



Click here for a keyboard diagram which will print more clearly on a black-and-white laser printer.

This figure is available as a printed *Keyboard Reference Chart Tentcard*, which is designed to stand up on your desktop beside your workstation. To request a copy, contact doc@aplusdev.org.

2. The Structure of Data

In this chapter, the concepts of A+ data and the vocabulary used in describing them are discussed first. Then some A+ primitive functions for creating and indexing arrays and for inquiring into their characteristics are introduced. In these sections a number of examples of arrays are given. Finally, certain classes of arrays which are useful in the description of A+ functions are treated.

Concepts and Terminology

The data objects in A+ are *arrays*, which can be visualized as rectilinear arrangements of individual values. An individual value in an array is called an *element*. In the simplest arrays, the elements are either all numbers or all characters. A number or a character is itself an array, of the most elementary kind.

In the rectangular visualization of an array, each set of parallel edges defines a direction. Corresponding to each of these directions is an *axis*. The axes of an array are ordered. In the visualization of an array with three axes, the first axis is directed away from the viewer, the second is directed downward, and the third is directed to the right. A two-dimensional display of an array with three axes shows it as a series of planes arranged vertically, representing cross sections perpendicular to the first axis. The term *leading axes* is used for any set composed of all the axes from the first up to some particular axis, inclusive, and *trailing axes* for any set composed of all the axes from some particular axis through the last one.

An array with no axes, necessarily consisting of a single element, is called a *scalar*. All elements of arrays are scalars. Arrays with one axis are called *vectors* or *lists*, or, if character, *strings*. Arrays with two axes are called *matrices*, and sometimes *tables*. A set of elements lying along, i.e., parallel to, the first axis of a matrix is called a *column*, and a set along the second axis a *row*, just the same as in ordinary usage for tables. These terms are also used for elements along the two trailing axes of arrays with more than two axes.

Dimension, Shape, and Rank

The *length of an axis* is the number of elements lying along any one of the edges defining that axis. This length is also called a *dimension*, so an array has as many dimensions as axes. (The word dimension is sometimes used as a synonym for the word axis, but not in this manual.) The vector composed of the lengths of all axes of an array, i.e., the vector of dimensions, is called the *shape* of the array. The ordering of the dimensions in the shape is the same as the ordering of the axes to which they correspond. The total number of elements in an array can be found by multiplying together all the elements of its shape.

An array can be *empty*, that is, it can have no elements at all. An empty array can have any number of axes except zero, which is disallowed, essentially because you can't have an empty container without a container. At least one of the dimensions of an empty array is equal to zero.

The *rank* of an array is the number of its axes, and therefore it is also the number of elements in its shape, i.e., the length of its shape. A scalar has an empty shape - its shape is a vector that has no elements - and its rank is 0. (Incidentally, when all the elements of an empty vector are multiplied together the result is 1, by convention, so that the usual formulation for the number of elements in an array `a` - namely, `×/ρa` - works for scalars also.)

Every element of an array can be referenced by a set of coordinates called *indices*, to retrieve the value of the element or to give it a new value. There is one index, or coordinate, for each axis, and A+ defines its value to be an integer between zero and one less than the length of that axis, inclusive. The number of indices of an element in an array, then, equals the rank of the array.

Some computational languages use the word *cell* as a synonym for element, but A+ does not (except in connection with the displays created by s, the screen management system): *cell* is used in connection with the partitioning of an array, as defined by a set of leading axes. In practice, multidimensional arrays are commonly viewed as partitioned into collections of lower dimensional arrays. For example, a numeric matrix containing bond prices may be organized so that the rows are time series of prices for bonds, with one row for each bond of interest, while the columns are collections of prices at particular times. For some calculations the rows would be emphasized, while for others, emphasis would be on the columns. One view represents a partition of the matrix into a collection of row vectors, and the other into column vectors.

A+ emphasizes partitions where the lower dimensional arrays lie along a set of trailing axes. The lower dimensional arrays that comprise such a partition are called cells. The complementary set of leading axes is called the *frame* of the partition that *holds* the cells; the cells are said to be *in their frame*. In the case of the numeric matrix of bond prices, the row vectors are the cells of rank 1, and the first axis is their frame.

Every set of leading axes defines a partition into cells for which it is the frame. The set of all axes is a particular set of leading axes, and therefore defines a partition. Since there are no axes left for the cells, the cells must be the elements of the array; the A+ notion of cell, then, includes the more common one. At the other extreme, the array itself is a cell, i.e., a partition of itself into one subarray. In this case the cell takes all the axes and therefore the frame has no axes.

A cell consists of all those elements that have one particular set of indices for the leading axes that define the partition, and all possible indices for the trailing axes. The entire cell can be selected by specifying only the particular indices for the leading axes. Those leading axes are the frame of the partition, and therefore the frame is, loosely speaking, an array of cells that can be indexed by valid indices of them. A partition creates, then, a view of an array as a frame of cells. There is more about frames and cells, including several examples, later in this chapter. The "Dyadic Operators" chapter, and especially its "Rank Deriving Dyadic" section, has a further discussion of this subject, with examples.

One partition plays a special role in A+, the one defined by the first axis alone; the cells for this partition are called the *items* of an array. Every array can be regarded as a vector of items, and many A+ functions look at them in just that way. In such a context, a scalar is regarded as having a single item, namely itself.

Type and Nesting

Another characteristic of arrays is type. In a simple array (definition later), all elements have the same type, but a nonsimple array can contain elements of several different types.

The most common simple arrays are numeric and character. Every element of a simple numeric array is a number, and every element of a simple *character* array is a character. Numeric arrays can be of either *integer* or *floating-point* type. These two types correspond to whole numbers and fractional (sometimes called decimal) numbers. A+ numeric primitive functions applied to integer arrays may automatically convert their arguments to floating point, like the Matrix Inverse function, or may attempt to produce an integer result, like Add and Subtract. If an overflow occurs during this attempt, the type of the result is changed to floating point.

The type of a simple array may also be *symbol* or *function* if it is nonempty, or *null* if it is empty. A symbol is a character string represented as a single scalar; it is denoted by a backquote followed by the string, as in `` `sym``. A function expression, e.g., `÷` or `+.×`, and a function scalar, e.g., `<{-}`, both have type function.

While the elements of arrays are often just individual numbers and characters, an element of an array can be an encapsulated multielement array. That is, any array can be *enclosed* to become a scalar, and this scalar can be an element of another array. Also, any enclosed array, except a function scalar, can be *disclosed*, in order to work with its contents. (A function scalar is an enclosed function expression. The operator Apply, given a function scalar, produces the underlying function expression.) An array that has an enclosed element other than a function scalar is called *nested*, and one that has no enclosed elements except function scalars is called *simple*. A function scalar is simple, but an enclosed function scalar is nested. Any nested array is necessarily nonempty, being or containing a scalar.

A simple scalar symbol or function scalar can be an element of a nested array. In order for data whose type is character, integer, floating point, function, or null to appear in a nested array, however, it must first be enclosed. Clearly, any nonscalar array must be enclosed before being inserted as an element in another array, since the elements of all arrays must be scalars.

When an array other than a function expression is enclosed, the resulting array is a scalar of type *box*. The type of a nonscalar nested array is the type of its first item. Since a nested array can contain elements whose types are box, symbol, and function, its type can be any one of these three. The disclosure of a box scalar, of course, can yield an array of any type.

Any empty array is simple, because if it were nested, it would contain an enclosed array. An empty array that is reshaped or selected from a character, integer, or floating-point array is of the same type. Empty arrays of these three types can also be produced by explicit type transformations from empty arrays of these types. The type of an empty array of symbols, functions, nulls, or boxes is null. The empty vector whose type is null is called Null or the Null; it can be represented as `()`.

There is also a type called *unknown*, to guard against weird cases that might arise. It will not be mentioned further, except in the description of the Type function.

Creating Arrays

A+ provides direct ways to specify constant arrays. A list of numbers separated by blank spaces is one description of a simple constant numeric array. For example, the constant

```
10 2.3e-2 34.156
```

is a floating-point array with one axis, of length three. The element at index 0 is 10, at index 1 is .023, and at 2 is 34.156. The expression with `e` means take the number on the left and multiply it by ten to the power shown on the right. If you omit the blanks between numbers - a poor idea indeed, since it would make your code very difficult to read - , A+ will give you a numeric vector, but probably not the one you intended. If a number is being parsed and a character is examined that can't be part of the number, then a new number is started if the character could begin a number. For instance,

`1e-3.5 40.358.62.7` is read by A+ as `0.001 0.5 40.358 0.62 0.7`

Simple symbol vectors can be written similarly, and blanks are not needed. One of length five is

```
`sym1 `sym2 `sym3`sym4`sym5
```

It is also easy to describe simple constant character vectors. For example,

```
'axrTVw'
```

is a character array with one axis, of length six. The elements at indices 0, 1, 2, 3, 4, and 5 are, respectively, `'a'`, `'x'`, `'r'`, `'T'`, `'V'`, and `'w'`. The empty character vector can be written most easily as `''` - just two quotation marks, with nothing between them.

A nested vector can be described conveniently by a *strand*, a parenthesized expression in which the vector's elements are separated by semicolons. Enclosure of each element is implied by strand notation. For example,

```
(`sym; +; 1 2 3 4; 1.7 3.14; 'example';)
```

is a nested vector of length six. The blanks after the semicolons are not required, but usually promote readability. All of its elements except the second are of type box; the second is a simple function scalar. The types (lengths) of its elements when each is disclosed are: symbol (a scalar), function (a scalar), integer (4), floating point (2), character (7), and null (0). The absence of an expression in any position of the strand implies a Null.

Arrays with more than one axis can be formed using the dyadic primitive function called Reshape and denoted by `ρ` (rho). For example, the result of the expression

```
     2 3ρ'axrTVw'  ⍝ Enter this in an A+ session, and press Enter.
axr                ⍝ This row and the next display the result.
TVw                ⍝ Text following "⍝" is a comment.
```

is an array with two axes - a matrix. The left argument of Reshape in this example is a vector, specifying the shape of the result. The index of an element in the matrix is a pair consisting of one index for axis 0, and one for axis 1. For instance, the element `'r'` is indexed by the pair 0, 2.

The monadic primitive function called Interval and denoted by `ι` (iota) is somewhat like Reshape. It creates arrays of any specified shape whose elements are the integers 0, 1, ... For example,

```
     ι17           ⍝ Simple vectors are always displayed horizontally
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

is an array with one axis. Note that this array has 17 elements, and the index of the i-th element is i for every i from 0 to 16.

The Interval primitive can also create arrays with more than one axis. For example:

```
     ⍳2 3 5        ⍝ Enter this in an A+ session, and press Enter.
  0  1  2  3  4    ⍝ These seven rows (one blank) show the result,
  5  6  7  8  9    ⍝ which is equal to 2 3 5 ⍴ ⍳2×3×5
 10 11 12 13 14    ⍝ (which could be written 2 3 5⍴⍳2×3×5).
                   ⍝ A blank line separates planes. If there were a fourth
axis,
 15 16 17 18 19    ⍝ two blank lines would separate subarrays corresponding to
 20 21 22 23 24    ⍝ indices along the first axis, with single blank lines
between
 25 26 27 28 29    ⍝ subarrays corresponding to indices along the second axis.
```

The empty integer vector is most easily written `⍳0` and the empty floating-point vector `0⍴0.` (decimal point). In displays, all empty arrays occupy one (blank) line, except the Null, which occupies no display space at all.

The function Enclose, denoted by `<`, is used to enclose arrays; `<` is used also to indicate enclosure in displays:

```
     <⍳2 5         ⍝ Much like the previous example, but an enclosed scalar.
<   0 1 2 3 4      ⍝ The   < is used to indicate enclosure.
    5 6 7 8 9      ⍝ < is displayed only at the start of each enclosed array.
```

Strand notation can be combined with Enclose:

```
     (1 2 3;<1 2 3;'abc';+;`smbl;) ⍝ The last element is Null.
<   1 2 3          ⍝ Strand encloses the simple vector.
< <   1 2 3        ⍝ Strand encloses the enclosed vector.
<   abc            ⍝ Enclosed character vector.
<   +              ⍝ Enclosed function expression.
<   `smbl          ⍝ Enclosed symbol.
<                  ⍝ Enclosed Null.
```

Indexing Arrays

A+ provides primitive functions to access the elements of an array. One such function is denoted by the bracket pair `[]` and is called Bracket Indexing. For example, using arrays displayed in the previous section:

```
     'axrTVw'[4]
V
     'axrTVw'[5 0 1]
wax
     (`sym;+;1 2 3 4;1.7 3.14;'example';)[2]
<   1 2 3 4
     (⍳2 3 5)[0;1;3]
 8
```

An omitted index implies all permitted indices for that axis, so one can easily obtain a row and a column:

```
     (⍳2 3 5)[0;0;]   ⍝ The first row.
 0 1 2 3 4
     (⍳2 3 5)[0;;4]   ⍝ The fifth column of the first plane of the array;
 4 9 14               ⍝ vector result.
```

For a 3-dimensional array, an item is a matrix. In Bracket Indexing, a semicolon may be omitted when all the indices following it are omitted, so one can index an array as if it were a vector containing the array's items:

```
     (ι2 3 5)[1]       ⍝ The second item: any element of it is
 15 16 17 18 19        ⍝ (ι2 3 5)[1;j;k] for some j and k.
 20 21 22 23 24
 25 26 27 28 29
```

More generally, one can index an array of rank r as if it were an (r-n)-rank array (frame) of rank-n cells. Say one has a five-dimensional array; one can view it as a three-dimensional array of two-dimensional cells:

```
     (ι 4 5 6 2 3)[0;0;0] ⍝ Any element of the first cell is
 0 1 2                  ⍝ (ι 4 5 6 2 3)[0;0;0;j;k]for some j, k.
 3 4 5                  ⍝ The first three indices index the frame.
```

One more example demonstrates the power of working with items, frames, and cells. For this example, a small part of the capability of the primitive function Take (↑) and the primitive operator Rank (@) must be explained. For positive `n`, the expression `n↑a` produces the first `n` items of `a`. The derived function `↑@1` applies Take to all cells of rank 1 in its right argument, i.e., to all rows, whose items are elements. Taking a certain number of elements in each row is equivalent to taking a certain number of columns. Thus the following expression takes three rows (items of a matrix) after taking five columns of a five by ten matrix:

```
     3↑5↑@1 ι5 10     ⍝  3↑(5(↑@1)(ι5 10)) is equivalent.
  0  1  2  3  4       ⍝  ↓↓ ↓....↓
 10 11 12 13 14       ⍝  ↓↓ Take 5 columns.
 20 21 22 23 24       ⍝ Take 3 rows.
```

Inquiring about Arrays

Shape and Rank

The primitive function denoted by the monadic (i.e., one-argument) form of the symbol `ρ` (rho) is called Shape. It produces the shape vector of its array argument. For example, `ρι2 3` is the vector 2 3, and `ρ'axrTVw'` is the one-element vector whose only element is 6.

The result of `ρρa`, a double application of Shape, is a one-element vector whose value is the rank of `a`. In particular, the element of the one-element vector `ρρ36` or `ρρ'X'` is 0; separately entered numbers and characters have no axes, and their rank is therefore 0; they are scalars.

Type and Depth

The Type monadic primitive function (`∨`) produces the type of its argument, as a scalar symbol. First, the six types of simple arrays.

```
     ∨ι2 5
 `int
     ∨2.71828 3.14159
 `float
     ∨'axrTVw'
 `char
     ∨`pp `rl
 `sym
     ∨{+}              ⍝ The parser requires the braces as a hint
 `func                 ⍝ that + is an argument.
     ∨<{+}
 `func               ⍝ A function scalar is also of type `func.
     ∨()
 `null
```

Next, the three types of nested arrays.

```
     ∨<2 3 4
 `box                ⍝ The type of a nested array is the type of its first
item.
     ∨`rl,(;2.7 3.1) ⍝ The comma joins its two arguments
 `sym                ⍝ into a single vector.
     ∨(+;)
 `func               ⍝ A function scalar.
```

Last, the four types of empty arrays.

```
     ∨''
 `char
     ∨ι0
 `int
     ∨0 12ρ10.1
 `float
     ∨0 4ρ(+;-;×;÷)
 `null
```

The Depth monadic primitive function (≡) produces the depth of nesting of its argument, as a scalar integer. The depth of a multi-item array is the greatest of the depths of its items. The depth of a function expression is -1, by convention, and the depth of a function scalar, which is an enclosed function expression, is 0.

```
     ≡ι2 3 4                 ⍝ Simple
 0
     ≡<'abc def'             ⍝ Result of Enclose
 1
     ≡(2 3;+;`a`b`c)         ⍝ Enclosure implied by strand
 1
     ≡(<2 3;=;`a`b`c)        ⍝ Strand with enclosed element
 2
     ≡(1 2;(3 4;(5 6;);7);8) ⍝ Strand in strand in strand
 3
```

A shorter definition of a *simple* array is any array whose depth does not exceed 0. A *nested* array, which is any array that is not simple, can be defined similarly as one whose depth is at least 1.

Pictorial Representation

To see a pictorial two-dimensional representation of data, use the "disp" (display) facility, available from `/common/a/disp.+`. After being loaded, it resides in the `disp` context. It is used as follows:

```
     $load disp
     disp.disp 2 3ρ('ab';`abc`def;ι2 4; 1.1 2.2;;×)
+3------------------------------+
2+2-+       +2-------+ +4------+|
||ab|       |`abc`def| 20 1 2 3||
|+"-+       +`-------+ |4 5 6 7||
|                      +i------+|
|+2-------+ +0         +0+      |
||1.1 2.2 | +∘         |×|      |
|+f-------+            +∇+      |
+⎕------------------------------+
```

Further information is available here.

Subtypes and Supertypes

Slotfillers

A special form of nested array called a *slotfiller* is recognized by certain primitive functions and toolkits. A slotfiller is a two-element vector `(sym;val)`. `sym` is a simple vector or scalar of distinct symbols. `val` has the same number of items as `sym` (recall that a scalar has one item). It can be either any nested scalar or any vector each of whose items either has a depth of at least 1 or is a function scalar that is the enclosure of a *defined* function. Thus primitive functions can appear in `val` only when they are enclosed at least twice, i.e., as enclosed function scalars. A slotfiller can be thought of as a dictionary of keys (with no repetitions) and values.

There is a way to test whether a variable or an expression is a slotfiller or not: `_issf x` is 1 if `x` is a slotfiller and 0 if it is not. Cf. the "Is a Slotfiller" section in the "System Functions" chapter.

Examples of slotfillers are:

```
    (`small `medium `large `super;(16;32;64;72))
    (`a;<97)
```

and

```
    (`g `l `w;(f;g;<{+}))
```

where `f` and `g` are user-defined functions, and + is enclosed by < and the strand; but not

```
    (`g `l `w;(+;-;×))
```

since nonnested primitive functions are prohibited in slotfillers.

Recall that when A+ displays a nested array, it uses an Enclose symbol (<) to indicate the beginning of the display of each nested array. It indents subarrays appropriately to show their total depth of nesting. The first sample slotfiller is displayed as:

```
<   `small `medium `large `super
< <   16
  <   32
  <   64
  <   72
```

The Pick function can extract values from slotfillers:

```
   `medium⊃(`small `medium `large `super;(16;32;64;72))
 32
```

Restricted Whole Numbers

Many functions require as arguments whole numbers that are within the range of integer representation but do not insist that the type of these arguments be integer. They also accept floating-point numbers that are tolerably equal to integers (see "Comparison Tolerance") and numbers whose absolute value is less than `1e-13`. I.e., they reject floating-point numbers that are significantly fractional or that are too large in magnitude to be represented as integer type. Furthermore, they accept empty arrays regardless of type. For convenience in this manual, the term restricted whole number is used for a member of the set consisting of the integers, these floating-point near-integers, and all empty arrays.

Since the functions that accept restricted whole number arguments use integers internally, floating-point values for these arguments involve a performance penalty, because of the implicit type conversion.

General Types

Many A+ functions and operators take arguments of several types, sometimes with some limitation, and it is convenient to have a terminology dividing A+ data objects into three classes, as they do. In this manual, these classes are called general types; they are:

- character, consisting of simple arrays of characters
- numeric, consisting of simple arrays, unrestricted as to value, of
  - floating-point numbers
  - integers
- mixed, containing all other data, namely
  - simple arrays of
    - functions
    - symbols
  - all nested arrays
    - box
    - function
    - symbol

Because general types are used mostly to indicate inclusion in the domains of functions and most functions accept empty arrays of any type, all empty arrays are included in each general type. (Although acceptance of empty arrays can cause anomalies like a character result for Add, such results are unlikely in fact to be created; if they do arise, they will probably be accepted by any function to which they are presented. For efficiency, the (empty) result of a mathematical function for a Null is whatever is most convenient for the function: Null, integer, or floating point.)

3. The Syntax and Semantics of A+

The main purpose of this chapter is to describe the syntax of A+, but through a series of examples, rather than in a formal way. Consequently some commonly understood terms are used without being formally defined. In particular, the phrase *A+ expression*, or simply *expression*, is taken to have the same general meaning it does in mathematics, namely a well formed sentence that produces a value. In addition, some discussion of semantics has been included, but only where it seemed reasonable in order to complete a description. A brief discussion of well formed expressions is presented at the end of this section, after all the rules for the components of expressions have been presented.

Names and Symbols

Primitive Function Symbols

A+ uses a mathematical symbol set to denote the functions that are native to the language, which are called *primitive functions*. This symbol set, part of the APL character set, consists of common mathematical symbols such as `+` and `×`, commonly used punctuation symbols, and specialized symbols such as `↑` and `↓`. In some cases it takes more than one symbol to represent a primitive function, as in `+/`, but the meaning can be deduced from the individual symbols. The symbols are listed in the table "Primitive Function and Operator Names and References".

Two of the symbols can be used alone, viz., `←` and `→`. If the execution of a function or operator has been suspended, they mean resume execution (with increased workspace size if necessary) and abandon execution, respectively; in the absence of a suspension, they are ignored. Instead of `→`, a dollar sign ($) can be used. Inside a function definition, an expression can consist of the symbol `←` alone, but it will be ignored, and the parser rejects `→` alone as a token error.

User Names

User names fall into two categories, unqualified and qualified. An *unqualified name* is made up of alphanumeric (alphabetic and numeric) characters and underbars (_). The first character must be alphabetic. For example, `a`, `a1c`, and `a_1c` are unqualified names, but `3xy` and `_xy` are not. (Although underbar is currently permitted as the first character in user names, this manual has been written as if it were not, and you should consider this form reserved for system names and avoid it.) The identifying words in control statements (case, do, else, if, while) are reserved by A+ for that use; they cannot appear as user names, even in qualified names.

A *qualified name* is either an unqualified user name preceded by a dot (`.`), or a pair of unqualified user names separated by a dot. In either case there are no intervening blanks. For example, `.xw1` and `w_2.r2_a` are qualified user names. An unqualified name preceding the dot in a qualified name is the name of a *context*. If there is a dot but no preceding name, the context is the *root* context.

System Names

System function names are unqualified names preceded by an underbar, with no intervening spaces, `_argv` for instance. The use of system function names is reserved by A+.

The name of an object traditionally (and therefore in A+) called a system variable is an unqualified name preceded by a backquote, with no intervening spaces. For example, `` `rl `` is the name of the system variable called Random Link. These objects cannot be dealt with directly in A+, but only through certain system and primitive functions and system commands, to which they act as parameters. As indicated in "Symbols and Symbol Constants", they look just like symbols (and may be considered such). They are not, however, the symbol forms of names: A+ will not recognize `rl`, for instance, as having anything to do with `` `rl ``; the quoted form `'rl'`, however, is recognized by system functions such as `_gsv`.

System Command Names

System command names begin with a dollar sign, followed immediately by an unqualified name, which is the name of the command. The name is sometimes followed by a space and then by a sequence of characters whose meaning is specific to the command, usually separated from the name by a space.

Comments

Comments can appear either alone on a line or to the right of an expression. A comment is indicated by the ⍝ symbol (usually called "lamp," since it looks like a bulb filament and since comments illuminate code), and it and everything to its right on the line constitute the comment. For example:

```
    a+b  ⍝ This is the A+ notation for addition.
```

Infix Notation and Ambi-valence

A+ is a mathematical notation, and as such uses infix notation for primitive functions with two arguments. In infix notation, the symbol or user name for a function with two arguments appears between them. For example, `a+b` denotes addition, `a-b` subtraction, `a×b` multiplication, and `a÷b` division.

In mathematics, the symbol - can also be used with one argument, as in -b, in which case it denotes negation. This is true in A+ as well. Because the symbol denotes two functions, one with one argument and the other with two, it is called *ambi-valent* (e.g., it uses "both valences"). A+ has extended the idea of ambi-valence to most of its primitive functions. For example, just as `-b` denotes the negative of `b`, so `÷b` denotes the reciprocal of `b`.

Defined functions cannot be ambi-valent.

Functions with one argument are called *monadic,* and functions with two arguments are called *dyadic*. One often speaks of the *monadic use* or *dyadic use* of an ambi-valent primitive function symbol.

Syntactic Classes

Numeric Constants

Individual numbers can be expressed in the usual integer, decimal, and exponential formats, with one exception: negative number constants begin with a "high minus" sign (¯) - including `¯Inf`, which we will come to later - instead of the more conventional minus sign (`-`), although negative exponents in the exponential format are denoted by the conventional minus sign.

Exponential format is of the form `1.23e5`, meaning 1.23 times 10 to the power 5, `¯5e2`, meaning -500, and `1e-2`, meaning .01. Only numbers can appear around the `e`. The one following it must be an integer - no decimal point - and have a regular minus sign if negative: a high minus there elicits a parse error report. A negative number before the `e` must have a high minus: a regular minus is considered to lie outside the format.

It is also possible to express a list of numbers as a constant, simply by separating the individual numbers by one or more blank spaces. For example:

```
 1.23 ¯7 45 3e-5
```

is a numeric constant with four numbers: 1.23, negative 7, 45, and .00003. `Inf` can appear in such a list. If you omit the blanks, A+ will give you a numeric vector, but probably not the one you intended. If a number is being parsed and a character is encountered that can't be part of the number, then a new number is started if the character could begin a number. For instance, `1e-3.5 40.358.62.7` is read by A+ as `0.001 0.5 40.358 0.62 0.7`.

Character Constants

A character constant is expressed as a list of characters surrounded by a pair of single quote marks or a pair of double quote marks. For a quote mark of the same kind as the surrounding quote marks to be included in a list of characters, it must be doubled. For example, both `'abc''d'` and `"abc'd"` are constant expressions for the list of characters `abc'd`. There is, however, a distinction between the two kinds of quotation marks.

Within single quotes (`'`) the C escape sequences and indeed any `\c` are not treated in any way, but left as is.

In strings contained within double quotes (`"`) these sequences and `\c` are treated as follows:

- `\n` is replaced by a newline character;
- *o*, *oo*, and *ooo* (each *o* a digit) are replaced by a character (see below); and
- the other sequences *simply have the leading backslash removed.*

These sequences and their translations are (where parenthesis indicates that A+ does not perform the substitution that the parenthesized term implies):

Double-Quote Translations

Name	String	Translation	Comment
newline	`\n`	newline character	
(horizontal tab)	`\t`	`t`	for tab use `"\11"`
(vertical tab)	`\v`	`v`	
(backspace)	`\b`	`b`	for backspace use `"\10"`
(carriage return)	`\r`	`r`	for carriage return use `"\15"`
(formfeed)	`\f`	`f`	for formfeed use `"\14"`
(audible alert)	`\a`	`a`	
backslash	`\\`	`\`	
question mark	`\?`	`?`	
single quote	`\'`	`'`	
double quote	`\"`	`"`	
octal number	*ooo*	a character	see below
(hex number)	*xhh*	*xhh*	
(any other char)	\c	c	

Thus `"\?\\"` is equal to `'?\'` and `"\r\t"` is equal to `'rt'`; `\"` prevents the double quote from ending a string within double quotes, and `\\` allows literal inclusion of `\` in a translated string in double quotes.

The translation of an octal sequence - which is of *variable length* and could be shown as \[[*o*]*o*]*o* - is best understood as occurring in three steps. First, the digits to be translated are found: there is at least one (else this would not be an octal sequence) and at most three, but the end of the string and any nondigit character also act as terminators. Second, the string of digits is taken as an octal number and is translated to a decimal number. Any `8` and `9` digits are accepted as 10 octal and 11 octal, and any overflow is ignored, since only the 256 residue is used. Third, the ASCII character corresponding to that number is found. If the string being translated is `digits`, the translation is
`` `char∨8⊥10 10 10⊤⍎digits`` where `1≤(⍴digits)≤3` and `∨digits` is `` `char``.

The foregoing implies these equivalences:
`"\99"←→"\121"`   `"\6a"←→"\006a"←→"\06",'a'`
`"\123456"←→"\123",'456'`.

Symbols and Symbol Constants

A symbol is a backquote (`` ` ``) followed immediately by a character string made up of alphanumeric characters, underscores (_), and dots (`.`). Symbol constants can be thought of as character-based counterparts to numeric constants, aggregating several characters into a single symbol. Just as `1 2.34 12e3 3e5` is a list of four numbers, so `` `a.s `12 `b`w_3`` is a list of four symbols. A backquote alone represents the empty symbol.

A user name, like `balance`, can be put in symbol form by placing a backquote before it, as in `` `balance``. A user name in symbolic form is always taken to refer to a global object (see "Scope of Names"), never a local object. If it has no dot in it, it refers to a global object in the current context.

System variable names, like `` `rl``, are in the form of symbols. Unlike backquoted user names, they are not decomposable. If `var` is a user name, then `` `var`` is recognized by A+ in certain situations as referring to the same object. A+ sees no relation, however, between `rl` and the system variable `` `rl``.

The Null

The Null is a special constant that can be formed as follows: `()`. It is neither numeric nor character, but has a special type, null. It is an empty vector, i.e., its rank is 1 and the length of its only axis is 0.

Variables

Variables are data objects that are named. They receive their values through Assignment, or Specification, which is denoted by the left-pointing arrow (←). For example, the expression

```
    abc←1 2 3
```

assigns the three-element list consisting of 1, 2, and 3 to the variable named `abc`. Any user name can serve as a variable name. For more on assignment, see the "Assignment, or Specification" section.

Functions and Function Call Expressions

Functions take zero or more arguments and return results. A sequence of characters that constitutes a valid reference to a function will be called a *function call expression*. That is, a function call expression includes a function symbol or name together with all its arguments and all necessary punctuation. It may also include unnecessary parentheses and blanks; if it does not, we will call it *irredundant*. In general, the arguments of a function are data objects, which may appear in function call expressions as variable names, constants, or expressions that require evaluation. In addition, for the various forms of function call expressions using braces, arguments can be function expressions (see "Function Expressions"). For example, `f{9.98;.0775;×}` and `f{59;125;g}`, where `g` is a defined function, are valid function call expressions.

A function with no arguments, or parameters, - which must be a defined or system, not a primitive, function - is said to be *niladic*. The only valid irredundant function call expression for a niladic function `f` is `f{}`.

Functions with one argument, *monadic* functions, can be primitive, defined, or system. The valid irredundant function call expressions for a function `f` with one argument `a` are `f a` and `f{a}`. In the form `f a`, the blank is required only if `f` followed by some initial part of `a` would form a valid name.

Dyadic functions can also be primitive, defined, or system. The valid irredundant function call expressions for a function `f` with two arguments `a` and `b` are `a f b` and `f{a;b}`, where `a` is called the *left argument* and `b` the *right argument*. In the infix form, each blank is required only if its absence could cause a name to be extended, and if the left argument is itself an infix expression it must be parenthesized.

Functions with more than two arguments must be defined or system, not primitive, functions. The only valid irredundant function call expression for a function of more than two arguments `a`, `b`, ... , `c` is `f{a;b; ... ;c}`.

In functional expressions that use braces, any position adjacent to a semicolon can be left blank. For example, each of the following is a valid functional expression: `f{a;}`, `f{;b}`, `f{;}`, `g{;a;b}`, `g{;;b}`. However, if `f` is monadic then `f{}` is not valid because `f{}` is reserved for niladic function call expressions. When an argument position is legitimately left blank, A+ assumes that the argument is the Null.

The number of arguments that a function takes is called its *valence*. The valence of a defined function is fixed by the form of its definition.

The table "Function Call Expressions and Function Header Formats" summarizes the function call expressions discussed here.

Operators and Derived Functions

There are three primitive formal operators in A+, known as Apply, Each, and Rank. By a *formal operator* we mean an operator in the mathematical sense, i.e., a function that takes a function as an operand, or produces a function as a result, or both. The resulting function is called a *derived function*.

The Apply and Each operators are both denoted by the dieresis, ¨. For a given function `f`, the function derived from the Each operator is denoted by `f¨`. The function `f` can be either monadic or dyadic, and `f¨` has the same valence as `f`. For a given function scalar `g`, where `g` is equal to `<{f}`, the function derived from the Apply operator is denoted by `g¨`. The function `f` can be either monadic or dyadic, and `g¨` has the same valence as `f`.

The Rank operator is denoted by the *at* symbol, `@`. Unlike the Each operator, the Rank operator has both a function argument and a data argument. For a given function `f` and data value `a`, the function derived from the Rank operator is denoted by `f@a`. This derived function has the same valence as `f`, which can be either monadic or dyadic.

A+ permits defined formal operators. As with primitive operators, only infix notation is allowed for operator and operands. Like the Each operator, the operand of a monadic defined operator is to the left of the operator name. For example, if the operator is `monop` then `+monop` denotes the derived function for `+`. In the case of a dyadic defined operator, one operand is on the left of the operator name and the other is on the right, like the Rank operator. For example, if the operator is `dyop` then `+dyop×` denotes the derived function for `+` and `×`. A dyadic defined operator can have a data right operand: see the note following the table "Operator Header Formats". See also the "Operator Syntax" section.

Unlike a primitive operator, the valence of a function derived from a defined operator is not determined by the valence of the function operands, but, like a defined function, by the form of the operator definition.

There are four other symbols (`\ . / ⍨`) that can appear with certain primitive function symbols and jot (`∘`), the resulting sequences representing functions. Their syntax might suggest that these symbols represent operators; however, not all primitive function symbols can be used in these sequences, and neither can defined function names. Consequently it would be misleading to think of them as formal operators, so we have simply listed all the sequences that are allowed. It is often convenient, however, to speak loosely of these sequences as representing derived functions, and of the three symbols in question as representing operators.

From now on, the general terms *operator* and *derived function* will include Apply, Each, Rank, defined operators, their derived functions, and the "operators" and "derived functions" in the table "Special Character Sequences (Quasi-Operators)".

Special Character Sequences (Quasi-Operators)

"Operator" Name	**"Derived" Functions**
Bitwise	`∨⍨` (Cast and Or) `∧⍨` `~⍨` `<⍨` `≤⍨` `=⍨` `≥⍨` `>⍨` `≠⍨`
Inner Product	`+.×` `⌊.+` `⌈.+`
Outer Product	`∘.+` `∘.-` `∘.×` `∘.÷` `∘.*` `∘.⌊` `∘.⌈` `∘.|` `∘.<` `∘.>` `∘.≤` `∘.≥` `∘.=` `∘.≠`
Reduction	`+/` `×/` `∧/` `∨/` `⌊/` `⌈/`
Scan	`+\` `×\` `∧\` `∨\` `⌊\` `⌈\`

Operator call expressions should be understood in terms of derived functions and function call expressions. Namely, an operator symbol and its function operands, or in the case of the Rank operator, its function operand to its left and its data object operand immediately to its right, form a derived function. A derived function is syntactically like any other function, and so can be used in the function position of any function call expression, as in f@a{c;d} and b f@a c . See the table "Operator Call Expressions" for a summary; it shows both irredundant expressions and expressions in which the derived functions are parenthesized. As in function call expressions, the blanks are not required in some instances and the left argument may need to be in parenthesis; moreover, a constant data operand and a constant right argument may require punctuation to separate them.

Operator Call Expressions

Operator Valence	Forms for Derived Function Having Monadic Valence		Forms for Derived Function Having Dyadic Valence	
monadic	(f op)a	f op a	a(f op)b	a f op b
	(f op){a}	f op{a}	(f op){a;b}	f op{a;b}
dyadic	(f op g)a	f op g a	a(f op g)b	a f op g b
	(f op g){a}	f op g{a}	(f op g){a;b}	f op g{a;b}

Function Expressions

The function arguments of operators are function expressions. The simplest function expressions are the names of defined functions and the symbols for primitive functions *other than Assignment and Bracket Indexing*. Any formulation of a derived function is also a function expression (see "Operators and Derived Functions").

Function expressions are limited to infix notation, since operators are limited to it.

A function expression can be enclosed in parentheses. For example, a(f@1)b is equivalent to a f@1 b. Moreover, a function expression is a valid function argument to a formal operator, and therefore quite complicated function expressions can be built. For example, +/ is a function expression, and therefore so are the following: +/¨, +/¨¨, and +/¨@a. See the "Scope Rules for Function Expressions" section.

Bracket Indexing

A+ data objects are arrays, and Bracket Indexing is a way to select subarrays. Bracket Indexing uses special syntax, whose form is

```
    x[a;b; ... ;c]
```

where x represents a variable name or an expression in parenthesis, a, b, ... , c denote expressions, and the number of semicolons is at most one less than the rank of the array being indexed. (The form x[] is, however, allowed for scalars.) The space between the left bracket and the first semicolon, between successive semicolons, and between the last semicolon and the right bracket, can be empty. If there are no semicolons, the space between the left and right brackets can be empty. Inserting semicolons immediately to the left of the right bracket does not change the meaning of the entire expression, as long as the maximum allowable number of semicolons is not exceeded. The form [a;b; ... ;c] is an *index group*. See "Sequences of Expressions" and "Bracket Indexing".

Expression Group

An expression group is a sequence of expressions contained in a pair of braces in which the expressions are separated by semicolons, where there is not a function expression immediately preceding it (except perhaps for spaces), so it is not a set of arguments for a function. Any of the expressions can be null, consisting of zero or more blanks. For example:

```
    {a;b; ... ;c;}
```

and

```
    {a;b; ... ;c}
```

are expression groups, where `a`, `b`, ... ,`c` denote expressions. See "Sequences of Expressions".

Expression Result and Expression Group Result

The result of an expression is the result of the last function executed in the expression, whether primitive, defined, or derived. See "Well Formed Expressions".

The result of an expression group is the result of the last expression executed. It is possible that the last expression in the group may not be the last one executed - indeed, may not be executed at all; see "Result".

Strands

Aggregate data objects (nested arrays) can be formed by separating the individual data objects by semicolons and surrounding the result with a pair of parentheses. For example:

```
    (a;b; ... ;c)
```

where `a`, `b`, ... , `c` denote expressions. Any of these expressions can be function expressions. There must be at least one semicolon. See "Sequences of Expressions".

Function Scalars

The above strand notation produces objects with at least two elements. One-element aggregates of data can be formed with the primitive function Enclose, denoted by `<`. A one-element object holding a function expression, such as

```
    <{a}
```

where `a` is a function expression, is called a function scalar.

The symbol `¨`, used also for the Each operator, serves as the Apply operator when the operand (argument) of the operator is a function scalar. For example, `a(<{ρ})¨b` is `aρb`.

Assignment, or Specification

The Assignment primitive, denoted by `←`, is used to associate a name with a value. For example:

```
    a←1
    f←+
```

assigns the value 1 to the name `a` and the function Add to the name `f`. The name to the left of the assignment arrow is assigned the value of the expression to the right. If that expression is a function expression, the name to which it is assigned represents a function - not the name of a function, but a function itself. Otherwise it represents a variable.

A series of names can be associated with a series of values, using strand notation; for example,

```
    (a;b;c)←(1 2 3;3 4⍴7;'txt')
```

Ordinary Assignment can also be expressed as `(a)←b`. Any appearance of `a←b` inside a function or operator definition means that `a` will be a local variable, if `a` is an unqualified name. The form `(a)←b` can be used to assign a value to the global variable `a`, provided that `a←`... doesn't appear elsewhere in the definition. If both `a←`... and `(a)←`... appear, they are equivalent: the latter has no special significance.

Assignment behaves somewhat like a dyadic function, in that it has a result, namely, the right argument. The left argument expression is syntactically limited to certain forms. See the table "Targets of Selective Assignment", for a summary of Selective Assignment target expressions, which are additional to those in ordinary assignment.

Assignment, in any form, cannot be the operand of an operator.

Precedence Rules

Precedence rules describe a hierarchy in the syntactic elements of a language that determines how these elements are grouped for execution in an expression. For example, in mathematics × has higher precedence than +, which means that × is evaluated before +. For example, in the mathematical expression a×b+c, the subexpression a×b is grouped for execution, and the result is added to c.

The precedence rules in A+ are simple:

- all functions have equal precedence, whether primitive, defined, or derived
- all operators have equal precedence
- operators have higher precedence than functions
- the formation of numeric constants has higher precedence than operators.

Right to Left Order of Execution

The way to read A+ expressions is from left to right, like English. For the most part we also read mathematical notation from left to right, although not strictly, because the notation is two-dimensional. To illustrate reading A+ expressions from left to right, consider the following examples.

```
b+c+d     ⍝ Read as: "b plus the result of c plus d."
x-÷y      ⍝ Read as: "x minus the reciprocal of y."
```

As you can see, reading from left to right in the suggested style implies that execution takes place right to left. In the first example, to say "`b` plus the result of `c` plus `d`" means that `c+d` must be formed first, and then added to `b`. And in the second example, to say "`x` minus the reciprocal of `y`" means that `÷y` must be formed before it is subtracted from `x`.

To be sure, reading from left to right is not necessarily associated with execution from right to left. For example, the expression `b÷c+d` is read left to right in conventional mathematical notation as well as A+, but the order of evaluation is different in the two; in mathematics `b` divided by `c` is formed and added to `d`, and consequently the expression is read as "`b` divided by `c`, [pause] plus `d`," while in A+, `b` is divided by `c+d`. The order of execution is controlled by the relative precedence of the functions, or operations. In mathematics, division has higher precedence than addition, so that in `b÷c+d`, division is performed before addition.

Another way to say that A+ expressions are executed from right to left is that functions have long right scope and short left scope. For example, consider:

```
a+b-c÷e×f
```

The arguments of the subtraction function are `b` on the left (short scope) and `c÷e×f` on the right (long scope). The left argument is found by starting at the subtraction symbol and moving to the left until the smallest possible complete subexpression is found. In this example it is simply the name `b`. If the first nonblank character to the left of the symbol had been a right parenthesis, then the left argument would have included everything to the left up to the matching left parenthesis. For example, the left argument of subtraction in `a+(x÷b)-c÷e×f` is `x÷b`.

The right argument is found by starting at the function symbol and moving to the right, all the way to the end of the expression; or until a semicolon is encountered at the same level of parenthesization, bracketing, or braces; or until a right parenthesis, brace, or bracket is encountered whose matching left partner is to the left of the symbol. In the above example, the right argument of subtraction is everything to its right. If the case of `a+b-(c÷e)×f`, the right argument is also everything to its right. However, for `a+(b-c÷e)×f`, the right argument is `c÷e`.

Scope Rules for Function Expressions

Interestingly enough, the scope rules for function expressions are the mirror image of those for ordinary expressions. Namely, operators have long scope to the left and short scope to the right. For example, `+/¨@a` is equivalent to `((+/)¨)@a`, and if `dyop` is a dyadic defined operator, `+dyop⍉¨` is equivalent to `(+dyop⍉)¨`, not `+dyop(⍉¨)`.

Sequences of Expressions

Index groups, expression groups, and strands are forms for sequences of expressions separated by semicolons. The expressions in an expression group are executed in the order suggested for reading, from left to right, like successive statements in a function. Index groups and strands, however, fall within other expressions and are executed right to left. For example, if the variable `a` has the value 2 and the strand

```
b←(a←5;a×a)
```

is executed, the value in the second element of `b` will be 4, proving that the assignment `a←5` happened after the multiplication `a×a`. (A Strand Assignment, however, like an expression group, is executed left to right, after its righthand argument has been evaluated in the usual way.)

To improve readability in source files, sequences of expressions are often broken at the semicolons and continued on the next physical line. Note that in such cases for expression groups the left to right order of execution for the expressions within a sequence becomes a natural top to bottom order.

Execution Stack References

Execution stack references are `&`, `&0`, `&1`, etc. The symbol `&` can be used in a function definition to refer to that function. For example, a factorial function can be defined in either of the following ways:

```
fact{n}:if (n>0) n×fact{n-1} else 1
```

```
fact{n}:if (n>0) n×&{n-1} else 1
```

When execution is suspended, the objects on the execution stack can be referred to by `&0` (top of the stack), `&1`, and so on. These objects can be examined and respecified, and execution resumed (←). The left to right order of arguments generally corresponds to increasing stack numbers.

In the definition of a dependency `a`, the symbol `&` refers to that definition but `a` always denotes the (stored) value of `a`, whereas in the definition of a function `f`, both `&` and `f` denote the definition of `f`.

Control Statements

For the interpretation of these control statements, see the "Control Statements" chapter. The words `case`, `do`, `else`, `if`, and `while` are reserved by A+; they cannot be employed as user names.

Case Statement

The form of a case statement is the word `case`, followed by an expression in parentheses, followed by an expression group. When `case` followed by an expression in parenthesis is entered alone on a line (with no pending unbalanced punctuation), the statement is taken to be complete, with Null for the expression group.

Do Statement

There are two do statements, which together have the same syntax as an ambi-valent primitive function (with the word `do` in place of the function symbol). Both the monadic and dyadic forms have an expression or expression group to the right of the word `do`. The dyadic form also has an expression to the left which would serve as the left argument if the word `do` were the name of a dyadic function. In the absence of pending punctuation, if `do` is entered alone on a line, it is taken to be complete, and echoed by A+, and if it is preceded by an expression but followed by nothing, a parse error is reported.

If Statement

The form of an if statement is the word `if`, followed by an expression in parentheses, followed by another expression or an expression group. When `if` followed by an expression in parenthesis is entered alone on a line (with no pending unbalanced punctuation), the statement is taken to be complete, with Null for the expression group.

If-Else Statement

The form of an if-else statement is the word `if`, followed by an expression in parenthesis, followed by an expression or expression group, followed by the word `else`, followed by another expression or expression group. When an if-else is entered, if there is nothing following the `else`, a parse error is reported in the absence of pending punctuation.

While Statement

The form of a while statement is the word `while`, followed by an expression in parentheses, followed by another expression or an expression group. When `while` followed by an expression in parenthesis is entered alone on a line (with no pending unbalanced punctuation), the statement is taken to be complete, with Null for the expression group. If the expression in parenthesis is valid and nonzero, it is necessary to interrupt execution (by **Control-c Control-c**) before anything else can be done.

Function Definitions

A function definition consists of a function header, followed by a colon, followed by the function body, which is either an A+ expression or an expression group.

Function headers take the same forms as functional expressions (see "Functions and Function Call Expressions"), except that only names can appear and none can be omitted. A function header has the monadic form, dyadic form, or general form. The monadic form is the function name followed by the argument name, with the two names separated by at least one space. For example, if the function name is `correlate` then

```
correlate a:{ ... }
```

is a function definition with the monadic form of the header.

The dyadic form of function header is the function name with one argument name on each side, with the names separated by at least one blank. For example:

```
a correlate b:{ ... }
```

is a function definition with the dyadic form of the header.

The third form of function header is the general form, which is the function name followed by a left brace, followed by a list of from zero to nine argument names separated by semicolons, and terminated by a right brace. For example:

```
correlate{a;b;c}:{ ... }
```

is a function definition with the general form of the header. In this example the function has three arguments. Names must appear in all positions of the argument list - no position can be left empty. (In a niladic function definition no argument position is left empty; there just is no argument position.)

A function with one argument can be defined with either the monadic form of function header or the general form and a function with two arguments can be defined with either the dyadic form or the general form. In a reference to the function, either form (of the correct valence) can be used, no matter how it was defined.

The number of arguments of a defined function is nine or fewer. See the table "Function Call Expressions and Function Header Formats" for a summary.

Note: A position adjacent to a semicolon can be left empty for function call expressions.

Function Call Expressions and Function Header Formats

Valence	**Forms (see note above)**
niladic	`f{}`
monadic	`f a` or `f{a}`
dyadic	`a f b` or `f{a;b}`
general	`f{a;b; ... ;c}`

Function Result

The result of a defined function is the result of the expression or expression group that forms the function body. The result can be used in the same ways as the result of a primitive function.

Operator Definitions

An operator definition consists of an operator header, followed by a colon, followed by the body of the definition, either an A+ expression or an expression group. The header must be in infix, not general, form.

An operator can be monadic or dyadic, depending on whether it has one argument or two, and the derived function can also be monadic or dyadic. Consequently there are four forms for the header. See the table "Operator Header Formats" for a summary.

Note the parentheses in the forms in this table. While parentheses are not necessary in operator call expressions, they are necessary in operator definition headers to specify the function expression part. Compare with the table "Operator Call Expressions".

Operator Header Formats

Operator Valence	Monadic Derived Function	Dyadic Derived Function
monadic	`(f op)a`	`a(f op)b`
dyadic	`(f op h)a`	`a(f op h)b`

NOTE: In the dyadic form, if the right operand is the letter `g`, then it must be a function; otherwise, it must be data unless every occurrence in the body of the operator syntactically requires it to be a function.

Operator Result

The result of a defined operator, which is strictly speaking the result of the derived function, is the result of the expression or expression group that forms the body of the definition. The result can be used in the same ways as the result of a primitive operator.

Dependency Definitions

A dependency definition consists of a name (the name of the dependency), followed by a colon, followed by either an A+ expression, or an expression group. An itemwise dependency has the same form except that the name is followed by `[i]` where `i` can be any unqualified user name (except the name of the dependency).

Dependency Result

The result of a dependency is either a value that was assigned to the name, or the result of the expression or expression group that forms the definition, or, for itemwise dependencies, a combination of the two - see the "Dependencies" chapter. The results of dependencies are referenced in the same way that values of variables are referenced, simply by their names.

Well-Formed Expressions

A well-formed expression is one of the basic forms described above, in which all of the constituent expressions are well formed. The potential for complicated expressions arises from the fact that every one of these basic forms produces a result and can therefore be used as a constituent in other forms, except that the right arrow (→) can only appear alone and the left arrow (←) must appear alone unless it has an expression to its right. In this building of expressions from simpler ones A+ is very much like mathematical notation.

The concept of the *principal subexpression* of an expression is useful for analysis. As execution of an expression proceeds in the manner described in the "Right to Left Order of Execution" section, one can imagine that parts of the expression are executed and replaced by their results, and then other parts are executed using these results, and are replaced by their results, and so on. Ultimately the execution comes to the last expression to be executed, which is called the principal subexpression. Once it is executed, its value is the value of the expression. If the principal subexpression is a function call expression or operator call expression, that function or derived function is called the *principal function*.

For example, the principal subexpression of `(a+b÷c-d)⋆10×n` is `x⋆y`, where `x` is the result of `a+b÷c-d` and `y` is the result of `10×n`. The power function `⋆` is the principal function.

As a second example, the principal expression of `(x+y;x-y)` is `(w;z)`, where `w` is `x+y` and `z` is `x-y`. In this case we do not refer to a principal function; the last thing done in executing the expression is what is implied by the strand notation - enclosing `w` and `z` and catenating them.

4. Monadic Scalar Functions

Name	Symbol	Name	Symbol	Name	Symbol
Absolute value	|	Identity	+	Pi times	○
Ceiling	⌈	Natural log	⍟	Reciprocal	÷
Exponential	⋆	Negate	-	Roll	?
Floor	⌊	Not	~	Sign	×

As stated in the introduction, the term *integer* is used in this manual to indicate not only a domain of values but also a particular internal representation. To refer to the same domain of values when both integer and floating-point representations are allowed, the term *restricted whole number* is used. These floating-point representations need only be tolerably equal to the integers.

Classification of Monadic Scalar Functions

Although they are listed alphabetically in this chapter, for convenient reference, the A+ monadic scalar primitive functions can be grouped - among other ways, to be sure - in four categories:

- the most common arithmetical functions: Reciprocal, Negate, Identity;
- other arithmetical functions: Exponential, Natural log, Pi times, Roll;
- extractive functions: Sign, Absolute value, Floor, Ceiling;
- logical function: Not.

Application and Result Shape

All monadic scalar functions produce scalars from scalars, and apply element by element to their arguments: they are applied to each element independently of the others. Consequently, the shape of the result is the same as the shape of the argument. This behavior is assumed in the following descriptions.

Error Reports

Multiple errors elicit but a single report. With only one exception, the error reports for monadic primitive scalar functions are common to all such functions. There are six reports, including interrupt, and each error report on the following list is issued only if none of the preceding ones apply:

- parse: this error class includes valence errors, which must result from three or more arguments in braces, since every symbol for a monadic scalar primitive function is also used for a dyadic function;
- value: the argument has no value;
- nondata: the argument is a function or some other nondata object;
- type: the argument is not a simple numeric array **-** for Not, of restricted whole numbers, and, for Natural Log, of nonnegative numbers -; the Identity function, however, cannot cause this error report;
- wsfull: the workspace is currently not large enough for execution of the function; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;
- interrupt (not an error): the user pressed **c** twice (once if A+ was started from a shell) while holding the **Control** key down.

An inadvertent left argument results not in a valence error, but in the invocation of a dyadic function that shares the function symbol.

Function Definitions

Absolute value |*x*

Argument and Result

The argument and result are simple numeric arrays. The result for an integer argument is integer if possible.

Definition

The absolute value of *x*. In other words, |*x* is equivalent to *x* times Sign of *x*.

Example

```
     |12.3 ¯3
 12.3 3
```

Ceiling   ⌈*x*

Argument and Result

The argument and result are simple numeric arrays. The result consists of nonfractional numbers, and is integer if all its elements can be represented that way (including if empty). If some element of the result has too great a magnitude to be represented as an integer, the result is floating point.

Dependency

Comparison tolerance, for most floating-point numbers (see "Comparison Tolerance").

Definition

The smallest nonfractional number greater than `x` or tolerably equal to `x`, except that `⌈x` is 0 when `x` exceeds zero but is equal to or less than `1e-13` (intolerantly).

Example

```
      ⌈10 10.2 10.5 10.98 ¯9 ¯9.2 ¯9.5 ¯9.98, 10+1e-13
 10 11 11 11 ¯9 ¯9 ¯9 ¯9 10
```

Exponential `*x`

Argument and Result

The argument and result are simple numeric arrays. The result is always floating point.

Definition

e (2.71828...) to the power `x`.

Example

```
      *¯1 0 1 2 710
 .3678794412 1 2.718281828 7.389056099 Inf
```

Floor `⌊x`

Argument and Result

The argument and result are simple numeric arrays. The result consists of nonfractional numbers, and is integer if all its elements can be represented that way (including if empty), else floating point.

Dependency

Comparison tolerance, for most floating-point numbers (see "Comparison Tolerance").

Definition

The largest nonfractional number less than `x` or tolerably equal to `x`, except that `⌊x` is 0 when `x` is less than zero but is equal to or greater than `¯1e-13` (intolerantly).

Example

```
      ⌊10 10.2 10.5 10.98 ¯9 ¯9.2 ¯9.5 ¯9.98, 10-1e-13
 10 10 10 10 ¯9 ¯10 ¯10 ¯10 10
```

Identity `+x`

Argument and Result

The argument, which is also the result, can be any array. (A type error cannot occur.)

Definition

The result is identical to `x`.

Example

```
      +'abc'
abc
```

Natural log `⍟x`

Argument and Result

The argument and result are simple numeric arrays. The elements of the argument must be nonnegative. The result is always floating point.

Definition

The natural logarithm of `x`, i.e., the logarithm of `x` to the base *e* (2.71828...).

Example

```
      ⍟ 1 10 100 0
 0 2.302585093 4.605170186 ¯Inf
```

Negate `-x`

Argument and Result

The argument and result are simple numeric arrays. The result for an integer argument is integer if possible.

Definition

`0-x`.

Example

```
      -23 ¯2 45 0 ¯1 .5
 ¯23 2 ¯45 0 1 ¯0.5
```

Not `~x`

Argument and Result

The argument is a simple array of restricted whole numbers. The result is always of integer type.

Definition

The value is 1 if `|x` is almost 0, viz., less than `1e-13` (intolerantly), and 0 otherwise.

Examples

```
    ~0 1
1 0
    ~¯1 0 1 2 3
0 1 0 0 0
```

Pi times ○*x*

Argument and Result

The argument and result are simple numeric arrays. The result is always floating point.

Definition

Pi (3.14159...) times *x*. The result is *Inf* or ¯*Inf* if it cannot be represented otherwise.

Example

```
    ○1 2 .5 1e308
3.141592654 6.283185307 1.570796327 Inf
```

Reciprocal ÷*x*

Argument and Result

The argument and result are simple numeric arrays. The type of the result is always floating point.

Definition

1÷*x*. The result is *Inf* or ¯*Inf* for elements that cannot be represented otherwise; in particular, the result is *Inf* for 0.

Example

```
    ÷.5 1.5 ¯2 100 0 ¯1e-309
2 0.6666666667 ¯0.5 0.01 Inf ¯Inf
```

Roll ?*x*

Argument and Result

The argument and result are simple arrays of restricted whole numbers. The result is always integer.

Dependency

The value of the Random Link system variable, `*rl*, which is changed each time a random number is chosen.

Definition

x is an array of positive restricted whole numbers, and the value is an array of integers with the same shape as *x*. Each element of the result is a random integer chosen from ⍳*e*, where *e* is the corresponding element of *x*. The result is dependent on the random link, `` `rl ``, which is set when the Random Link system command, `$rl`, is executed and each time a random integer is chosen.

Example

```
     ?20⍴10
 3 4 7 2 1 5 0 1 7 0 7 9 9 6 9 3 9 0 0 6
     ?20⍴10
 2 9 2 4 5 1 3 5 8 3 0 3 7 8 6 9 5 8 0 4
```

Sign ×x

Argument and Result

The argument and result are simple numeric arrays. The type of the result is always integer.

Definition

Signum *x*. The value of ×*x* is -1 for negative elements, 0 for zero, and 1 for positive elements.

Example

```
     ×100 ¯2.5 0 5 ¯Inf
 1 ¯1 0 1 ¯1
```

5. Dyadic Scalar Functions

As stated in the introduction, the term *integer* is used in this manual to indicate not only a domain of values but also a particular internal representation. To refer to the same domain of values when both integer and floating-point representations are allowed, the term *restricted whole number* is used. These floating-point representations need only be tolerably equal to the integers.

Classification of Dyadic Scalar Functions

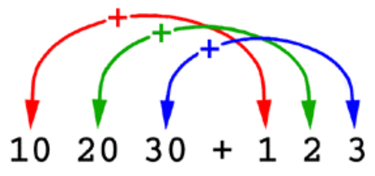
Although they are listed alphabetically in this chapter, for convenient reference, the A+ dyadic scalar primitive functions can be grouped in five categories:

- the most common arithmetical functions: Add, Subtract, Multiply, Divide, Power;
- other computational functions: Residue, Log, Circle, Combine Symbols;
- selection functions: Max (greater of), Min (lesser of);
- comparison functions: Equal to, Not equal to, Less than, Less than or Equal to, Greater than, Greater than or Equal to;
- logical functions: And, Or.

Application, Conformability, and Result Shape

All dyadic scalar functions produce scalars from scalars, and apply element by element to their arguments: they are applied to each pair of elements - one from each argument - independently of the others. There are three conformable cases for dyadic scalar functions:

1. The arguments have identical shapes. In this case, corresponding elements from the two arguments are paired. The shape of the result equals the common shape of the arguments.



11 22 33

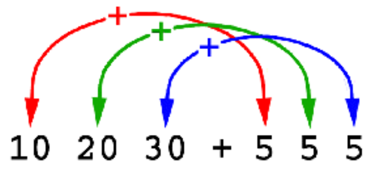
2. One argument has exactly one element and the other does not. Then the single element (a "singleton") from the one argument is paired independently with each element of the other. The shape of the result equals the shape of the one with either more or fewer than one element. This case is called *scalar extension*.

10 20 30 + 5

is equivalent to

10 20 30 + 5 5 5

then:



15 25 35

3. Each argument has one element. Then the result has a single element also. The rank of the result equals the larger of the two argument ranks.

```
      2+,2        ⍝ Shapes are conformable
4

      ⍴2+2        ⍝ Scalar plus scalar
(empty)

      ⍴2+,2       ⍝ Scalar plus one-element vector
1
```

The element-by-element application of the functions and the above conformability rules for their arguments are assumed in the following descriptions.

Common Error Reports

Multiple errors elicit but one report. Eight reports, including interrupt, are common to all dyadic primitive scalar functions, and each of these reports is issued only if none of the preceding ones apply:

- parse: this error class includes valence errors that result from three or more arguments in braces;
- value: an argument has no value;
- nondata: an argument is a function or some other nondata object;
- type: an argument is of an illicit type;
- rank: conformability rules are not satisfied and the ranks of the arguments differ;

- length: conformability rules are not satisfied because of a mismatch in a dimension of the arguments;
- wsfull: the workspace is currently not large enough for execution of the function; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;
- interrupt (not an error): the user pressed **c** twice (once if A+ was started from a shell) while holding the **Control** key down.

Except where noted, the omission of the left argument results not in a valence error report, but in the invocation of a monadic function or an operator that shares the function symbol.

Function Definitions

Add `y+x`

Arguments and Result

The arguments and result are simple numeric arrays. For two nonempty arguments, the result is integer if both arguments are integer and all result elements lie inside the range of integer representation, and floating point otherwise. If exactly one argument is empty, the result is floating point if that argument is floating point, and otherwise its type is the type of the nonempty argument. If both are empty, then if one is floating point and the other integer the result is floating point, and otherwise its type is the type of the right argument.

Definition

`y` plus `x`. The result may include `Inf` or `¯Inf`.

Example

```
     ¯1 0 1 1e308+10 20 30 1e308
 9 20 31 Inf
```

And `y∧x`

Arguments and Result

The arguments are simple arrays of restricted whole numbers. The result is an integer array.

Definition

If `x` and `y` have boolean values (0 and 1) then `y∧x` is the Logical And of `y` and `x`. That is:

- `1∧1` equals `1`
- Any other boolean combination (`1∧0` or `0∧1` or `0∧0`) equals `0`

  And is strictly boolean, never bitwise. All nonzero restricted whole numbers are treated as if they were 1.

  To get bitwise behavior, use the Bitwise operator.

Additional Error Report Condition

If none of the common error conditions are reported (including an illicit, i.e., not simple numeric, type) then:

- a type error is reported if an argument is not a restricted whole number.

Examples

```
     0 0 1 1∧0 1 0 1
 0 0 0 1
     43∧14
 1
```

Circle *y*○*x*

Arguments and Result

The arguments and result are simple numeric arrays. Additionally, the left argument can also be symbolic. The result is always in floating point.

Definition

Strictly, the elements of a numeric left argument *y* must be restricted whole numbers from -7 to 7; however, all floating-point numbers greater than -8 and less than 8 are accepted and, in effect, rounded toward zero to produce integers. Each element of *y* indicates the trigonometric, hyperbolic, or algebraic function to be applied to the corresponding element of the right argument *x*. All angles are in radians. See the table for details.

Notation for the Circle Functions

A+ Expression	Meaning		A+ Expression	Meaning
`` `sinarccos○x`` or `0○x`	`(1-x*2)*0.5`			
`` `sin○x`` or `1○x`	sin x		`` `arcsin○x`` or `¯1○x`	arcsin x
`` `cos○x`` or `2○x`	cos x		`` `arccos○x`` or `¯2○x`	arccos x
`` `tan○x`` or `3○x`	tan x		`` `arctan○x`` or `¯3○x`	arctan x
`` `secarctan○x`` or `4○x`	`(1+x*2)*0.5`		`` `tanarcsec○x`` or `¯4○x`	`(¯1+x*2)*0.5`
`` `sinh○x`` or `5○x`	sinh x		`` `arcsinh○x`` or `¯5○x`	arcsinh x
`` `cosh○x`` or `6○x`	cosh x		`` `arccosh○x`` or `¯6○x`	arccosh x
`` `tanh○x`` or `7○x`	tanh x		`` `arctanh○x`` or `¯7○x`	arctanh x

When both arguments are scalar, using the symbolic form adds about 40% to the processing time; the symbolic form adds less, of course, when the right argument is non-scalar. Symbolic form is heartily encouraged for all but the most time-critical applications.

Additional Error Report

If none of the common errors listed above are reported, then:

- a domain error is reported if the absolute value of an element of the left argument is equal to or greater than 8, and also if the absolute value of the right argument is less than 1 for left argument -4 or greater than 1 for left arguments of 0, -1, -2, and -7.

Example

```
  1 1 3 ¯3 ○ ○÷2 4 2 0  ⍝ sin(pi/2), sin(pi/4), tan(pi/2), arctan(Inf)
1 0.7071067812 1.633177873e+16 1.570796327
```

Combine Symbols *y*∪*x*

Arguments and Result

The arguments and result are simple arrays of symbols.

Definition

This function takes context names and unqualified names and produces qualified names. More generally, for each scalar pair *y*,*x*: if, as displayed, *x* has a dot (period) in it, then the value of *y*∪*x* is *x*, and *y* is ignored; otherwise, the result is the symbol that is displayed as *y*, followed by a dot, followed by *x* without its backquote.

Examples

```
    `c ∪ `x `d.y `.z
`c.x `d.y `.z
    `b.c `a ∪ `x `y
`b.c.x `a.y
```

Divide *y*÷*x*

Arguments and Result

The arguments and result are simple numeric arrays. The result is always floating point.

Definition

y divided by *x*. Division of a positive number by zero yields *Inf*, a unique scalar, and division of a negative number by zero yields ¯*Inf*.

Additional Error Report

If none of the errors listed in "Common Error Reports" are reported, then:

- a domain error is reported for 0÷0.

Example

```
    0 1 2 3 4 ¯5÷2 2 2 2 0 0
0 0.5 1 1.5 Inf ¯Inf
```

Equal to `y=x`

Arguments and Result

The arguments can be of any type. The result is boolean (integer type with values 0 and 1).

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

The value is 1 if *y* tolerably equals *x*, and 0 if not.

Additional Error Report

If there is no parse or value error (see "Common Error Reports"), then:

- a valence error is reported if the left argument is missing.

Examples

```
      ' '='this is it'
0 0 0 0 1 0 0 1 0 0
      (<2 3, 4+1e-13)=(2 3 4;'abcde';⍳5 6)
1 0 0
      1 2 3 = '123'
0 0 0
```

Greater than `y>x`

Arguments and Result

The arguments are simple numeric, character, or symbol arrays. The result is boolean (integer type with values 0 and 1).

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

The value is 1 if *y* is greater than *x* and not tolerably equal to *x*, and 0 otherwise. Characters are compared using their ASCII codes and symbols using the usual lexical ordering based on the ASCII codes of their component letters.

Examples

```
      (¯200 0 90 100 101 200,(100+1e-12),100+1e-11)>100
0 0 0 0 1 1 0 1
      'b' > 'abc'
1 0 0
      'B' > 'abc'      ⍝ ASCII, not English, order.
0 0 0
      `b > `a`b`c
1 0 0
      `B > `a`b`c      ⍝ Likewise.
0 0 0
```

```
      `pint > `cup `pints `pound `quart `snootful `gallon
 1 0 0 0 0 1
```

Greater than or Equal to $y \geq x$

Arguments and Result

The arguments are simple numeric, character, or symbol arrays. The result is boolean (integer type with values 0 and 1).

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

The value is 1 if y is greater than x or tolerably equal to x, and 0 otherwise. Characters are compared using their ASCII codes and symbols using the usual lexical ordering based on the ASCII codes of their component letters.

Additional Error Report

If there is no parse or value error (see "Common Error Reports"), then:

- a valence error is reported if the left argument is missing.

Example

```
      ¯200 0 90 100 101 200≥100
 0 0 0 1 1 1
```

Less than $y < x$

Arguments and Result

The arguments are simple numeric, character, or symbol arrays. The result is boolean (integer type with values 0 and 1).

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

The value is 1 if y is less than x and not tolerably equal to x, and 0 otherwise. Characters are compared using their ASCII codes and symbols using the usual lexical ordering based on the ASCII codes of their component letters.

Example

```
      ¯200 0 90 100 101 200<100
 1 1 1 0 0 0
```

Less than or Equal to `y≤x`

Arguments and Result

The arguments are simple numeric, character, or symbol arrays. The result is boolean (integer type with values 0 and 1).

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

The value is 1 if `y` is less than `x` or tolerably equal to `x`, and 0 otherwise. Characters are compared using their ASCII codes and symbols using the usual lexical ordering based on the ASCII codes of their component letters.

Additional Error Report

If there is no parse or value error (see "Common Error Reports"), then:

- a valence error is reported if the left argument is missing.

Example

```
     ¯200 0 90 100 101 200≤100
 1 1 1 1 0 0
```

Log `y⍟x`

Arguments and Result

The arguments and result are simple numeric arrays. The result is always in floating point.

Definition

The logarithm of `x` to the base `y`.

Example

```
     10⍟.1 1 10 100 1000 1234.5 0
 ¯1 0 1 2 3 3.091491094 ¯Inf
```

Additional Error Report

If none of the reports cited in "Common Error Reports" is issued, then:

- a domain error is reported if an element of either argument is negative or if corresponding elements of the two arguments are both 1.

Max `y⌈x`

Arguments and Result

The arguments and result are simple numeric arrays. For two nonempty arguments, the result is integer if both arguments are integer, and floating point otherwise. If exactly one argument is empty, the result is floating point if that argument is floating point, and otherwise its type is the type of the nonempty argument. If both are empty, then if one is floating point and the other integer the result is floating point, and otherwise its type is the type of the right argument.

Definition

The greater of `y` and `x`. When this function is used in Reduction (`⌈/`), the name Max is appropriate.

Example

```
    0 ⌈ 3 .5 ¯1 5 ¯.1
3 .5 0 5 0
```

Min `y⌊x`

Arguments and Result

The arguments and result are simple numeric arrays. For two nonempty arguments, the result is integer if both arguments are integer, and floating point otherwise. If exactly one argument is empty, the result is floating point if that argument is floating point, and otherwise its type is the type of the nonempty argument. If both are empty, then if one is floating point and the other integer the result is floating point, and otherwise its type is the type of the right argument.

Definition

The lesser of `y` and `x`. When this function is used in Reduction (`⌊/`), the name Min is appropriate.

Example

```
    99.5 100 91.1 112 99 ⌊ 100
99.5 100 91.1 100 99
```

Multiply `y×x`

Arguments and Result

The arguments and result are simple numeric arrays. For two nonempty arguments, the result is integer if both arguments are integer and all result elements lie inside the range of integer representation, and floating point otherwise. If exactly one argument is empty, the result is floating point if that argument is floating point, and otherwise its type is the type of the nonempty argument. If both are empty, then if one is floating point and the other integer the result is floating point, and otherwise its type is the type of the right argument.

Definition

`y` times `x`.

Example

```
    10×0 1 2 3 1e308
0 10 20 30 Inf
```

Not equal to *y*≠*x*

Arguments and Result

The arguments can be of any type. The result is boolean (integer type with values 0 and 1).

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

The value is 1 if *y* is not tolerably equal to *x*, and 0 if it is.

Additional Error Report

If there is no parse or value error (see "Common Error Reports"), then:

- a valence error is reported if the left argument is missing.

Examples

```
    ' '≠'this is it'
1 1 1 1 0 1 1 0 1 1
    (<2 3, 4+1e-13)≠(2 3 4;'abcde';⍳5 6)
0 1 1
    1 2 3 ≠ '123'
1 1 1
```

Or *y*∨*x*

Arguments and Result

The arguments are simple numeric arrays of restricted whole numbers. The result is an integer array.

Definition

If *x* and *y* have boolean values (0 or 1) then *y*∨*x* is the Logical Or of *x* and *y*. That is:

`1∨1` equals `1∨0` equals `0∨1` equals `1`;
`0∨0` equals `0`.

Or is strictly boolean, never bitwise. All nonzero restricted whole numbers are treated as if they were 1.

To get bitwise behavior, use the Bitwise operator.

Additional Error Reports

If none of the common error conditions is reported, then, with a domain report preempting a type report:

- a domain error is reported (by Cast, actually) if the left argument is Null;

- a type error is reported if an argument is not a restricted whole number, unless the arguments are suitable for Cast.

Examples

```
    0 0 1 1∨0 1 0 1
0 1 1 1
    43∨14
1
```

Power *y⋆x*

Arguments and Result

The arguments and result are simple numeric arrays. The result is always floating point.

Definition

y to the power *x*. `10⋆2` is exactly equal to `1e2` but in general there is a very slight (tolerable) difference between `10⋆`*N* and `1e`*N*, because logarithms are used except in this special case, whereas `1e`*N* is exact.

Example

```
    2⋆0 .5 1 2 3 4 5 6 7 8 1025
1 1.414213562 2 4 8 16 32 64 128 256 Inf
```

Residue y|x

Arguments and Result

The arguments and result are simple numeric arrays. For two nonempty arguments, the result is integer if both arguments are integer, and floating point otherwise. If exactly one argument is empty, the result is floating point if that argument is floating point, and otherwise its type is the type of the nonempty argument. If both are empty, then if one is floating point and the other integer the result is floating point, and otherwise its type is the type of the right argument.

Dependency

Comparison tolerance, if an argument is in floating point (see "Comparison Tolerance").

Definition

y|x is the remainder when *x* is divided by *y*. `0|`*x* equals *x*. If *y* is nonzero, then *y|x* is *x-y×⌊x÷y*, in accordance with the mathematical definition of modular arithmetic, except as follows. If *x* is tolerably equal to *n×y*, where *n* is a whole number not necessarily representable by type `` `int``, then the result is 0. (So *Inf|x* and *¯Inf|x* are always 0.)

Examples

```
    100 | 1930 1941 1952 1978, 100+1e-12
30 41 52 78 0
    1.4 1.4 ¯1.4 ¯1.4 | 3.7 ¯3.7 3.7 ¯3.7
0.9 0.5 ¯0.5 ¯0.9
```

Subtract *y-x*

Arguments and Result

The arguments and result are simple numeric arrays. For two nonempty arguments, the result is integer if both arguments are integer and all result elements lie inside the range of integer representation, and floating point otherwise. If exactly one argument is empty, the result is floating point if that argument is floating point, and otherwise its type is the type of the nonempty argument. If both are empty, then if one is floating point and the other integer the result is floating point, and otherwise its type is the type of the right argument.

Definition

y minus *x*.

Example

```
     ¯1 0 99.5 1e308 - .5 ¯1 .5 ¯1e308
 ¯1.5 1 99 Inf
```

6. Nonscalar Primitive Functions

As noted earlier, the term *integer* indicates here a domain of values and a particular internal representation. The term *restricted whole number* refers to the same domain of values when both integer and floating-point representations are allowed. The floating-point representations need only be tolerably equal to integers.

Classification of Nonscalar Primitive Functions

Although (except for Beam and Time) they are listed alphabetically in this chapter, for convenient reference, the A+ nonscalar primitive functions can be grouped, among many other ways, in these eight categories:

- informational functions: Shape, Count, Depth, Type;
- structural functions: Reshape, Interval, Restructure, Reverse, Rotate, Transpose, Transpose Axes, Ravel, Item Ravel, Enclose, Disclose, Rake, Raze, Partition, Take, Drop, Replicate, Expand, Catenate, Laminate;
- selection functions: Pick, Choose, Bracket Indexing, Right, Left, Null, Separate Symbols;
- computational functions: Matrix Inverse, Solve, Deal, Pack, Unpack, Encode, Decode;
- comparison functions: Match, Member, Find, Bins, Grade up, Grade down, Partition Count;
- format and representation functions: Format, Default Format, Cast;
- specificational functions: Assignment, Append, Bracket Indexing Selective Assignment, Replace All, Selective Assignment; Beam;
- evaluative, display, and control functions: Execute, Execute in Context, Protected Execute, Value, Value in Context, Result, Signal, Print, Stop; Time.

Common Error Reports

Multiple errors elicit but one report. If an error report in the following list is issued, then the ones preceding it do not apply. Five reports are common to all nonscalar primitive functions:

- parse: this error class includes valence errors that result from three or more arguments in braces;
- value: an argument has no value;
- nondata: an argument is a function or some other nondata object;
- wsfull: the workspace is currently not large enough to execute the function in; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;
- interrupt (not an error): the user pressed **c** twice (once if A+ was started from a shell) while holding the **Control** key down.

Except where noted, inappropriate omission or inclusion of a left argument results not in a valence error report but in the invocation of the function or operator that shares the function symbol.

Function Definitions

Assignment `x←a` and `(x)←a`

Arguments and Result

The right argument `a` is any array or function expression (see "Function Expressions"). The left argument is the *target* of the assignment. `x` represents a valid user name. The explicit result is equal to `a`; it is not displayed in an Emacs session, unlike other explicit results, unless it is the result of an expression group.

Definition

The explicit result is the value `a` if `a` is an array, and otherwise a function expression identical to `a`. The side effect is to assign the value or function expression `a` to the object `x`. For a function expression, note that it is the function that is assigned to the target, not the name of the function. See also "Selective Assignment".

The two forms of Assignment are equivalent when used outside function and operator definitions, but have different meanings inside if `x` is an unqualified name. Inside a function or operator definition, any appearance of `x←a` means that `x` will be a local variable, while, in the absence of that form, the form `(x)←a` can be used to assign a value to the global variable `x`, whose context will be the context in which the function was defined.

Assignment, in any form, cannot be the operand of an operator.

Strand Assignment

Several *ordinary* assignments can be incorporated into one by means of strand assignment. For example:

```
     (a;c.b;f;d;d)←('ABC';10+⍳3 4;+.×;2;5)
```

Each item of the right argument is disclosed and assigned to the corresponding name on the left. Note that `f` becomes a function scalar in this example, not a function, and that `d` ends up with the value 5. After all the individual assignments have been made, the saved values for any dependencies among the targets are marked valid. Strand assignment is important when working with dependencies. See the discussion of commit and cancel for sets of dependencies in "Cyclic Dependencies". Strands are not permitted as targets of Selective Assignments.

The left argument must appear in strand notation. The right argument, therefore, must be a vector (of the same length as the left argument) or a scalar, but, since it is its *value* that is used, it can appear in any form, and, since Disclose accepts a simple argument, it can be a simple vector.

This example demonstrates that the righthand side is evaluated right to left and then the assignments are made left to right:

```
      a←1;  b←2;
      f{s;d;i;p;c;v}:↓v
      (a;b)←(b ⊣ `a`b _scb¨ <(f;); a←3)
 `a
 `b
```

Strand Assignment does not trigger a screen update until the interpreter has made the assignments for *all* the variables involved. This is done in order to avoid problems which could occur when several columns of a displayed table object, including the first, are being updated, with changed lengths.

Value and Value in Context Assignment

Both Value and Value in Context can be used in targets, viz., `(c%x)←a` and `(%x)←a`, where `c` and `x` are any expressions producing simple scalar symbols. The form `(k%¨y)←a` can also be used; it has the same effect as Strand Assignment, but is more flexible, since the statement does not involve a fixed number of variables with fixed names. The latter form cannot be combined with Selective Assignment in any way. See the Strand Assignment discussion regarding the timing of screen updates.

The context for the object of a Value Assignment is always the current context, so if the Assignment is in a function, the context is the one in which the function was called or the one which the function set before executing the Value Assignment. The use of these functions in assignment is important for working with callback functions; see the "Callback Functions" chapter.

Assignment of Objects Bound to Display Classes

The assignment primitive `x←a` is affected by display class bindings. If `a` is bound but `x` is not, `x` does not inherit the class of `a` or any of its attributes. If `x` is bound then `a` must be in the domain of the class of `x`, i.e., it must be possible to bind `a` to the class of `x`. This rule applies to Selective Assignment as well.

Additional Error Report

The following report is issued only if there is no parse or value error (see "Common Error Reports"):

- an invalid error is reported if `x` is bound to a display class and the value it is to be given is not valid for objects bound to that display class - see `s.VERIFY` in the "s-Context Parameters (Global Variables)" table regarding the message that is issued.

Example

The following illustrates successful and rejected assignments for objects bound to display classes:

```
      $load s
      a←"Spec test"
      `a is `label
      b←"any character vector"    ⍝ A valid label value.
      a←b                         ⍝ Okay.
      b←⍳2 3                      ⍝ An invalid label value.
      a←b                         ⍝ Specification fails.
 ←: invalid
*     →
```

Bins `y⍋x`

Arguments and Result

The left argument is a simple numeric or character array; its items must be in ascending order (no duplicates) for a meaningful result. (Ascending order has the Grade-up meaning: items compared lexicographically, i.e., leading elements in their ravels compared first, without comparison tolerance, and following elements compared only in the case of ties; and alphabetic characters sorted in accordance with their ASCII codes, which are shown in "Graphic Characters for Atomic Vector".) The right argument, `x`, is of the same general type as `y`, and its `¯1+⍴⍴y` trailing axes must have length `1↓⍴y`. The result is an array of integers whose shape is defined by the A+ expression `(-0⌈(⍴⍴y)-1)↓⍴x`.

Definition

The idea is to partition `x` into cells of the same shape as the items of `y` and then to find where each cell falls among those items. Specifically, partition `x` in the same way as for Find: viz., into cells of rank `0⌈(⍴⍴y)-1`. For each such cell, the result has one element, whose value is the number of items of `y` which the cell is greater than, in the sense described in the "Arguments and Result" subsection.

For a numeric vector `y`, each element of the result indicates the subinterval of `y` into which the corresponding element of `x` falls, without using comparison tolerance. That is, if `i#r` is an element of the result `r`, its value can be determined from the "Bins for Numeric Vectors" table.

Bins for Numeric Vectors

Value of `i#r`	**Condition (comparisons without tolerance)**
`0`	`(i#x)≤y[0]`
`j`, where `0<j` and `j<#y`	`y[j-1]<i#x` and `(i#x)≤y[j]`
`#y`	`y[(#y)-1]<i#x`

One use for Bins is preparing data for a bar graph.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if an argument is not simple or if the arguments are not of the same general type;

- a rank error is reported if the rank of *x* is less than that of the items of *y*, so the partitioning fails;
- a length error is reported if the items of *y* and the cells into which *x* is partitioned have different shapes.

Examples

```
     ¯1 0 1 ⍋ 0.3 ¯0.3 ¯2 .1 1 5
 2 1 0 2 2 3

     v←?5000⍴1000      ⍝ sample set of random numbers
⍝ Put in cells; count how many in each cell, showing distribution of the
random numbers:
     +/((100×1+⍳10)⍋v)∘.=⍳10
 494 480 477 533 506 472 521 512 524 481
⍝ An equivalent but slightly faster way; shows distribution of the set:
     ⊃+/¨(<(100×1+⍳10)⍋v)=¨⍳10
 494 480 477 533 506 472 521 512 524 481
```

Bracket Indexing *x*[*a*;*b*;**…**;*c*]

Arguments and Result

x is any array, and *a*, *b*, ... , *c* are simple arrays of restricted whole numbers, or absent or the Null. The number of semicolons must be less than the rank of *x*, or zero. Roughly speaking, the shape of the result is the shape of *x* with the shapes of *a*, *b*, ... , *c* substituted for the corresponding dimensions.

More precisely: If *x* is a scalar, the form is *x*[ ], and the result is *x*. Otherwise, the shape of the result is determined as follows. Let *Ra* be ⍴*a*, *Rb* be ⍴*b*, etc., except that if the *i*th one, *d*, is absent or the Null, let *Rd* be *i*#⍴*x*. If there are *k*-1 semicolons, the shape of the result is *Ra*,*Rb*, ... ,*Rc*,*k*↓⍴*x*.

Definition

Bracket Indexing is a way of selecting elements from an array. Semicolons separate indices for different axes. The result has the shape described above. For each axis, a corresponding argument can give an array of indices. For an axis for which no argument or a Null argument is given, the vector 0,1, ... ,*n*-1 is used, where *n* is the length of that axis. Elements are selected by taking one index from each index array. The order of selection is determined by running through the index arrays in odometer fashion: all of the rightmost for the first combination of the rest, then all of the rightmost for the second combination of the rest, and so on. Duplicate selections are permitted.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- rank: the number of unenclosed semicolons within the brackets is equal to or greater than the rank of *x* and *x* is nonscalar, or *x* is a scalar and there is something (anything) between the brackets;

- type: a member of the index group (*a*, *b*, ... , or *c*) is not a simple array of restricted whole numbers;
- maxrank: the rank of the result would be greater than nine;
- index: a number in the index group is negative or not less than the length of the axis it indexes.

Examples

```
     (⍳2 3 4)[1;2;3]
 23
     'adrv'[2 3⍴0 0 2 3 0 2]
aar
var
     (⍳5 5 5)[0;⍳2;⍳3]
 0 1 2
 5 6 7
     (⍳5 5 5)[0;2 5⍴0 1;0]
 0 5 0 5 0
 5 0 5 0 5
     (`a `b;5 1 9;<{+})[1]
<  5 1 9
     ⍴(⍳2 3 4)[⍳0;'';()]
 0 0 4
```

Cast `y∨x`

Arguments and Result

`y` is a scalar symbol and `x` is any simple character or numeric array or the Null.
The result has the type indicated by `y`. (Since the Null is already an empty symbol vector, if `x` is the Null then `∨`sym∨x` is `` `null``.)
The shape of the result is:
`⍴x` if `x` and the result are neither or both symbolic; else
`¯1↓⍴x` if `y` is `` `sym``; and otherwise
`(⍴x),n`, where `n` is the number of characters (excluding `` ` ``) in the representation of the "longest" symbol in `x`.

Definition

The value of `y` is one of the symbols `` `char `int `float`` or `` `sym`` (but see the last paragraph of the additional error reports section, below). The result is `x` with each element converted to the type specified by `y`.

Any character, integer, floating-point, or simple symbol array can be converted to any of these four types.

In the case of floating point to integer, the elements of `x` are rounded. For integer to character, `256|x` is used. The conversion is based, of course, on the character codes used in the implementation, namely ASCII.

Related functions: Pack and Unpack convert between symbol and character. Format and Default Format convert numbers to characters and employ IEEE rounding; they also convert symbols, and Default Format handles function scalars. Floor and Ceiling round in specific directions to whole numbers and may perform type conversions. Fix Input and its variants and Execute convert characters to numbers. Bitwise Cast leaves the data part of the variable unchanged but changes the type indicator and (usually) shape.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"), and a type error is reported only if a domain error is not:

- a domain error is reported if either argument is nested or *y* is Null or contains more than one element;
- a type error is reported if the arguments specify an impermissible conversion (but see note below) or *y* is not a symbol, unless the arguments are suitable for the function Or.

Examples

```
      `char∨97      ⍝ Return character.
a
      `int∨3.4 4.5 5.6
 3 5 6
      `sym∨'abcd'
 `abcd
```

Catenate `y,x`

Arguments and Result

The arguments *x* and *y* are any arrays of the same general type, conforming in shape as described here:

- If *x* is a scalar then *y* can be any shape, and the shape of the result is `(1+#y),1↓ρy`. Similarly if *y* is a scalar.
- If the rank of *x* is one less than the rank of *y*, then the shape of *x* must equal the shape of the items of *y*, and the shape of the result is `(1+#y),ρx`. Similarly if the rank of *y* is one less than the rank of *x*.
- If the rank of *x* is equal to the rank of *y*, then the items of *x* and *y* must have the same shape, and so do the items of the result: `1↓ρx` is equal to `1↓ρy` and the shape of result is `((#x)+#y),1↓ρx`.

The result has the same type as:

- the floating-point argument, if the other one is integer; else
- the left argument, if neither argument is empty; else
- the nonempty argument, if exactly one argument is empty; else
- the right argument.

Definition

If *x* and *y* are the same rank, the items of *y* and the items of *x* are joined in the result. That is, item *i* of *y* equals item *i* of the result and item *j* of *x* equals item `j+#y` of the result.

If the arguments differ in rank by one, the argument of lower rank is treated as though it had an additional leading axis of length one. If one argument is a scalar, it is treated as though it had been reshaped to have the shape of the items of the other argument with an additional leading axis of length one.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the arguments are not of the same general type;
- a rank error is reported if neither argument is a scalar and their ranks differ by more than 1;
- a length error is reported if either their ranks are equal and the shapes of their items are not the same or their ranks differ by 1 and the argument of lesser rank is not the same shape as the items of the other.

Examples

```
    (⍳2 3),(100+⍳4 3)        ⍝ Same rank.
  0   1   2
  3   4   5
100 101 102
103 104 105
106 107 108
109 110 111
    'Treasury ','note'       ⍝ Same rank.
'Treasury note'


   (⍳3),(100+⍳4 3)          ⍝ Ranks differ by 1.

  0   1   2

100 101 102

103 104 105

106 107 108

109 110 111



    3,⍳2 3 4        ⍝ A scalar argument is extended to the size of an item

 3   3   3   3

 3   3   3   3

 3   3   3   3



 0   1   2   3

 4   5   6   7

 8   9  10  11
 12  13  14  15

 16  17  18  19

 20  21  22  23
```

Choose *y#x*

Arguments and Result

y is either a simple array of restricted whole numbers, or a nested scalar or vector whose elements are enclosed, simple arrays of restricted whole numbers, or the Null. *x* is any array. If *y* is nested then `⍴y` is less than or equal to `⍴⍴x`. The result has the same type as *x* if *x* is numeric or character, and otherwise the same general type.

Definition

If *x* is scalar, *y* must be the Null, and the result is *x*. Assume *x* is nonscalar for the rest of the definition.

If *y* is simple the result is

```
      x[y;;...;]
```

In particular, if *y* is Null the result equals *x*. If *y* is not Null the shape of the result is `(⍴y),1↓⍴x`.

If *y* is nested, the result is

```
      x[0⊃y;1⊃y;...;(¯1+#y)⊃y]
```

If `i⊃y` is Null then it is treated as if it were `⍳(⍴x)[i]`. Bracket Indexing treats any omitted trailing axes in a similar fashion, so the result can also be written as

```
      x[0⊃y;...;(¯1+#y)⊃y;⍳(⍴x)[#y];...;⍳(⍴x)[¯1+⍴⍴x]]
```

If no element of *y* is Null, the shape of the result is `(⊃⍴¨y),(⍴y)↓⍴x`.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a rank error is reported if (1) the left argument is nested and of length greater than the rank of the right argument, or (2) the right argument is a scalar and the left argument is other than a simple Null;
- a type error is reported if the left argument (1) is not a simple array or a nested scalar or vector of enclosed, simple arrays, or (2) contains a simple scalar which is not a restricted whole number;
- a maxrank error is reported if the rank of the result would be greater than nine;
- an index error is reported if an element of the left argument is not an appropriate index for the corresponding axis of the right argument, i.e., it is negative or equal to or greater than the length of that axis.

Examples

```
      2 0 # 3 3⍴'abcdefghi'      ⍝ Row 2 and row 0.
ghi
abc
```

```
     (0;0 2) # 3 3⍴'abcdefghi'  ⍝ Row 0, columns 0 2.
ac
     (;0) # 3 3⍴'abcdefghi'     ⍝ All rows, column 0.
adg
     (1;0) # ⍳2 3 4             ⍝ Plane 1, row 0, all columns.
 12 13 14 15
```

Count *#x*

Argument and Result

The argument is any array. The result is a scalar integer.

Definition

The result is the number of items of *x*. If *x* is a scalar the result is 1, while if *x* is a nonscalar the result is the length of the leading axis, i.e., `(⍴x)[0]` (see "Shape").

Examples

```
     #⍳5000
 5000
     #168 3 4⍴0
 168
     #,12
 1
     #12
 1
```

Deal *y?x*

Arguments and Result

Both *x* and *y* are simple one-element numeric arrays of nonnegative restricted whole numbers, with *y≤x*. The result is a vector of integers of length *y*.

Dependency

The value of the Random Link system variable, `` `rl ``, which is changed each time a random integer is chosen.

Definition

The result is a vector of integers of length *y*, chosen at random from *⍳x* without duplication, or, in mathematical terms, without replacement. The result is dependent upon the random link, `` `rl ``, which is set when the system command `$rl` is executed and each time a random integer is chosen.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports"), and a domain error report is issued only if a type error report is not:

- a type error is reported if an argument is nested or has an element that is not a restricted whole number;
- a domain error is reported if either argument has a negative element or has more than one element, or if the left argument exceeds the right argument.

Example

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

Decode `y⊥x`

Arguments and Result

`y` is a simple numeric scalar or vector, and `x` is any simple numeric array. If `y` has more than one element, then `#y` must equal `#x`. The result is a simple numeric array whose shape equals `1↓ρx` and whose type is integer if the arguments are of type integer and every element of the result can be faithfully represented that way and otherwise floating point.

Definition

If `x` and `y` are vectors with the same number of elements, the result is the evaluation of `x` in the number base system with radices `y[0]`,`y[1]`, ... , `y[¯1+#y]`. If `y` has one element it is treated as if it were `(#x)ρy`. If `x` is a matrix, each element `i#r` of the result `r` is the evaluation of the column `x[;i]` in the number base system represented by `y`. More generally, if `x` has rank greater than 2, each element of the result is the evaluation of the corresponding vector along the first axis of `x`, in the number system represented by `y`.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if either argument is not a simple numeric array;
- a rank error is reported if the left argument is not a scalar or vector;
- a length error is reported if `#y` is not equal to either 1 or `#x`.

Examples

```
    ⍝ Convert 2 hours, 5 minutes, and 59 seconds to seconds.
    24 60 60⊥2 5 59
7559
    ⍝ Present value (price) from interest rate and cash flows.
    cf←0 100 100 100 100 1000
    7.2⍕ pv ← (÷1+i) ⊥ ⌽cf ⊣ i←.05
1138.12
    ⍝ Present value at various interest rates.
    8.2⍕ pv ← (÷1.03+.01×⍳5) ⊥@0 1 ⌽cf
1234.32 1184.92 1138.12 1093.77 1051.71
```

Default Format `⍕x`

Argument and Result

`x` is a simple numeric or a simple character array, a simple symbol scalar, a function scalar, or a simple symbol array. The result is a simple character array of rank less than or equal to 2.

Dependency

For numeric *x*, the value of Printing Precision, `` `pp``.

Definition

If *x* is numeric or character, the result is an array containing *x* in default format. That is, if the expression *x* is entered in an A+ session, a display of the value of *x* appears; this primitive function captures that display in its result. In particular, if *x* is a character array of rank less than or equal to 2, the result equals *x*. If *x* is numeric, the number of digits shown and the format (fixed or exponential) used are dependent on the printing precision system variable `` `pp``. A nonscalar symbol array displays the symbols as described in the next paragraph, and for arrays of rank 2 or greater, are padded with blanks on the right to match the length of the longest (unpadded) symbol involved.

If x is a scalar symbol, the result is a character vector, giving the display of that symbol, with a blank preceding the backquote.

When *x* is a function scalar, the result is a character vector. If *x* is primitive, the result is its symbol. If *x* is a defined function, the result is its definition. If the definition has more than one line, the lines are separated by newlines in the result. If the function is derived (e.g., `⍳¨`), then the result is the string `*derived*`. (Note that Default Format shows that expressions such as `+.×` are not derived functions strictly speaking.)

Additional Error Reports

If there is no parse or value error (see "Common Error Reports"),

- a rank error is reported if the argument is enclosed (nested or function scalar) and is not a scalar; and otherwise
- a type error is reported if the argument is not a simple character or numeric array, a simple symbol scalar, or a function scalar.

Examples

```
     _gsv `pp
 10
     ⍝ The Printing Precision is now 10. Digits after an e don't count. If
there would
     ⍝ be too many digits in ordinary notation, e-notation is used in a
display.
     '(',(⍕○1),')',⍕12345.678912e24 1234567891 12345678912
( 3.141592654) 1.234567891e+28 1234567891 1.234567891e+10
     ⍝ A function definition, to be shown by Default Format.
     cube x:x*3
     'cube fn → ',⍕<{cube}
cube fn → cube x:x*3
     ⍝ Indentation of first line of definition is always ignored by Default
Format.
```

Depth `≡x`

Argument and Result

The argument is any array; if it is a function expression it must be in braces. The result is a scalar integer.

Definition

The result is the maximum depth of nesting in the argument. All simple arrays (and so all empty arrays) are depth 0, except for function expressions, whose depth is -1. (The Enclose of a function expression, which is called a function scalar, is simple and depth 0.) The depth of a nested scalar is 1 plus the depth of the disclosed scalar (see Disclose, below). The depth of a nonscalar nested array is the greatest of the depths of its elements.

Examples

```
     ≡'abc'
0
     ≡{+}            ⍝ The parser needs the braces as a hint
¯1                   ⍝ when two functions are juxtaposed.
     ≡(2;3;<(4;5))   ⍝ Three enclosings: strand, Enclose, strand.
3
```

Disclose `>x`

Argument and Result

The argument `x` is a simple array, a nested scalar, or a uniform nested array (see Definition, Case 3, for the meaning of uniform). Letting `s` and `t` be the shape and type of the first element of `x` with that element's top level of nesting (if any) removed, the shape of the result is `(⍴x),s` and its type is floating point if some element of `x` is a nonempty floating-point array after its top level of nesting (if any) is removed, else its type is `t`.

Definition

Case 1. Simple Array `x`

If `x` is a simple array, perhaps a simple scalar, then the result is `x`. Note that function scalars are simple arrays; to turn a function scalar `fnsc` into the corresponding function you can use `fnsc¨` or `%_name{fnsc}`.

Case 2. Nested Scalar `x`

If `x` is a nested scalar, then its depth is at least one. The result is `x` with the top level of nesting removed, and it may or may not be a scalar.

Case 3. Nested Vector `x`

The elements of `x` must all be nested - i.e., each element of `x` must have a depth of at least one (remember that function scalars have depth zero) -, and the Discloses of all these elements must have the same shape and the same general type. Arrays with this property are called *uniform* in this definition.

Disclose of a nested vector `x` is defined in terms of Disclose of a nested scalar, as follows. The array `>x` has depth one less than the depth of `x`. The number of items of `>x` equals the number of elements of `x`, i.e., `#x`. For each index `i` of `x`, the `i`th item of `>x` equals the Disclose of the nested scalar `i#x`, i.e., `>i#x`.

Case 4. Nested Array `x`

Disclose of a nested array *x* is defined in terms of Disclose of a nested vector, as follows. As in the vector case, *x* must be uniform. Let *s* denote the common shape of all the Discloses of the scalar elements of *x*. Then `>x` is `((ρx),s)ρ>,x`. That is, ravel *x*, apply Disclose to this vector, and reshape the disclosed vector to have *x*'s shape for its leading axes and the shape of a disclosed element for its trailing axes.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a maxrank error is reported if the result would have more than nine dimensions;
- a domain error is reported if *x* contains both nested and nonnested elements other than symbols and function scalars;
- a rank error is reported if the Discloses of the elements of *x* have different ranks;
- a mismatch error is reported if the Discloses of the elements of *x* have different shapes;
- a type error is reported if the Discloses of the elements of *x* have different general types.

Example

```
     >(ι3;10×ι3)
 0  1  2
 0 10 20
```

Drop `y↓x`

Arguments and Result

The argument *y* is a simple one-element array whose value is a restricted whole number, and *x* is any array. The shape of the result equals the shape of the right argument *x* along all but the first axis, while the length of the first axis is the larger of `#x` minus the absolute value of *y* and `0`, i.e., `0⌈(#x)-|y`.

Definition

The result is *x* without its first *y* items if *y* is nonnegative, or its last `-y` items if *y* is negative. If `|y` is greater than `#x` then the number of items in the result is 0.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports"), and nonce only if there is no type error:

- a type error is reported if the left argument is not simple or has an element that is not a restricted whole number;
- a nonce error is reported if the left argument has more than one element.

Examples

```
     3↓ι5 2
 6 7
 8 9
 ¯4↓'15 January'
15 Jan
```

Enclose `<x`

Argument and Result

The argument is any array; if it is a function expression, it must be enclosed in braces, to aid the parser. The result is a nested scalar unless the argument is a function expression, in which case it is a simple scalar.

Definition

The result is a scalar that contains the argument `x`. The depth of the result is the depth of the argument plus one. If `f` is a function expression, `<{f}` is a function scalar. Note that strand notation is equivalent to the concatenation of the Enclose of each of its components.

Examples

```
     2ρ<ι3
<   0 1 2
<   0 1 2
          ⍝ Functional notation required for + .
     (≡`sym),≡{+},(≡'abc'),(≡2 3 4),(≡5.5),≡()
 0 ¯1 0 0 0 0
          ⍝ Strand encloses each element.

     ≡@0(`sym;+;'abc';2 3 4;5.5;)
 1 0 1 1 1 1
          ⍝ Depths increased by Enclose.
     ≡@0(<`sym;<{+};<'abc';<2 3 4;<5.5;<())
 2 1 2 2 2 2
```

Encode `y⊤x`

Arguments and Result

`y` is a simple numeric vector or scalar and `x` is any simple numeric array. The result has shape `(ρy),ρx`.

Definition

For positive elements of `x`, the result is the representation of that element, as far as is possible, in the number base system with radices `y[0]`,`y[1]`, ... ,`y[¯1+#y]`. That is, `y⊤x` contains the `#y` low-order "digits" of its representation in this number base for each positive element of `x`. For a negative element `n`, the result is the representation, in the same way, of `(×/y)|n`. No matter what the signs of the elements of `x` are, `y⊥y⊤x` equals `(×/y)|x`. Thus Decode is the left inverse of Encode for any element `p` for which `(×/y)|p` is equal to `p`.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"), and a rank error report is issued only if a type error report is not:

- a type error is reported if either argument is not a simple numeric array;
- a rank error is reported if the left argument is not a vector or scalar.

Examples

```
     24 60 60 ⊤ 7559
 2 5 59
     100 ⊤ 12345
 45
```

Execute ⍎x

Argument and Result

x is a character vector or scalar. The result is the explicit result of the A+ expression in x, except that it is the Null when the last function executed in x is a Specification (ordinary or selective), and when the argument is a system command or a definition of a function, operator, or dependency.

Definition

This function evaluates x as an A+ expression. It cannot be used to establish the target of an Assignment. See also "Execute in Context or Protected Execute", "Value", and "Value in Context". It can be used to execute a system command. It can be traced by $dbg xeq.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the argument is not simple or is not of character type;
- a rank error is reported if the rank of the argument exceeds 1;
- any error from the execution of the argument is reported (and execution is suspended); entering y produces a domain error for Execute and a second suspension. See example.

Examples

```
     100×⍎'1 1.23 4.567'
 100 123 456.7
     ⍎'a←14 23 34'
          ⍝ Result is Null. (⍎'a')←14 23 34 not allowed.
     a
 14 23 34
     ⍎"(⍳10)⍴0"
⍝[error] ⍴: maxrank
*     →
⍝[error] ⍎: domain
```

Execute in Context or Protected Execute *y*⍎*x*

Arguments and Result

y is a symbol or an integer or Null and *x* is a character vector or scalar. The result is the explicit result of the A+ expression in *x* under circumstances controlled by *y*, except that it is null when the last function executed is any Specification, and when *x* is a system command or a definition of a function, operator, or dependency.

Definition

If `y` is a symbol, then this function is **Execute in Context** and evaluates `x` as an A+ expression in the context `y`. If `c` is the context when the function is invoked, it is equivalent to `{_cx y;⍎x;_cx c}`, except that the explicit result is that of `⍎x` rather than `_cx c`. Note: an unqualified name `z` in `x` remains unqualified; it is not treated as a qualified name; `x` is just executed in the context `y`. In particular, the name of a local variable `z` in `x` remains unqualified. See "Execute", "Value", and "Value in Context".

Execute in Context is useful when one wants to create multiple instances of a dependency or dependencies, distinguished by contexts. You can create a dependency by, for instance,

```
    `cxt⍎'a:b+c'
```

and if you have a list `c` of contexts, then, for example,

```
    c⍎¨<'a:b+c'
```

creates a whole set of dependencies in different contexts.

If the left argument is an integer or Null, the function is **Protected Execute**, in the current context (cf. "Do - Monadic (Protected Execution)"). If the execution fails, the result is the error code, as listed in the table "Error codes for Protected Execution", as a simple integer. There is no suspension - i.e., execution proceeds as if no error has occurred. (Actually, there are a *few* cases, such as an attempt to give a bound variable an impermissible value, that do result in a suspension, with a `stop` error message.)

If the execution is successful, the result is Enclose applied to the result of Execute for `x`, viz., `<⍎x`. Enclose is used to enable you to distinguish between the two cases.

If the Protected Execute Flag, `` `Gf ``, is 0, however, a suspension will occur if Protected Execute encounters an error in its right argument; entering → clears the suspension and produces the error code as the result. (This behavior is different from that of an ordinary Execute: clearing a suspension in the execution of its argument causes a second suspension, on the Execute itself, and clearing that yields a null result.) See `` `Gf `` and `$Gf`. It can be traced by `$dbg xeq`.

Messages that are not strictly A+ error messages will still appear in the log, e.g.

```
filename: No such file or directory
not an `a object
```

and many Adap messages.

The treatment of input errors while stopped in protected execution is described under monadic do.

Warning! Within a Protected Execute - as in a protected do - Result exits from Protected Execute only, with a 0 return code and the Result argument as result; it does not exit from the function containing the Protected Execute.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the left argument is not a simple symbol or integer or Null, or if the right argument is not a simple character array;
- a rank error is reported if the rank of the right argument exceeds one;
- if the left argument is a symbol or if the protected execute flag, `` `Gf ``, is zero, any error from the execution of the argument is reported (and execution is suspended).

Example

```
      try.x←⍳3
      ⍎'x÷10'         ⍝ try.x has a value, but x does not.
 .x: value
*       →
 ⍎: domain            ⍝ Error gets passed on to Execute.
*       →
      0⍎'x÷10'        ⍝ Protected execute (`Gf is 1).
 4                    ⍝ Numeric code for the value error.
      `try⍎'x÷10'     ⍝ Execute in Context okay; uses try.x.
 0 0.1 0.2
                      ⍝ Protected execute using the var that has a value:
      0⍎'try.x÷10'
<   0 0.1 0.2         ⍝ Success shown by an enclosed result.
```

Expand `y\x`

Arguments and Result

`y` is a simple scalar or vector of restricted whole numbers whose elements are all 0 or 1, and `x` is any array.

Definition

The number of ones in `y` equals the number of items in `x`, i.e., `+/y` equals `#x`, or `x` is a scalar. If the `i`th item of `y` is 1, then the `i`th item of the result is the `(+/(i+1)↑y)`th item of `x`, or `x` if it is a scalar. If the `i`th item of `y` is 0, then the `i`th item of the result is composed entirely of fill scalars. The table "Fill Elements" shows the fill scalars.

Additional Error Reports

Each of the following reports is issued only if none of the reports preceding it on this list applies and there is no parse or value error (see "Common Error Reports") - except that token precedes value:

- a token error (like a parse error) is reported if there is a primitive function or operator symbol to the immediate left of `\` that is not one of `+×⌈⌊∧∨`;

and, unless there is one of `+×⌈⌊∧∨` to the immediate left of `\` (that is, unless you are thrown into Scan, a monadic operator):

- a valence error is reported if there is no argument to the immediate left of `\`;
- a type error is reported if the left argument is nested or does not consist of restricted whole numbers;
- a domain error is reported if any element of the left argument is not 0 or 1;
- a rank error is reported if the left argument is not a scalar or vector;

- a length error is reported if the number of ones in the left argument is unequal to the number of items in the right argument, unless the right argument is a one-element array.

Examples

```
     1 0 1 1\⍳3 4
 0   1   2   3
 0   0   0   0
 4   5   6   7
 8   9  10  11
     1 0 1\'a'
a a
```

Find `y⍳x`

Arguments and Result

The arguments are any arrays (including scalars) of the same general type. The rank of the items of y must not exceed the rank of x, and the `¯1+⍴⍴y` trailing axes of x must have length `1↓⍴y`. The result is an array of integers whose shape is defined by the A+ expression `(-0⌈(⍴⍴y)-1)↓⍴x` - i.e., the shape of x with the last s dimensions omitted, where s is the rank of the items of y.

Dependency

Comparison tolerance, if an argument is in floating point.

Definition

Partition x into cells of rank `0⌈(⍴⍴y)-1`. The result has one element for each such cell. The value of that element is the index of the first item of y to which the cell is identical - in the sense of Match, i.e., at all levels the cell and the item are the same shape and type (except that integer and floating point match here) and all their simple scalar components, at whatever level, are tolerably equal. If the cell is not found among the items of y, the resulting element is `#y`. A scalar y is treated as a one-element vector.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the arguments are not of the same general type;
- a rank error is reported if the rank of x is less than that of the items of y, so the partitioning fails;
- a length is reported if the items of y and the cells into which x is partitioned have different shapes.

Examples

```
     (4 3⍴'fatbatcathat')⍳(2 3⍴'catpat')
 2 4
     (+;-;×;÷)⍳<{×}
 2   ⍝ Strand makes function expression a function scalar.
```

Format `y⍕x`

Arguments and Result

`y` is a simple numeric scalar or vector, and `x` is a simple numeric or symbol array. If `#y` is greater than 1 then it must equal `¯1↑⍴x` - i.e., `#x` if `x` is a vector, and the number of columns if the rank of `x` is greater than 1. The result is a simple character array of rank less than or equal to 2.

Definition

The result is an array containing `x` in formatted form, with the format controlled by `y`. Negative numbers are formatted with an ordinary minus sign (`-`). Elements of `y` specify the appearance of certain elements in `x`:

- if `#y` equals 1, then the element of `y` applies to all elements of `x`;
- if `#y` is greater than 1, then element `i#y` applies to element `i#x` if `x` is a vector, to column `x[;i]` if `x` is a matrix, and to columns `x[...;i]` otherwise.

The format of every element of `x` is controlled by an element of `y`, which is of the form [`-`]*width*[`.`*digits*], where the *digits* specification is expected to be a *single* digit and *any digits following it are ignored.* The total number of characters in the format is *width.* If *digits* is present, it specifies the number of digits to appear to the right of the decimal point; if it is 0 or absent, the element is shown as an integer. If the minus sign is present, the format is in exponential format (e-notation).

Understanding how rounding works

Because Format is often used as a means of rounding numbers to some desired precision —and because that rounding often creates questions about the accuracy of the rounding— this seems to be a good place to discuss how rounding works, and why it is that we sometimes see some unexpected results.

There is nothing within A+ that imposes any special rules on the rounding of numbers. Rounding is done entirely using IEEE symmetric rounding rules. That is, if a number lies halfway between two other numbers to which it can legitimately be *equally* rounded either way, it is rounded to the one ending in an *even* digit. Therefore, a value of 22.5, for instance, could logically be rounded to either 22 or 23; the IEEE rules tell us that it should be rounded to 22, because that is the number which ends with an *even* digit. As further examples, if we are starting with values which are exactly halfway between two integers, and rounding them to integers:

- 0.5 should round *down* to an integer value of 0 (not to 1, because 1 is an odd number);
- 1.5 should round *up* to an even integer value of 2;
- 2.5 should round *down* to an even integer value of 2;
- 3.5 should round *up* to an even integer value of 4; and
- 4.5 should round *down* to an even integer value of 4.

In A+ terms, this would be shown as follows:

```
     4⍕.5 1.5 2.5 3.5 4.5     ⍝ This example shows IEEE rounding.
   0   2   2   4   4
```

In actual practice, however, machine approximation of decimal numbers and machine rounding may obscure the regular IEEE rules.

For example, given the following vector:

```
    V←1.055 1.155 1.255 1.355 1.455 1.555 1.655 1.755 1.855 1.955
```

IEEE rules dictate that rounding it to two decimal positions *should* cause the "n.n55" values to round to "n.n6" (and if we had "n.n45" values, they should round to "n.n4"). However, that's not necessarily what happens. Let's explore what happens, and why.

Original Value	**IEEE Rounding**	`6.2⍕V` **Rounding**	**Actual Internal Value**
1.055	1.06	1.05	1.05499999999999994...
1.155	1.16	1.16	1.15500000000000003...
1.255	1.26	1.25	1.25499999999999989...
1.355	1.36	1.35	1.35499999999999998...
1.455	1.46	1.46	1.45500000000000007...
1.555	1.56	1.55	1.55499999999999994...
1.655	1.66	1.66	1.65500000000000003...
1.755	1.76	1.75	1.75499999999999989...
1.855	1.86	1.85	1.85499999999999998...
1.955	1.96	1.96	1.95500000000000007...

Notice that using `6.2⍕V` will round the values in `V` to two decimal places... but it doesn't necessarily give us what we expect from the IEEE rounding rules. Some of the results look okay, but realize that the internal values are imprecise for *all* of our values, and some of them just happen to round the way that we want them to.)

The reason for this is that we are doing decimal operations on a hexadecimal machine, so there will always be some errors in the internal representation of many of the numbers that we deal with every day. It is simply not possible to represent these values *exactly* on a digital machine.

In the same way that we simply cannot represent exactly one-third as a decimal number, we also cannot represent many other decimal numbers in hexadecimal and be able to convert them back to *exactly* the number that we started with. Therefore, a value of "1.055" actually gets represented within the machine as "1.05499999999999994...", so that value is *correctly rounded down* to "1.05", while a value of "1.155" is actually seen by the machine as "1.15500000000000003...", so that value is *correctly rounded up* to "1.16". Although the first value is rounded to "1.05" instead of the desired IEEE rounding value of "1.06", realize that the rounding is correct based upon the internal value that is seen by the hardware. These internal values are incorrect, but they are as close an approximation of the exact values as we can get on this hardware.

So why don't we see this imprecision at every step of our calculations? ...Simply because the printing precision normally masks this error, and over the course of normal work, these errors tend to cancel out.

Before you get *too* concerned about this imprecision, realize that the Comparison Tolerance in A+ (the amount by which two values may differ and still be considered to be equal) is set to 1e-13 (or 0.0000000000001); that is a tolerance of one part (of error) in ten trillion. To put this into perspective, if our measurements were to represent distance, a measurement of 250,000 miles (approximately the distance from the earth to the moon) could be carried out within an error of no more than one-third of the thickness of a piece of copier paper. For most operations, this is deemed to be "close enough."

Also realize that rounding of values through the use of the Format function is typically done just for display of final output (where 16 digits of precision would be inappropriate anyway). Calculations prior to that final rounding step are done at full precision.

Finally, we just want to emphasize again that none of the rounding that you see through A+ is any different than it would be in other environments. A point that may make it *seem* different is the ease with which you can adjust the precision and look at alternate views of the same values. This is a general hardware numeric conversion issue, not an A+ issue.

Creating the left argument to Format:

When a number has been formatted in the specified form the result is fitted in the specified width in one of two ways:

- exponential format: the result for a positive number is padded on the left with two blanks and for a negative with one, and this padded result is left justified within the width, after being truncated on the right if it is too long;
- otherwise: the result is right justified within the width, after being truncated on the right if it is too long.

Because these rules effectively handle inconsistent left arguments like `6.6`, no error message is given for them. It is up to you to see that the widths you specify are sufficient, allowing for the padding in exponential format and for any blanks you want between adjacent numbers not in exponential format.

If the right argument consists of symbols, they are formatted in their displayed form with backquotes removed. Each of them is right justified in the specified width, after being truncated if it is too long. Any *digits* or exponential-format specification is ignored.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"), and a length error is reported only if a type error is not:

- a type error is reported if the left argument is not simple and numeric, or if the right argument is not simple and either numeric or symbol;
- a length error is reported if the number of elements in the left argument is neither 1 nor equal to ¯1↑ρx.

Examples

```
    2 10.4 ¯12.2⍕2 3ρ1 ¯3.14159 123456 2 123.7 ¯55
1    -3.1416   1.23e+05
2   123.7000  -5.50e+01

    4⍕.5 1.5 2.5 3.5 4.5    ⍝ This example shows IEEE rounding.
  0    2    2    4    4

    16.12⍕1.123456789012    ⍝ 16.12 on the left is equivalent to
            1.1             ⍝ 16.1: the 2 is ignored.

    _gsv `pp
10                          ⍝ Printing Precision is 10

    12345.678912            ⍝ See that ordinary display obeys Printing
Precision:
 12345.67891                ⍝ 5 digits before the decimal point, so 5 (10-5)
digits after.

    14.7 ⍕ 12345.678912
 12345.6789120              ⍝ Format ignores Printing Precision and obeys the
left argument.
```

Grade down ⍒x

Argument and Result

The argument x is either a simple numeric array or a simple character array or a simple symbol array. The result is a vector of integers of length #x, i.e., of length equal to the number of items in x.

Definition

The result r is a permutation of the vector ⍳#x such that the items of r#x are in nonascending, lexicographic order. To determine whether or not one item is greater than or equal to another in the lexicographic sense, leading elements in their ravels are compared first, and only in the case of ties are the elements that follow compared. Alphabetic characters are sorted in accordance with their ASCII codes, which are shown in "Graphic Characters for Atomic Vector". The indices of equal items are in ascending order in r. That is, if i is less than j and r[i]#x is identical to r[j]#x, then r[i] is less than r[j]. Comparison tolerance is not used. If x is a symbol array, the result is the same as it would be for ⊤x.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports"), and a type error is reported only if a valence error is not:

- a valence error is reported if there is a left argument;
- a type error is reported if the argument is not a simple numeric or character array.

Examples

```
     ⍒ 10.2 6 999 0 6
 2 0 1 4 3
     (⍒a)#a←⍕`apl `a `apple
apple
apl
a
```

Grade up ⍋x

Argument and Result

The argument x is either a simple numeric array or a simple character array or a simple symbol array. The result is a vector of integers of length #x, i.e., of length equal to the number of items in x.

Definition

The result r is a permutation of the vector ⍳#x such that the items of r#x are in nondescending, lexicographic order. To determine whether or not one item is less than or equal to another in the lexicographic sense, leading elements in their ravels are compared first, and only in the case of ties are the elements that follow compared. Alphabetic characters are sorted in accordance with their ASCII codes, which are shown in "Graphic Characters for Atomic Vector". The indices of equal items are in ascending order in r. That is, if i is less than j and r[i]#x is identical to r[j]#x, then r[i] is less than r[j]. Comparison tolerance is not used. If x is a symbol array, the result is the same as it would be for ⍕x.

Additional Error Report

The following report is issued only if there is no parse or value error (see "Common Error Reports"):

- a type error is reported if the argument is not a simple numeric or character array.

Examples

```
     ⍋ 10.2 6 999 0 6
 3 1 4 0 2
     (⍋a)#a←⍕`apl `a `apple
a
apl
apple
```

Interval ⍳x

Argument and Result

The argument is a simple scalar or vector of nonnegative restricted whole numbers. The result is an array of integers whose shape is x if x is a vector, or ,x if x is a scalar.

Definition

If the argument x is a scalar or one-element vector, the result is the vector of integers from 0 to x-1. If the argument x is a vector with 2 or more elements, the result is the vector of integers from 0 to (×/x)-1, reshaped to shape x: viz., x⍴⍳×/x (see "Reshape").

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the argument is not simple or an element of it is not a restricted whole number;
- a rank error is reported if the argument is not a scalar or a vector;
- a domain error is reported if the argument has a negative element;
- a maxrank error is reported if the argument has more than nine elements.

Examples

```
    ⍳5
0 1 2 3 4
    ⍳2 3
0 1 2
3 4 5
```

Item Ravel `!x`

Argument and Result

The argument `x` is any array of rank at least 2. The result has shape `(×/2↑⍴x),2↓⍴x` and the same type as the argument.

Definition

The result is `((×/2↑⍴x),2↓⍴x)⍴x`, i.e., the ravel of the first two axes of `x` becomes the first axis of the result.

Additional Error Report

The following report is issued only if there is no parse or value error (see "Common Error Reports"):

- a rank error is reported if the argument is a scalar or vector.

Example

```
    !⍳2 2 4
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

Laminate `y~x`

Arguments and Result

Either `x` and `y` have equal shapes, or one is a scalar. They also must have the same general type. If `x` is nonscalar, the shape of the result is `2,⍴x`; otherwise, the shape of the result is `2,⍴y`.

Definition

The result `r` always has two items. If `x` and `y` have equal shapes then item `r[0]` equals `y` and item `r[1]` equals `x`. If `x` is nonscalar and `y` is a scalar then item `r[0]` equals `(⍴x)⍴y` and item `r[1]` equals `x`. If `x` is scalar and `y` is a nonscalar then item `r[0]` equals `y` and item `r[1]` equals `(⍴y)⍴x`.

The result has the same type as:

- the floating-point argument, if the other one is integer; else
- the left argument, if neither argument is empty; else
- the nonempty argument, if exactly one argument is empty; else
- the right argument.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the arguments are not of the same general type;
- a rank error is reported if neither argument is a scalar and their ranks differ;
- a length error is reported if neither argument is a scalar and their shapes differ;
- a maxrank error is reported if the rank of either argument is 9.

Example

```
     1 2 3~10 20 30
  1  2  3
 10 20 30
```

Left `y⊣x`

Arguments and Result

`y` and `x` are any arrays. The result is the same shape and type as `y`.

Definition

The result is identical to the left argument. Left can be used to execute two expressions when you would only discard the explicit result of the first one anyway.

Examples

```
     2 3⊣'abc'
 2 3
⍝ Evaluate a dependency to keep its last value, then remove its definition:
     _undef{`x} ⊣ %`x
     a[⍋a] ⊣ a←0 2 4 5 1 3 7 6
 0 1 2 3 4 5 6 7
     ((a⍳a)=⍳⍴a)/a ⊣ a←'keep only the unique characters'
keponlythuiqcars
```

Match `y≡x`

Arguments and Result

The arguments can be any arrays, including function arrays. The result is an integer, either 1 or 0.

Dependency

Comparison tolerance, if an argument is in floating point.

Definition

The result is 1 if at all levels *y* and *x* are the same shape and type (except that integer and floating point match here), and all their simple scalar components, at whatever level, are tolerably equal. The result is 0 otherwise.

Examples

```
      'abc'≡'a','b','c'
 1
      ''≡⍳0
 0
      (⍳2 4)≡'no match'
 0             ⍝ Unlike Equal to, never an error; always 0 or 1.
```

Matrix Inverse ⌹*x*

Argument and Result

x is a simple numeric array of rank less than or equal to 2 - if 2, then at least as many rows as columns. The result has shape `⌽⍴x`.

Definition

If *x* is a nonsingular matrix with the same number of rows as columns, then the result is its matrix inverse. If *x* is a matrix with more rows than columns, and if the columns are linearly independent, the result is the unique left matrix inverse of *x*. That is, `(x)+.×x` equals an identity matrix, but `x+.×x` may not. If *x* is a vector the result is `,((⍴x),1)⍴x`. If *x* is a scalar the result is `÷x`.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"):

- a rank error is reported if the rank of the argument exceeds 2;
- a domain error is reported if (1) the argument is not a simple numeric array, (2) the argument is a matrix with more columns than rows, or (3) the argument is singular or very ill-conditioned.

Example

```
        ⌹2 2⍴ 1 2 3 4
 ¯2     1
  1.5 ¯0.5
```

Member `y∊x`

Arguments and Result

The arguments are any arrays of the same general type. The rank of the items of `x` must not exceed the rank of `y`, and the trailing `¯1+ρρx` trailing axes of `y` must have length `1↓ρx`. The result is an array of integers whose shape is defined by the A+ expression `(-0⌈(ρρx)-1)↓ρy`.

Dependency

Comparison tolerance, if an argument is in floating point.

Definition

Partition `y` into cells of rank `0⌈(ρρx)-1`. The result has one element for each such cell. The value of that element is 1 if at all levels the cell and at least one of the items of `x` are the same shape and type (except that integer and floating point match here) and all their simple scalar components, at whatever level, are tolerably equal - i.e., if Match yields 1 for the cell and some item of `x`. The result is 0 otherwise.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the arguments are not of the same general type;
- a rank error is reported if the rank of `y` is less than that of the items of `x`, so the partitioning fails;
- a length error is reported if the items of `x` and the cells into which `y` is partitioned have different shapes.

Examples

```
     (123+1e-10 1e-11)∊123,1950+⍳50
 0 1
     1992 1991 1870 1992 2001∊123,1950+⍳50
 1 1 0 1 0
     (2 3ρ'catpat')∊(4 3ρ'fatbatcathat')
 1 0
```

Null `⊣x`

Argument and Result

`x` is any array.

Definition

The result is Null, i.e., `()`, the empty symbol vector, whose type is `` `null``.

Examples

```
     ⊣'junk'
     ()≡⊣9 10
 1
```

Pack `⊥x`

Argument and Result

`x` is a simple character array. The result is an array of symbols with shape `¯1↓ρx`.

Definition

If `i#r` is an element of the result `r` then `i#x` is a character vector along the last axis of `x`. Item `i#r` is the symbol that, when displayed, looks exactly like the character vector `i#x` except for the leading `` ` ``. Trailing blanks in `i#x` are ignored, but included blanks are allowed. The null character, `` `char∨0``, is used by the interpreter to delimit symbols, so never try to include it in a symbol: it will exclude any characters following it from the symbol.

Additional Error Report

The following report is issued only if there is no parse or value error (see "Common Error Reports"):

- a type error is reported if the argument is not a simple character array.

Example

```
    ⊥2 3ρ'a  abc'
 `a `abc
```

Partition `y⊂x`

Arguments and Result

`y` is a simple array of restricted whole numbers, and `x` is any array. The result is a nested vector whose length is `⌈(#x)÷y` if `y` consists of one element and that element is not zero, and otherwise a nested array of the same shape as `y`.

Definition

The elements of `y` are nonnegative. If `y` has one element and that element is not zero, then the first element of the result is Enclose of the first `y` items of `x`, the second element is Enclose of the next `y` items of `x`, and so on, until there are fewer than `y` items of `x` remaining. If there are no remaining elements, then the result is complete, and otherwise the last element of the result is Enclose of the remaining items of `x`.

If `y` has more than one element or is 0, then the result has the same shape as `y`, and the first element of the ravel of the result is Enclose of the first `(,y)[0]` items of `x`, the second element is Enclose of the next `(,y)[1]` items of `x`, and so on until `y` is exhausted. If for some element of the result there are fewer items remaining in `x` than `y` specifies for that element, then those remaining items of `x` are enclosed to form that element, and any remaining elements of the result are empty. On the other hand, not all items of `x` need appear in the result.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"), and a domain error is reported only if a type error is not:

- a type error is reported if the left argument is nested or does not consist of restricted whole numbers;
- a domain error is reported if the left argument contains a negative number.

Examples

```
     3 2 2 1⊂⍳6 3
<   0  1  2
    3  4  5
    6  7  8
<    9 10 11
    12 13 14
<   15 16 17
<
     3⊂⍳5
<   0 1 2
<   3 4
     (⊂' '=x)⊂x←' this is'     ⍝ See Partition Count.
<   this
<   is
```

Partition Count ⊂x

Argument and Result

x is a simple vector or scalar of restricted whole numbers. The result is a numeric vector of integers.

Definition

In general, *x* is a vector or scalar of restricted whole numbers, but in the usual case it is a vector composed of zeros and ones. All nonzero numbers are treated alike, and the description assumes ones for simplicity.

The first element of *x* must be 1. The length of the result is the number of ones in *x*, i.e., +/*x*, and each element of the result corresponds to a 1 in *x*. The value of an element is the length of the subvector consisting of the 1 it corresponds to and all the zeros preceding the next 1. Put another way, the elements of the result are the lengths of the contiguous sequences of zeros starting at each 1 in *x*, plus 1.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports"), and none of the reports preceding it on this list applies:

- a type error is reported if the argument is nested or does not consist of restricted whole numbers;
- a rank error is reported if the argument is not a scalar or vector;
- a domain error is reported if the argument begins with a zero.

Examples

```
     ⊂1 0 0 0 1 0
 4 2
     ⊂1 0 0 0 1 1 0 1
 4 1 2 1
```

Pick `y⊃x`

Arguments and Result

The left argument `y` is the Null, or a simple scalar or vector of integers or symbols, or a nested scalar or vector whose items are simple scalars or vectors of integers. The right argument `x` is any array.

Definition

All permitted combinations of arguments are considered by turns.

Any valid right argument and empty left argument (`⍳0` or the Null)

An empty vector `y` picks all of `x`. (**Warning!** A possible point of confusion: although Pick acts like Choose and Bracket Indexing for the Null, selecting all, for `⍳0` it differs from those two functions, picking all whereas they select none.)

Scalar right argument

The result is the same as if the right argument were a one-element vector; if the left argument is not empty, it must be 0.

Slotfiller right argument and simple symbol left argument

A symbol is used to pick from a slotfiller, an array of the form `(s;v)`, where `s` is a scalar or vector of symbols and `v` is any nested scalar or vector of the same length as `s` (see "Subtypes and Supertypes"). If `y` has one element and is identical to the `i`th element of `s`, i.e., `i#s`, (and not to any previous element - Pick does not check that the elements of `s` are distinct) then `y⊃(s;v)` is defined and equals `>i#v`. If `x` is a slotfiller composed of a pair of scalars, ``(`sym;<array)`` for example, and `y` equals `0⊃x`, then `y⊃x` is defined and equals `>1⊃x`.

If `x` is a nested slotfiller and `y` a simple vector of symbols (a *path vector*), the result is `y[¯1+#y]⊃...⊃y[1]⊃y[0]⊃x` where the scalar Picks are as just defined. Note that the order of the elements in the left argument corresponds to the depth of the selection from the right argument: element 0 in the left argument refers to the top level, element 1 refers to the second level, and so on.

Simple right argument and simple numeric left argument of length 1

A simple array can be picked from only when `y` is `⍳0` or Null or a scalar or one-element vector. That is, there must be only one step to the Pick, and either a single item or all of `x` must be picked. An empty vector picks all of `x`, as stated above, and for a scalar or one-element vector and a simple right argument, Pick is the same as Choose:

```
     y⊃x  ←→  y#x.
```

If `y` is a scalar or one-element vector, then the rank of `x` must not exceed 1. For example,

```
   ()⊃1      ←→  1
   ()⊃1  2   ←→  (⍳0)⊃1  2  ←→  1  2
```

`1⊃1 2 3 ←→ 2`
`0⊃⍳2 3` is a rank error.

Nested vector right argument and simple left argument

If `y` has a single element, `x` is a vector and the result is the Disclose of the (scalar) `y`th element of `x`, i.e.,

```
>y#x.
```

A vector `y` is called a *path vector*, and `y⊃x` (for a simple `y`) reaches into `x` with a series of Picks that select scalars and Disclose them, using one element of `y` at each step:

```
y[¯1+#y]⊃...⊃y[1]⊃y[0]⊃x.
```

The order of the elements in the left argument corresponds to the depth of the selection from the right argument: element 0 in the left argument refers to the top level, element 1 refers to the second level, and so on. The argument `x` and the result of each but the last scalar Pick must be a vector, since the left argument of each step of Pick is a scalar. Picking from a simple array is not allowed as a continuation step: a simple array can be produced only as the final step; `1 0⊃(1;2 3;4)` is an error.

Nested right argument and nested left argument

If `y` is nested, its items are used in a way similar to the previous case and must each select a scalar, but the rank of `x` and any of its components can exceed 1. If `y` is scalar and `x` is of rank 2 or greater (to distinguish this case from the previous one), `y` must contain a full set of indices for `x`, as an enclosed simple vector. The result is the Disclose of a single element chosen from `x`, viz.,

```
>(<@0>y)#x
```

(A small complexity in this expression arises from the fact that Pick, in general picking a single element during each of several steps, requires all its indices for each depth to be in one box, whereas Choose, in general choosing several elements or cross sections but only at the top level, requires the indices for each axis to be in one box.) If `y` is a vector, then `y⊃x` reaches into `x` with a series of scalar Picks of the same form as just shown for the simple case, but now each element of `y` is an enclosed simple vector and the corresponding component of `x`, from which `y` selects, need not be a vector.

Picking from a simple array is not allowed as a continuation step: a simple array can be produced only as the final step; `(1;0)⊃(1;2 3;4)` is an error.

By the way, `((j;i)⊃a)←b` is a valid Selective Assignment, although `(i⊃j⊃a)←b` is not.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"):

- a type error is reported if `y` is not composed of symbols or integers (not just restricted whole numbers);
- a rank error is reported if (1) `y` or anything it contains is not a vector or a scalar, or (2) one of its items has a length unequal to the rank of the array to which it is applied;

- a domain error is reported if (1) *y* is symbolic and *x* is not (more or less) a slotfiller array, or (2) during the execution of Pick and not as its last step, an array is picked that is not nested;
- an index error is reported if an element of *y* is not in the range 0 to n-1, where n is the length of the axis it is applied to, or if *y* is a symbol and is not a member of the vector of symbols *s* in *x*.

Examples

```
     1⊃('ab';(2 2⍴'cdef';10 20))
< cd
  ef
<  10 20
     0⊃1⊃('ab';(2 2⍴'cdef';10 20))
cd
ef
     1 0⊃('ab';(2 2⍴'cdef';10 20))
cd
ef
     (1;0;0 1)⊃('ab';(2 2⍴('c';'d';'e';'f');10 20))
d
     `was⊃(`this `was `that;(10;20;30))
 20
     `this `was `that ⍳`was
 1
     1⊃(10;20;30)
 20
     `is⊃(`this `was `that;(10;20;30))
 ⊃: index
*    →
     `a⊃(`a;<10)
 10
```

Print ⍎x

Argument and Result

x is any array and its value is also the result.

Definition

The result is *x*, but additionally the value of *x* is displayed in the A+ session.

Example

```
     2+⍎3
 3
 5
```

Rake ∊x

Argument and Result

x is any array. The result is a nested vector whose depth is one, or a function scalar or the Null.

Definition

If *x* is the Null, so is the result. If *x* is simple and not the Null, the result is ,<x. Otherwise, the result is a vector composed of the simple components of *x*, except that Nulls are discarded. The simple components of *x* are all those simple objects obtainable by repeated selection and

disclosure. Each of these components is enclosed in the result, except that scalars of type symbol and function that were at depth zero remain so.

Examples

```
     ∊`a `b
<   `a `b   ⍝ A vector whose only element is of
            ⍝ depth 1 and, when disclosed, length 2.
     ∊`a`b,(+;'ac';1 2;(3 4;(5;<();););6);7 8)
<   `a
<   `b      ⍝ Despite the display,
<   +       ⍝ the first three elements are simple: `a `b <{+}
<   ac      ⍝ This and all the rest of the elements are each at depth 1.
<   1 2
<   3 4
<   5       ⍝ Notice that two Nulls were discarded following this element.
<   6
<   7 8
```

Ravel `,x`

Argument and Result

The argument `x` is any array. The result is a vector of shape `×/⍴x`.

Definition

The result is the vector of the elements in `x`, taken in row-major, or odometer, order. For a scalar, it is a one element vector.

Example

```
     a
 101    2
   3 ¯45
  34   21.5
     ,a
 101 2 3 ¯45 34 21.5
```

Raze `⊃x`

Argument and Result

The argument is either a simple array or a nested scalar or vector. If it is a nested vector, then the Discloses of its elements must all be conformable: they must be of the same general type and, treating scalars as one-element vectors, their ranks and the shapes of their items must be same.

The type of the result depends upon the Discloses of the elements of `x`. If one of them is a nonempty floating-point array, the result type is floating point. Otherwise, if any of them are nonempty, the result type is that of the first nonempty one; else it is the type of the first one.

The result shape is `⍴x` if `x` is simple, `⍴>x` if `x` is a nested scalar, `⍴>''⍴x` if `x` is a nested one-element vector, and otherwise `(+/>@0#¨x),s`, where `s` is the common shape of the items of the disclosed elements of `x`.

Definition

If `x` is simple, then the result equals `x`. If `x` is a nested scalar, the result is `>x`. If `x` is a one-element nested vector, then if that one element is an enclosed scalar, the result is `,>x`, and otherwise the result is `>x`. If `x` is a nested vector with more than one element, the result is the catenation of the Discloses of all elements of `x`, i.e.,
`(>0#x),(>1#x),...,>(¯1+#x)#x`
Note in the latter case that `x` is in the domain of Raze only if (1) this series of catenations can be formed and (2) the Discloses of all the elements are either all the same rank or a mixture of scalars and vectors.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a domain error is reported if some elements of `x` are simple and some elements are nested;
- a rank error is reported if the Discloses of the elements of `x` have different ranks, unless they are all either vectors or scalars;
- a mismatch error is reported if the items of the Discloses of the elements of `x` have different shapes;
- a type error is reported if the Discloses of the elements of `x` have different general types.

Examples

```
     ⊃('ab';'cde';'f')
 Abcdef




     ⊃(ι2 3;10×ι2 3)
  0  1  2
  3  4  5
 10 11 12
 13 14 15
```

Replicate `y/x`

Arguments and Result

`y` is a simple vector or one-element array of nonnegative restricted whole numbers, and `x` is any array. If `y` is a vector of length unequal to one, then its length equals the number of items of `x`, i.e., `#x`, or `x` can be a scalar. The items of the result are items of `x`.

Definition

The items of the result are taken from the items of `x`. Each item `i#x` (or `x` itself if it is a scalar) is replicated `y[i]` times in the result. A scalar or other one-element array `y` is treated like `(#x)ρy`, i.e., each item of `x` is replicated `y` times in the result. The number of items in the result is `+/(#x)ρy`. When every element of `y` is 0 or 1, this function is called Compress.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports"), except token precedes value, and none of the reports preceding it on this list applies:

- a token error (like a parse error) is reported if there is a primitive function or operator symbol to the immediate left of / that is not one of +×⌈⌊∧∨;

and, unless one of +×⌈⌊∧∨ is to the immediate left of /, throwing you into Reduce, a monadic operator:

- a valence error is reported if there is no left argument to the immediate left of /;
- a type error is reported if the left argument is nested or does not consist of restricted whole numbers;
- a domain error is reported if the left argument contains a negative number;
- a rank error is reported if the left argument is not a vector or a one-element array;
- a length error is reported if the number of elements in the left argument equals neither one nor the number of items in the right argument, unless the right argument is a scalar.

Examples

```
     3 0 1/3 2⍴'abcdef'
ab
ab
ab
ef
     (a≥0)/a←4 ¯1 3 0 8 ¯5      ⍝ Compress.
 4 3 0 8
     2 0 3 0/'a'
aaaaa
```

Reshape y⍴x

Arguments and Result

The argument y is any simple array of nonnegative restricted whole numbers, and x is any array. The result is an array whose shape is ,y and whose type is that of x except in certain cases when the result is empty (see the definition, below).

Definition

The value is an array whose elements come from ,x (see "Ravel") in ascending index order. The number of elements of x needed for the result is ×/,y. If x has at least ×/,y elements, any excess is ignored. If x is nonempty but has fewer than ×/,y elements, the elements of x are used cyclically. If x is empty, the resultant array is filled with zeros if x is numeric, blanks if x is character, or enclosed Nulls if x is nested, as shown in the table "Fill Elements".

If y is all zeros, the result is of course empty, and its type is the type of x if x is character, integer, floating point, or null, but its type is null if x is box, symbol, or function.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if y is not simple or has an element that is not a restricted whole number;
- a domain error is reported if y has a negative element, or if `×/,y` is too large to be represented as an integer - i.e., is not a restricted whole number;
- a maxrank error is reported if y has more than nine elements.

Examples

```
      2 3ρ'abcdefgh'
abc
def
      10ρ'abcdefgh'
abcdefghab
```

Restructure `y!x`

Arguments and Result

The left argument `y` is a simple one-element array whose value is a restricted whole number, and the right argument `x` is any array, except that it must be nonscalar unless `y` is 1. The result has rank 1 greater than the rank of `x`. Except for the first two axes of the result and the first axis of `x`, the shapes of the result and `x` are equal. There are two quite different cases.

- If `y` is positive, it must evenly divide `#x`, and the result has shape `((#x)÷,y),(,y),1↓ρx`.
- If `y` is negative or zero, the result has shape `((,y)+1+#x),(-,y),1↓ρx`.

Definition

In both cases of this function the result is obtained by arranging the items of `x` in ways that replace the first axis of `x` with two new axes. In the case where `y` is positive it must evenly divide `#x`, and then the result is simply a reshape of the leading axis of the `x` into two axes, the first of length `(#x)÷y`, and the second of length `y`. See the first example.

In the case where `y` is negative or zero, the elements of `x` are used repeatedly. The result `r` can be thought of in terms of clipping sections of the first axis of `x` using a window of length `-y`. That is, if `w←ι-,y` then `0#r` is `w#x`, `1#r` is `(w+1)#x`, and in general `i#r` is `(w+i)#x`. See the second example. This form of the function is useful for producing moving averages.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the left argument is not simple or has an element that is not a restricted whole number;
- a nonce error is reported if the left argument has more than one element;
- a rank error is reported if the right argument is scalar and the left argument is not 1;
- a maxrank error is reported if the right argument is of rank 9;
- a length error is reported if the left argument is positive and does not evenly divide the number of items in the right argument, or if it is negative and more than one greater in absolute value than the number of items in the right argument (for negative `y`, if `|y` equals `#x` the result has one item, and if it equals `1+#x` the result has zero items).

Examples

```
     3!'abcABC'
abc
ABC


     ¯3!0 1 2 3 4 5 6
 0 1 2
 1 2 3
 2 3 4
 3 4 5
 4 5 6
     +/@1⊢¯3!0 1 2 3 4 5 6
 3 6 9 12 15
     (+/@1⊢¯3!0 1 2 3 4 5 6)÷3 ⍝ Moving average of length 3
 1 2 3 4 5
```

Result ←x

Argument and Result

x is any array and its value is also the result.

Definition

The effect of Result is to exit from the current function or immediate execution code and return *x* as the result. (In immediate execution, this value will not be displayed if what the A+ display mechanism believes was the last function executed is thought by it to be an Assignment **-** i.e., the display mechanism may not recognize the exit or it may take the Result arrow to be an Assignment arrow.) The niladic use of this symbol, a bare left arrow (←), is meaningful only outside function definitions, where its effect is to cause resumption of the most recently suspended function execution, with the workspace size increased if necessary; within a function definition it is ignored. Thus, within a function, `←()` causes an exit and the return of a Null result, whereas ← alone has no effect at all.

Warning! Within protected execution, whether do or Execute, Result exits from that function only, with a 0 return code and the Result argument as result; it does not exit from the function containing the protected execution text.

Example

```
     fact n: {if (n=0) ←1; n×fact n-1}
     fact 5
 120
     fact 0
 1
```

Reverse ⌽x

Argument and Result

x is any array. The shape of the result equals the shape of *x*.

Definition

The result is *x* with the items reversed. `(⌽x)[i]` is `x[(¯1+#x)-i]`.

Examples

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

Right ⊢*x*

Argument and Result

x is any array. The result is the same shape and type as *x*.

Definition

The result is the value of the argument. This primitive is useful in separating the right operand from the right argument in an expression, and also in getting just the value of a mapped file.

Additional Error Report

If there is no parse or value error (see "Common Error Reports"),

- a valence error is reported if there is a left argument.

Examples

```
      ⊢a←⍳3     ⍝ Continuation beyond Assignment, so the result is displayed.
 0  1  2
                ⍝ Separate right argument from data operand using ⊢ :
      2  4  8,⍤0⊢.5 .25 .125
 2  0.5         ⍝ Items of the result are the scalar cells
 4  0.25        ⍝ of the arguments joined by Catenate.
 8  0.125
```

Rotate *y*⊖*x*

Arguments and Result

x is any array. *y* is a simple array of restricted whole numbers whose shape equals 1↓⍴*x* unless *y* consists of a single element (and is then of any rank). The shape and, for simple *x*, the type of the result equal the shape and type of *x*.

Definition

Suppose *x* is a matrix and *y* is a vector with ⍴*y* equal to 1↓⍴*x*. Then the result is *x* with each column vector *x*[;*i*] rotated by the number of elements indicated in *y*[*i*]. If *y*[*i*] is positive, then *x*[;*i*] is rotated towards the origin, while if *y*[*i*] is negative, *x*[;*i*] is rotated away from the origin. If *y*[*i*] equals 0, there is no rotation.

If *x* is a vector and *y* is a scalar (or other one-element array), then the result is *x* rotated by the amount *y*, as described above. If *x* is a matrix and *y* has one element, then the result is *x* with each column vector *x*[;*i*] rotated by *y*. The case for *x* of higher rank is defined by reshaping *x* into a matrix with all but the first axis combined into one. Formally, the result is:

```
      (⍴x)⍴(,y)⊖((1↑⍴x),×/1↓⍴x)⍴x
```

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if `y` is nested or does not consist of restricted whole numbers;
- a rank error is reported if `y` has more than one element and its rank is not 1 less than that of `x`;
- a length error is reported if `y` has more than one element and `⍴y` is unequal to `1↓⍴x`.

Example

```
      0 ¯2 1⌽⍳5 3
 0 10  5
 3 13  8
 6  1 11
 9  4 14
12  7  2
```

Selective Assignment `target←a`

Arguments and Result

The right argument `a` is any array. The left argument is the *target* of the assignment, and takes one of the forms in the table "Targets of Selective Assignment". The compatibility requirements for the two arguments are discussed in the definition. The explicit result is `a`, except that its type will be that of the selected subarray (see Definition) if they are both numeric but one is integer and the other floating point.

Definition

See also Assignment. Like it, Selective Assignment, in any form, cannot be the operand of an operator.

Each of the various forms of the left argument (see the table "Targets of Selective Assignment"), if executed separately from Selective Assignment, produces a subarray of some target array that appears in the left argument. Executed in Selective Assignment, it selects a subarray of locations in the target array. The effect of Selective Assignment is to replace the values of the target array at the locations specified in the selected subarray by the values in `a`.

The conformability rule for Selective Assignment is that the shape of the selected subarray must be identical to the shape of the right argument `a` unless `a` has only one element.

The type rule is that the selected subarray and `a` must be of the same general type, with one additional requirement: if the type of the selected subarray is integer and `a` is floating point, then `a` must consist entirely of restricted whole numbers, so that it can be coerced to integers before insertion in the selected subarray.

If the same location appears repeatedly in the selected subarray, then in general several distinct values in `a` will thereby be specified for that one location. The element of `a` that is actually chosen as the new value for that location is the one with the highest index in `,a`. See the examples.

A description of each basic form of Selective Assignment follows.

Performance Note: Assignment In Place

Choose Assignment, `(...#x)←y`, and the corresponding Bracket Assignments, `x[...]←y`, are the only assignments guaranteed to be done in place - i.e., without copying `x` to a new location while revising it. Append Assignment, `x[,]←y`, is performed in place if there is enough space. All other Selective Assignments, like ordinary Assignment to an existing variable, involve copying the target variable while revising it.

Bracket Indexing Selective Assignment

Bracket Indexing can be used to select subarrays as values. It can also be used, in the same form, on the left side of the Assignment arrow to select subarrays of locations for assignment of new values.

Replace All (`x[]←a`)

This is a special form of Bracket Indexing Selective Assignment; it would be just a particular case except that it applies to scalars also. It replaces all the elements in the variable named on the left of the assignment arrow. It is the most efficient way to replace all elements of a mapped file; see "Mapped Files".

Append (`x[,]←a`)

This is a special form of Bracket Indexing Selective Assignment for appending items to an array. `x` must not be a scalar, else a rank error is reported. The conformability rule is that the shape of the items of `x` must be identical to the shape of either `a` or its items, except that `a` can be a scalar for any `x` (and, as with any scalar extension, a scalar `a` can "disappear" - be extended to empty - if the items of `x` are empty).

Execution of `x[,]←a` can be more efficient than `x←x,a` because if there is enough unused space following `x` in the storage area allocated to `x`, the result is formed simply by copying `a` into that space. Because storage is allocated roughly in powers of two, an array has between exactly and twice the space it needs. In particular, this form of assignment is the only efficient way to extend a mapped file; see "Mapped Files".

Append can cause an itemwise invalidation of an itemwise dependency. See "Recognition of Itemwise Changes".

Additional Error Report for Append

- A length error is reported if there is a preset callback on an Append Assignment and the callback function changes the shape of the data to be appended, so that it no longer agrees with the indices that were passed to the callback function as an argument.

Targets of Selective Assignment

Target Type	Target Form (see below regarding `f` and see below regarding `,` (Ravel))			
monadic	`(f x)`	`(,x)`	`(f p⊃x)`	`(,p⊃x)`
dyadic	`(y f x)`	`(y#,x)`	`(y f p⊃x)`	`(y#,p⊃x)`
general	`f{...;x}`	`#{y;,x}`	`f{...;p⊃x}`	`#{y;,p⊃x}`
index expression	`x[a;...;c]`	`(,x)[a]`	`(p⊃x)[a;...;d]`	`(,p⊃x)[a]`

choose	`((a;...;c)#x)`	`(a#,x)`	`((a;...;d)#p⊃x)`	`(a#,p⊃x)`
append	`x[,]`			
replace all	`x[]`			

Warning! The forms `(f,x)`, `(f,p⊃x)`, `(y f,x)`, `(y f,p⊃x)`, `f{...;,x}`, and `f{...;,p⊃x}` are accepted, but unless `f` is Choose (`#`) the Ravel (`,`) is ignored!

In this table, `f` denotes either a primitive or a defined function, appropriate to the particular form - or `f` can be omitted. `x`, the *target array*, denotes a variable name or `%y` or `c%y`, where `c` and `y` are any expressions producing simple scalar symbols, but `x` *cannot* denote `%¨v` or `c%¨v`. The variable named by x is the one that is modified.

Primitive Functions in Selective Assignment Expressions

Some A+ primitive functions can be used in expressions on the left of the assignment arrow. They are:
Take, Drop,
Replicate, Expand,
Ravel, Item Ravel, Reshape,
Rotate, Reverse,
Transpose, Transpose Axes.
(By the way, `((j;i)⊃a)←b` is a valid Selective Assignment, although `(i⊃j⊃a)←b` is not.)

For a monadic primitive function `f` in the above list, the form of the specification is, in the simplest case:

```
     (f x)←a
```

The definition is as follows: first evaluate

```
      i←f ιρx
```

where `x` represents a variable name, and then do the Bracket Indexing Selective Assignment:

```
     (,x)[i]←a
```

This definition is based on the following facts:

- the elements of `ιρx` are exactly all the indices of `,x`;
- for each monadic primitive `f` listed above, `∧/(,f ιρx)∈,ιρx` has the value 1. That is, all elements of `f ιρx` are valid indices of `,x`.

The definition for a dyadic primitive function is similar. In this case the right argument `x` of the primitive function, which must represent a variable name, is the target of the assignment and the left argument `y` is any array for which `y f x` is valid. Then

```
     (y f x)←a
```

is defined by

```
     i←y f ιρx
     (,x)[i]←a
```

Defined Functions in Selective Assignment Expressions

Defined functions can also be used on the left side of the assignment arrow. The definition is the same as the one above for primitive functions, only now it is up to the programmer to see that the function, when applied to arrays whose values are in `⍳n`, produces results whose elements are all members of `⍳n`.

Brace Assignment (`{w;y;...;z;x}←a`)

The general form `f{w;y;...;z;x}←a` has a syntactic special case, where the function name is not present:
`{w;y;...;z;x}←a`
is permitted and is equivalent to `x[]←a`. In particular, `{x}←a` is equivalent to `x[]←a`.

Selective Assignment of Objects Bound to Display Classes

See "Assignment of Objects Bound to Display Classes".

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- an error is reported if one of the functions selecting the subarray encounters an error; this includes any defined function, so the report may come from a primitive function within a defined function;
- a type error is reported if the general type of the target differs from the general type of `a` or if the target is integer but `a` is floating point and not composed entirely of restricted whole numbers;
- a rank error is reported if the rank of the selected subarray differs from the rank of `a` and `a` is not a one-element array;
- a length error is reported if corresponding dimensions of `a` and the selected subarray differ and `a` is not a one-element array;
- a maxitems error is reported if an attempt is made to append an item to a mapped file beyond its maximum size as specified by _items (see "Items of a Mapped File");
- an invalid error is reported if the target array is bound to a display class and the value it is to be given is not a valid value for (the selected part of) objects bound to that display class;
- an index error is reported if an index not in `⍳⍴x` is generated; the error may not be detected before partial replacement of values has occurred - see the examples.

Examples

```
    v←⍳10
    v[2 2 2]←5 9 ¯1  ⍝ Repeated indices
    v
0 1 ¯1 3 4 5 6 7 8 9

    m←⍳4 4
    0 0⍉m            ⍝ Diagonal
0 5 10 15


    (0 0⍉m)←1 2 3 4  ⍝ Primitive function selection: diagonal.
    m
 1  1  2  3
 4  2  6  7
 8  9  3 11
12 13 14  4
```

```
          ⍝ Display argument, then return last element (to show how
          ⍝ defined functions are called during Selective Assignment):
     last x: ¯1↑,⍞x
     a←3 2⍴'abcdef'
     last a
ab
cd
ef
f


     (last a)←'F'  ⍝ User-defined function selection.
 0 1               ⍝  Argument it was given: ⍳⍴a, i.e., ⍳3 2
 2 3
 4 5               ⍝ It returns 5.


     a
ab
cd
eF                 ⍝ The new value 'F' was placed at (,a)[5].


     x←(10;20;(⍳2 3;'abc'))
     (1;1)#2 0⊃x   ⍝ Pick-Choose selection
 4                 ⍝ Present value of selected component


     ((1;1)#2 0⊃x)←99   ⍝ Pick-Choose selection for
     x                  ⍝ Selective Assignment, new value 99
 <  10
 <  20
 < <  0  1  2
      3 99  5      ⍝ New value is at the selected location.
   < abc


     f x:x+5       ⍝ User function to manipulate indices
     v←⍳10         ⍝ Variable for Selective Assignment
     (f v)←100+⍳10 ⍝ Make erroneous Selective Assignment:
 ←: index          ⍝ error detected.
*       →


      v            ⍝ In time?
 0 1 2 3 4 100 101 102 103 104
⍝ Not quite. Indices up to first incorrect one were used.
```

Separate Symbols `∪x`

Argument and Result

The argument and result are simple arrays of symbols. The shape of the result is the same as that of the argument except that a last axis of length 2 has been added; i.e., the result shape is `(⍴x),2`.

Definition

Each element of the argument is separated into two symbols, the first coming from whatever precedes the rightmost dot, and the second from whatever follows it; if there is no dot, a leading one is assumed. Unqualified names and root-context names cannot be distinguished in the result of this function.

Example

```
     ∪`x `.y `c.z `a.b.d
```
yields
```
 `     `x
 `     `y
 `c    `z
 `a.b  `d
```

Additional Error Report

If there is no parse or value error (see "Common Error Reports"),

- a maxrank error is reported if the rank of the argument is nine.

Shape ⍴x

Argument and Result

The argument is any array. The result is a vector of integers with an element for each axis of the argument.

Definition

The shape of a scalar is `⍳0` because a scalar has no axes. For a nonscalar the *i*th element of the result is the *i*th dimension of the argument. That is, if the result is *r* then `r[i]-1` is a valid index of the *i*th axis of *x*, but `r[i]` is not.

Examples

```
     ⍴ 3 4⍴0
 3 4
     ⍴ 10 20 30 40
 4
     ⍴ 'The Cat in the Hat'
 18
     ⍴ 42
              ⍝ This is the display of the empty integer vector.

     ⍴ ⍳5
 5
```

Signal ↑x

Argument and Result

x is a simple symbol or character array.

Definition

The effect during function execution is to signal an error in the current function, display the text of the first symbol of *x* or the ravel of the character array *x* as an error message, and cause a suspension at the statement where the current function was called. In protected execution, it returns error code -1. Outside function execution and protected execution, it simply aborts execution of the expression, without any message.

Additional Error Report

The following report is issued only if there is no parse or value error (see "Common Error Reports"):

- a domain error is reported if the argument is not a simple symbol or character array.

Example

```
      f1 a:1+f2{a}
      f2 x:if (x=0) ↑`zero else 2×x
      f1 2
 5
      f1 0
 f2: zero
*      $si
f1 0
.f1: 1+f2{a}
*      a
 0
*      →
```

Solve *yx*

Arguments and Result

x and *y* are numeric arrays whose ranks are less than or equal to 2, with `#y` equal to `#x`. The result has shape `(1↓ρx),1↓ρy`.

Definition

`yx` equals `(x)+.×y`. If *x* is a nonsingular matrix with the same number or rows as columns, the result is the solution *a* to the linear system of equations *y* equals `x+.×a`. If *x* is a matrix with more rows than columns, and if the columns are linearly independent, the result is the least squares solution *a* to that linear system of equations. That is, *a* is the array for which `+/(y-x+.×a)*2` has the smallest possible value for each element.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"):

- a rank error is reported if the rank of either argument exceeds 2;
- a domain error is reported if (1) an argument is not a simple numeric array, or (2) an argument is a matrix with more columns than rows, or (3) the right argument is singular or very ill-conditioned.

Example

Given the following linear system of equations:

```
 10 = 3x + 5y -  z
 -3 = 7x - 2y + 4z
  5 =  x +  y + 2z
```

the solution vector (`x,y,z`) can be determined as follows:

```
      10 ¯3 5  3 3ρ3 5 ¯1 7 ¯2 4 1 1 2
 ¯.5180722892 2.602409639 1.457831325
```

Stop ^*x*

Argument and Result

x is any array, and its value is the explicit result.

Dependency

The value of the system variable `` `stop ``, which is set by the Stop command (`$stop`) and the Set System Variable function (`_ssv`).

Definition

The effect of this function depends on the setting of `` `stop ``. Either the function has no effect (when `` `stop `` is `0`), or it causes execution to halt (when `` `stop `` is `1`), or it causes *x* to be displayed (when `` `stop `` is `2`). A Stop halt is the same as an error halt, except that it can occur within protected execution (⍎ or do).

See "Example of a Defined Function" and, immediately following it, "Errors and Stops" in the appendix.

Example

```
     sum{x}:{
      z←0;
      (i←x) do ^z←z+i+1   ⍝ Call Stop function after each "z←".
      z }
     $stop 0              ⍝ No effect.
     sum 3
 6
     $stop 1              ⍝ Actual stop each iteration.
     sum 3
: stop
*     i,z                 ⍝ Check i and z.
 0 1
*     ←
: stop
*     z←z+10              ⍝ Give z a little boost.
*     ←
: stop


*     &0                  ⍝ Check the top of the stack.
 16
*     ←
 16                       ⍝ The doctored result.
     $stop 2              ⍝ Display the value each iteration.
     sum 3
 1
 3
 6
 6                        ⍝ The result of sum 3.
```

Take *y*↑*x*

Arguments and Result

The argument *y* is a simple one-element array whose value is a restricted whole number, and *x* is any array. A scalar *x* is treated as a one-element vector. The shape of the result equals the shape of the right argument *x* for all but the first axis, while the first dimension is the absolute value of *y*.

Definition

There are four cases for Take, depending on whether or not `y` is nonnegative, and whether or not `|y` is less than or equal to `#x`. The result always has `|y` items. The two cases when `|y` is less than or equal to `#x` are the most straightforward: the result consists of the first `y` items of `x` if `y` is nonnegative, and the last `-y` items if `y` is negative. See the first two examples.

In the two cases where `|y` is greater than `#x`, the difference `(|y)-#x` represents the number of excess items in the result . These excess items consist entirely of *fill elements* (see the "Fill Elements" table). Let `F` be the array of excess items. The shape of `F` equals the shape of `x` along all but the first axis, while the length of the first axis is `(|y)-#x`. The result is the catenation `x,F` if `y` is nonnegative, and `F,x` if `y` is negative. See the last two examples.

Fill Elements

General type of right argument	Fill element
numeric	0
character	blank
symbol	the empty symbol
nested	enclosed Null

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports"), and nonce only if there is no type error:

- a type error is reported if the left argument is nested or is not all restricted whole numbers;
- a nonce error is reported if the left argument has more than one element.

Examples

```
     3↑ι5 2
0 1
2 3
4 5
     ¯3↑ι5 2
4 5
6 7
8 9
     5↑10 20 30
10 20 30 0 0
     ¯5↑10 20 30
0 0 10 20 30
     5↑`a`b`c
`a `b `c ` `
```

Transpose `⍉x`

Argument and Result

`x` is any array. The shape of the result is the reverse of the shape of `x`, i.e., `⌽⍴x`.

Definition

The result is `x` with its axes reversed. In particular, if `x` is a matrix then its rows become the columns of the result, and its columns become the rows. More generally, for any `x` and any nested vector of simple vectors `y` for which `y#x` is valid
`y#x ←→ ⍉(⌽y)#⍉x`
If `y` selects a scalar, then the equivalence is simply `y#x ←→ (⌽y)#⍉x`.

Example

```
      ⍉⍳2 3
 0 3
 1 4
 2 5
```

Transpose Axes `y⍉x`

Arguments and Result

`x` is any array. `y` is a simple vector of restricted whole numbers whose length equals the rank of `x`, i.e., `⍴y` equals `⍴⍴x` (unless `y` is a scalar, which is treated as a one-element vector, yielding the uninteresting case `0⍉x ←→ x` for vector `x`). The rank of the result is equal to the number of distinct elements in `y`, so it is equal to or less than the rank of `x`.

Definition

The result is `x` with its axes permuted and coalesced according to `y`. The elements of `y` must all be in `⍳⍴⍴r`, where `r` denotes the result, and no element of `⍳⍴⍴r` can be missing from `y`. The elements of `y` specify where in the result the corresponding axes of `x` appear. For instance, if `x` has three axes and `y[1]` equals 2, then axis 1 of `x` becomes axis 2 in the result. See the first example. If `y` contains duplicates, then, for each set of duplicates, the corresponding axes in `x` are coalesced into one axis in the result, and the length of the resulting axis is the least of the lengths of the coalesced axes. In particular, if `x` is a matrix then `0 0⍉x` is the diagonal of `x`. See the second and third examples.

In terms of indices, if `i` is a vector of indices which selects an element of `x`, then there are two possibilities. (1) For some set of coalesced axes, the corresponding elements of `i` are different. Then the indexed element of `x` is not in `y⍉x`. (2) The element is in `y⍉x` and its indices there are `i[y⍳⍳⍴⍴y⍉x]` (using, for each set of coalesced axes, the index for the first axis in the set, as an arbitrary choice).

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if the left argument is nested or does not consist of restricted whole numbers;

- a rank error is reported if the left argument is not a scalar or vector, or if its length is unequal to the rank of the right argument;
- a domain error is reported if the left argument contains a negative number or does not contain *all* the restricted whole numbers between zero and its largest element, inclusive, possibly with duplications.

Examples

```
     0 2 1⍉⍳2 3 4
 0  4  8
 1  5  9
 2  6 10
 3  7 11

12 16 20
13 17 21
14 18 22
15 19 23
     0 0⍉⍳2 3
0 4
     1 0 0⍉⍳2 3 4
 0 12
 5 17
10 22
```

Type ∨*x*

Argument and Result

The argument is any array; if the argument is a function expression it must be in braces. The result is a symbol scalar.

Definition

The result is the type of the first item of the argument, represented as a symbol. The types are:

character, integer, floating point, null, symbol, function, and box
(`` `char ``, `` `int ``, `` `float ``, `` `null ``, `` `sym ``, `` `func ``, and `` `box ``, respectively).

Symbol, function, and box items can be mixed in a single array. See the last three examples in the set that follows. There is also a type included to provide for possible weird cases; its symbol is `` `unknown ``.

Examples

```
     ∨'a'
 `char
     ∨42
 `int
     ∨4.2
 `float
     ∨`a
 `sym
     ∨{+} ⍝ The parser needs the braces as a hint when a function is an
argument.
 `func
     ∨<{+}
 `func    ⍝ Function scalar and function expression are both type `func.
     ∨()
```

```
`null
     v<'a'
`box
     v`a,(<1251),<{+}
`sym
     v(<1251),(<{+}),`a
`box
     (<{+}),`a,<1251
`func
```

Unpack ⊤x

Argument and Result

x is a simple array composed entirely of symbols. The result r is a simple character array whose shape along all axes except the last is ρx. I.e., ¯1↓ρr is ρx.

Definition

Pack is the left inverse of Unpack. That is, for any array x composed entirely of symbols, ⊥⊤x equals x. The result has one more axis than x, the last. Each character vector along the last axis of the result consists of the characters - except for the leading backquote - in the display of the corresponding symbol of x, perhaps padded with blanks on the end. The last dimension of the result is the largest number of characters in any of these vectors, before padding.

Additional Error Reports

The following reports are issued only if there is no parse or value error (see "Common Error Reports"):

- a maxrank error is reported if the rank of the result would be greater than nine;
- a type error is reported if the argument is not a simple array of symbols.

Example

```
     ⊤`a `abc
a
abc
```

Value %x

Argument and Result

x is a scalar symbol that contains a user global name (see "User Names" and "Symbols and Symbol Constants"). The result is an array or a function.

Definition

If the name in the symbol x is not qualified by a context name, then the current context is assumed. If it is so qualified, the designated context is assumed. For example, `v.a refers to a in the context v, `.a refers to a in the root context, and `a refers to a in the current context. See the example with the function f below.

There are two cases. If the thing named by the symbol x is a *global variable*, the result is the value of that variable. If the thing is a *defined function* (not a primitive function, because a primitive function does not have a user name), then the result is that function: if %x occurs bare, the function definition will be displayed; if it occurs with argument(s) the function will be executed, *provided general function form is used*, with those arguments in braces.

Unlike Execute, Value can be used on the left of Assignment to set the value of a global variable.

Additional Error Reports

The following reports are issued only if there is no parse error (see "Common Error Reports"). A value error report (another of the common error reports) can happen here because either ⊤*x* is not a user name or it lacks a *global* value. A value error will be reported only if there is no type or rank error.

- A type error is reported if the type of *x* is not symbol.
- A rank error is reported if *x* is not a scalar.
- If *x* names a defined function, any further errors will be reported for that function.

Examples

```
     sq x:x*2
     f←`sq
     x←11 12 13
     a←`x
     %f
sq x:x*2
     (%f){%a}
 121 144 169
     a←⍳4
     %`a
 0 1 2 3
     (%`a)←'new'    ⍝ Value can be used on the left of Assignment.
     a
new
     %`q            ⍝ q has no value.
 .q: value
*       →
     f x:{
       a←x;         ⍝ Set local a
       ⇣a;          ⍝ Display local a
       ⇣%`a;        ⍝ Display global a, current context.
       ⇣%`.a; }     ⍝ Display global a, root context.
     a←1 2 3

     f 5
 5
 1 2 3              ⍝ The current context is the root context.
 1 2 3
     $cx cc         ⍝ Set a new context.
     a←4 5 6        ⍝ Set a in the new context, cc.a




     f 5
 5                  ⍝ The local a
 4 5 6              ⍝ The value in the current context, cc
 1 2 3              ⍝ The value in the root context.
```

Value in Context *y*%*x*

Arguments and Result

x and *y* are scalar symbols such that *y*∪*x* is a symbol containing a user global name (see "User Names" and "Symbols and Symbol Constants"). The result is an array or a function.

Definition

Value in Context generates the name it uses in one of two ways:

- if the name represented by the symbol *x* does not specify a context, then *y*%*x* qualifies it by the context named by the symbol *y*. For example, `` `ctx%`a `` uses the name `ctx.a`;
- otherwise, the name represented by *x* specifies a context, as in `` `ctx.a ``, and *y*%*x* uses that name, and ignores the value of the symbol *y*. For example, `` `any%`ctx.a `` uses `ctx.a`.

The root context is represented by ` (the blank or empty symbol: just backquote alone).

There are two cases. If the thing named by *y* and *x* is a *global variable*, the result is the value of that variable. If the thing is a *defined function* (not a primitive function, because a primitive function does not have a user name), then the result is that function: if *y*%*x* occurs bare, the function definition will be displayed; if it occurs with argument(s) the function will be executed, *provided general function form is used*, with those arguments in braces.

Unlike Execute, Value in Context can be used on the left of assignment to specify the value of a global variable in a context. And unlike Execute in Context, Value in Context in effect constructs a qualified name and then evaluates it. Within a defined function or operator, `` `cxt%`x `` always refers to the global variable `cxt.x`, whereas `` `cxt⍎'x' `` refers to the local variable *x* if there is one, and only if there is no such local variable does it refer to `cxt.x`.

Additional Error Reports

The following reports are issued only if there is no parse error (see "Common Error Reports"). A value error report (another of the common error reports) can happen here because either *x* and *y* do not define a user name or the name they produce lacks a *global* value. A value error will be reported only if there is no type or rank error. An error report for *x* is preferred to an error report for *y*.

- A type error report indicates that the type of *x* or *y* is not symbol.
- A rank error report indicates that *x* or *y* is not a scalar.
- If a defined function is named by *y* and *x*, any further errors will be reported for that function.

Examples

```
     my.a←⍳4
     cx←`my
     v←`a

     cx%v                   ⍝ Value of a in the context my
 0 1 2 3
     `anything%`my.a        ⍝ `anything is ignored.
 0 1 2 3
     x←30 40
     $cx your
     `%`x                   ⍝ Value of x in the context.
 30 40
     cx%`x                  ⍝ The value of cx is my
 my.x: value
*      →
      my.a←ι4
      (cx%v)←'new'
      my.a
new
```

7. Representation of Numbers

The machines on which A+ is run represent numbers in binary (base 2), for efficient storage and computation. A+ (via C) represents integers in 32 bits (binary digits) and floating-point numbers in 64. The largest integer representable in this way is `(2*31)-1` and the smallest is `¯2*31` (-2147483648); the greatest magnitude representable as a floating-point number is approximately `1.7977e+308`.

In display and print, of course, A+ represents numbers in decimal. People tend to think in decimal and their input is usually decimal. Numbers like .1 and .01 seem exactly representable intrinsically (as they *are* in decimal), but, in fact, these numbers cannot be exactly represented in binary, and decimal to binary and binary to decimal conversions are one source of imprecision.

Another source of imprecision is the necessarily limited number of digits that can be used to represent a number. One effect of this limitation is inexactitude, which can be seen in simple examples:

```
      123456789123456789 - 123456789123456788
 0
      12345678912345679 - 12345678912345678
 2
      .123456789123456789 - .123456789123456788
 0
```

`and yet:`

```
      .000000000000000001

 1e-18
```

Another effect of this limitation is, of course, that there are numbers that are simply too large to be numerically represented in A+ at all, such as `10*1234`.

Results can only be *meaningfully* displayed out to a maximum of 16 digits. Any digits shown beyond the 16th digit are invalid. For example, the actual value of pi begins with 3.141592653589793238; but if we set $pp to 20 (a value that is beyond the meaningful range), `o1` will return 3.14159265358979311 6. Only the first 16 digits are correct.

To allow for slight imprecisions in the representation of numbers by considering numbers to be equal when they are very close to equal, the concept of a *comparison tolerance* is used. To allow computations to continue when some of the numbers involved have become too large to be represented, a special representation, *`Inf`*, has been introduced, together with its negative, *`¯Inf`*.

Comparison Tolerance

Comparison tolerance cannot be set; it is fixed at `1e-13`.

It is used in both conversion of floating-point numbers to integers and explicit comparisons, as in Equal to, Match, and Find.

The fundamental notion is to put a band around one of the comparands and see whether the other comparand falls within that band. Consider `x=y`, for instance. Roughly speaking, the equality holds if `x` is within (without tolerance, of course) the interval from `y×1-1e-13` to `y×1+1e-13`. More precisely, the difference between the comparands must be less (without tolerance) than the tolerance times the smaller of them in absolute value: `|x-y` must be less than `1e-13×(|x)⌊|y`. Because multiplication is involved, only zero is equal to zero within the comparison tolerance.

Comparison tolerance applies only to floating-point comparisons. The largest magnitude that can be represented in A+ in integer type is `2*31`, which is `|¯2*31`. That times the tolerance, i.e., `1e-13×2*31`, is only about 0.0002. Thus the tolerance could not have any effect on comparisons between two integers.

When examining the effect, or apparent effect, of comparison tolerance, you must take into account the printing precision (which may make two unequal numbers look equal) and the fact that the Equal to function itself uses comparison tolerance. As an intolerant check for equality of `a` and `b`, you can use `0=a-b`.

Consider numbers shown in an example above, where their difference was approximated as 2:

```
     1234567891234567 9 = 1234567891234567 8
 1
```

Clearly, Equal to uses comparison tolerance to arrive at the result: A+ disregards the insignificant (and in fact inexact) difference it finds between them. But doesn't comparison tolerance apply only to floating-point numbers? Yes, but there are several ways to enter numbers for representation in floating point: e.g., with a decimal point (`99.` or `3.14`, for instance), in exponential notation (`6.023e-23`), and with too many digits for it to be represented internally as an integer (`12345678912345679`). Each number entered is converted to its internal representation before anything else is done with it.

Comparison tolerance is used in:

- Ceiling, Floor, and Residue;
- Equal to, Greater than, Greater than or Equal to, Less than, Less than or Equal to, and Not equal to;
- Find, Match, and Member;

and no other functions.

When two floating-point numbers are equal within the comparison tolerance, they are called *tolerably equal*. This term is also used in a more general way, to mean equal within the tolerance for floating-point numbers and strictly equal where no floating-point number is involved.

Note that the term *restricted whole number*, as discussed above and in "Restricted Whole Numbers", involves even more toleration, since it includes numbers whose absolute value is less than `1e-13`, whereas zero is not tolerably equal to any nonzero number.

Arithmetic of Inf

A+ uses `Inf` to denote positive numbers too large for it to represent, and `¯Inf` for negative numbers whose magnitude is too great to be represented. `Inf` is tantamount to infinity. You can enter these notations directly; also, if you enter, say, `1e309`, it will be shown and stored as `Inf`.

Instead of a domain error, division by zero of a positive number yields `Inf`, and of a negative number, `¯Inf`. The Min reduction of an empty vector yields `Inf`, and the Max reduction yields `¯Inf`. And so on.

When either of these values is an argument to a function, most of what you might expect happens: `-¯Inf` is `Inf`, and `×Inf` is 1, and `4⌊Inf` is 4.

Any operation which heightens `Inf` or `¯Inf` produces it as its result: `10×Inf` is `Inf`, and `¯Inf-100` is `¯Inf`, and `Inf+Inf` is `Inf`, and so forth. Operations involving representable numbers that tend to lessen `Inf` or `¯Inf` are also accepted, however, so that, for example, `Inf-1e99` is `Inf`. (Indeed, although they are unlikely to occur, `1e309÷1e308` is `Inf` and `1e308÷1e309` is 0, because `1e309` is converted to `Inf` upon input, before the division, and `1e308` is representable.)

Domain errors result from indeterminate expressions like `Inf-Inf` and `0×Inf`, as well as `0÷0`.

NaN

A+ does not handle NaN (Not a Number) but it provides tools that let you detect NaN's. See the `_nanfind` system function and the `$dbg` command, especially the table "$dbg Subcommands (Arguments)".

8. Monadic Operators

Common Error Reports

Multiple errors elicit but one report. If an error report in the following list is issued, then the ones preceding it do not apply. Five reports are common to all monadic operators:

- parse;
- value: an argument has no value;
- nondata: an argument (not an operand) is a function or some other nondata object;
- an error report from the operand function;
- wsfull: the workspace is currently not large enough to execute the function in; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;
- interrupt (not an error): the user pressed **c** twice (once if A+ was started from a shell) while holding the **Control** key down.

Definitions of Monadic Operators

Apply `g¨x` and `y g¨x`

Arguments

g is a function scalar; that is, *g* is `<{f}` for some function *f* (other than Assignment). The arguments *y* and *x* are suitable arguments for *f*.

Definition

The function *f* described in the arguments section is evaluated, as `f x` in the monadic case or as `y f x` in the dyadic case. Indeed, although the two infix forms are shown for definiteness, `f{...}` is evaluated for any valence of *f*, zero through nine.

Warning! If instead of a function scalar you give a function expression, (e.g., instead of `(+;-;×)[1]¨x` or `(<{f})¨x` you give `-¨x` or `f¨x`), you will get a wrong result, namely, the result for the Each operator.

Additional Error Report

The following report is issued only if there is no parse or value error (see "Common Error Reports"):

- a nonfunction error is reported if *d* is not a function scalar (or function expression: see the warning in the definition section); *d* precedes the colon in the error report.

Examples

```
     fns←(×;÷;ρ)    ⍝ Equivalent to <{×},<{÷},<{ρ}

     2 3 fns[0]¨ 4
8 12

     2 3 fns[1]¨ 4
0.5 0.75

     2 3 fns[2]¨ 4
4 4 4
4 4 4
```

Bitwise `f⍨x` and `y f⍨x`

Arguments

The operand *f* is Cast, And, Or, Not, or one of the six relational functions (`< ≤ = ≥ > ≠`). The arguments *y* and *x* are suitable arguments for *f*.

Definition

The Bitwise operator applies Not to each bit of the argument and applies all the other functions except Cast to each pair of corresponding bits of the arguments.

The derived function Bitwise Not is a monadic scalar function and all the others except Bitwise Cast are dyadic scalar functions, obeying the usual rules of conformability.

Examples for Bitwise Not, And, and Unequal

```
     ~⍨ ¯16 15 ¯123 122
 15 ¯16 122 ¯123          ⍝ One's complement.
```

```
     5 ∧⍤ ⍳10
 0 1 0 1 4 5 4 5 0 1

     5 ≠⍤ ⍳10
 5 4 7 6 1 0 3 2 13 12
```

And and Or, without the Bitwise operator, give boolean results for all valid arguments - see above.

Bitwise Cast `∨⍤x`

Bitwise Cast leaves the data part of the variable unchanged but changes the type indicator and (usually) shape. The conversion of symbol to character, for example, is quite different from plain Cast: for a symbol, the data part is a four-byte pointer, which may differ from process to process but is guaranteed unique and invariant within a process. The data part of a character is, of course, its ASCII code.

The left argument can be `` `int ``, `` `float ``, or `` `char ``; a Bitwise Cast to `` `sym `` is not allowed since not all integers are valid symbols, i.e., not all point to symbols. The type of the right argument can be of any of the four.

Since scalars of various types are represented using various numbers of bytes, the length of the last dimension of the result may differ from that of the right argument. There is no padding or truncation. If `rb` bytes are required to represent the result type and `ab` to represent the right argument type and `rb>ra`, then the last dimension of the right argument must be divisible by `rb÷ab`. The table below exhibits the relation between the shape of the right argument and that of the result.

The rank is changed only when a scalar is cast bitwise to a type that is represented in fewer bytes.

Shape of the Result of Bitwise Cast, for Argument Shape s

	From `` `int ``	**From `` `float ``**	**From `` `char ``**	**From `` `sym ``**
To `` `int ``	`s`	`(¯1↓s), 2×¯1↑1,s`	`(¯1↓s), (¯1↑s)÷4`	`s`
To `` `float ``	`(¯1↓s), (¯1↑s)÷2`	`s`	`(¯1↓s), (¯1↑s)÷8`	`(¯1↓s), (¯1↑s)÷2`
To `` `char ``	`(¯1↓s), 4×¯1↑1,s`	`(¯1↓s), 8×¯1↑1,s`	`s`	`(¯1↓s), 4×¯1↑1,s`

Examples for Bitwise Cast

```
     `int ∨⍤ `
 4163890


     `int ∨⍤ 1.2 3.4
 1072902963 858993459 1074475827 858993459

     `char ∨ `int ∨⍤ 'abcdefgh'
dh  ⍝ Plain Cast converts just the last byte of an integer to character
```

```
     `char ∨⍨ `int ∨⍨ 'abcdefgh'
abcdefgh

     `int ∨⍨ 'ab'
⍝[error]  ∨⍨: length  ⍝ The shape is 2, which is not divisible by 4.
```

Additional Report

- An "undefined" token report is issued if `f` is not one of the functions listed above.

Each `f¨x` and `y f¨x`

Arguments and Result

`f` is any function expression (so Assignment is excluded, as it is for all operators). The result is nested. In the monadic case, the shape of the result is the shape of `x`. In the dyadic case, the shapes of the arguments and result follow the rules for dyadic scalar functions (see "Application, Conformability, and Result Shape"), except that only scalars are extended, not one-element arrays in general, and nonscalar arguments of differing ranks elicit a rank error report.

Definition

The derived function, `f¨`, is a scalar function.

In the case of monadic `f`:

```
     (i#f¨x) ←→ < f >i#x
```

for every valid index `i`. That is, each element of `x` is selected, `f` is applied to the Disclose of the selected element, and the result of `f` is enclosed and inserted in the result of `f¨`. (See "Disclose" for the definition of `>` and "Enclose" for `<`.) `f¨x` is equivalent to `(f Each)x` for the following defined operator `Each`:

```
     (f Each) x: {
          s←ρx;
          x←,x;
          z←(ρx)ρ<();
          (i←ρz) do z[i]←< f >x[i];
          sρz}
```

In the dyadic case, if `x` and `y` have the same shape, then

```
     (i#y f¨x) ←→ <(>i#y)f>i#x
```

for every valid index `i`. If `x` has one element, then

```
     (i#y f¨x) ←→ <(>i#y)f 0⊃x
```

for every valid index `i`. (See "Pick" for the definition of `⊃`.) The definition is similar if `y` has one element, or both have one element.

Warning! If instead of a function expression you give a function scalar, (e.g., instead of `-¨x` or `f¨x` you give `(+;-;×)[1]¨x` or `(<{f})¨x`), you will get a wrong result, namely, the result for the Apply operator.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a nonfunction error is reported if *d* is not a function expression (or function scalar: see the warning in the definition section); *d* precedes the colon in the error report;
- a rank error is reported, when the derived function is dyadic, if neither argument is a one-element array and their ranks differ;
- a length error is reported, when the derived function is dyadic, if neither argument is a one-element array and their shapes differ.

Examples

In the first expression, the Each operator is applied to a monadic function (Shape) to give the rank of a scalar, a vector, and a matrix. In the second, it is applied to a monadic (Count, to find the vector lengths) as well as to a dyadic (Take, to pad the vectors to equal lengths for the principal function, Disclose); notice the other use of Disclose to make the Count Each result simple for Max Reduce, which finds the greatest length.

```
     ⊃ρ¨ρ¨ (2;⍳2;⍳2 3)
 0 1 2

     (⌈/>#¨vs)↑¨vs←('ab';'----';'cde')
ab
----
cde
```

Reduce `f/x`

Arguments and Result

`f` is one of `+`, `×`, `⌊`, `⌈`, `∧`, `∨`. The argument `x` is any array whose items are suitable left and right arguments of `f`. The shape of the result is `1↓ρx`.

Definition

If `#x` is at least two, i.e., if `x` has at least two items, then `f/x` is defined to be:

```
     x[0] f x[1] f ... f x[¯1+#x]
```

If `#x` is one, then `f/x` is `x`; while if `#x` is zero, then `f/x` is `(1↓ρx)ρidentity`, where `identity` is a scalar that depends on `f`, and whose type for `+` and `×` depends on `x`. See the table "Identity Scalars for Reduction".

As pointed out in "Operators and Derived Functions", Reduce is not in the strictest sense an operator, but for most purposes it can be regarded as one.

Identity Scalars for Reduction

Function	Identity
+	0
×	1

Function	Identity
∨	0
∧	1

Function	Identity
⌈	`¯Inf`
⌊	`Inf`

Additional Error Reports

If there is no error reported as a parse error then a similar error may be reported:

- a token error is reported if `f` is a primitive function or operator symbol but not one of `+×⌊⌈∧∨`.

The following report is issued only if there is no parse, token, or value error (see "Common Error Reports"):

- a valence error is reported if `f` is a defined function (this report seems worth mentioning here, although it actually comes from Replicate - which of course also produces error reports and results when `f` is a variable).

Examples

```
      +/3 5 ¯2
6

      3+5+¯2
6

      ⌈/⍳6 2
10 11

      +/⍳0
0
```

Scan `f\x`

Arguments and Result

`f` is one of `+`, `×`, `⌊`, `⌈`, `∧`, `∨`. The argument `x` is any array whose items are suitable left and right arguments of `f`. The shape of the result is the same as the shape for `x`.

Definition

If `#x` is at least one, i.e., if `x` has at least one item, then:

```
          (f\x)[i] ←→ f/x[⍳i+1]
```

for every valid scalar index `i`. That is, Scan produces a sequence of moving reductions. If `#x` is zero, then `f\x` is empty, and its type is integer for `∧` and `∨` and depends upon `x` for the rest.

As pointed out in "Operators and Derived Functions", Scan is not in the strictest sense an operator, but for most purposes it can be regarded as one.

Additional Error Reports

If there is no error reported as a parse error then a similar error may be reported:

- a token error is reported if `f` is a primitive function or operator symbol but not one of `+×⌊⌈∧∨`.

The following report is issued only if there is no parse, token, or value error (see "Common Error Reports"):

- a valence error is reported if *f* is a defined function (this report seems worth mentioning here, although it actually comes from Expand - which of course also produces error reports and results when *f* is a variable).

Examples

```
      +\3 5 ¯2
3 8 6

      ⌊\⍳3 4
0 1 2 3
0 1 2 3
0 1 2 3

      ⌈\⍳3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11
```

9. Dyadic Operators

Common Error Reports

Multiple errors elicit but one report. If an error report in the following list is issued, then the ones preceding it do not apply. Five reports are common to all dyadic operators:

- parse;
- value: an argument has no value;
- nondata: an argument (not an operand) is a function or some other nondata object;
- an error report from an operand function;
- wsfull: the workspace is currently not large enough to execute the function in; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;
- interrupt (not an error): the user pressed **c** twice (once if A+ was started from a shell) while holding the **Control** key down.

Definitions of Dyadic Operators

Inner Product `y f.g x`

Arguments and Result

There are three inner products in A+: `+.×`, `⌈.+`, and `⌊.+`. The arguments *y* and *x* must be nonscalars, and `¯1↑⍴y` must equal `1↑⍴x`. The shape of the result is `(¯1↓⍴y),1↓⍴x`.

Definition

For matrices *y* and *x*, the function *g* is applied to row vectors of *y* on the left and column vectors of *x* on the right, in all combinations; `f/` is applied to each these results. That is:

```
      (y f.g x)[i;j] ←→ f/y[i;]g x[;j]
```

for all scalar indices *i* and *j*. The general definition of `+.×` is:

```
      y (pdt⍤1 1 0)(¯1⌽⍳⍴⍴x)⍉x
```

where

```
      a pdt b:+/a×b
```

The function `pdt` must be applied to vectors along the last axis of `y` and vectors along the first axis of `x`, in all combinations. In order to do this with the Rank operator, the vectors along the first axis of `x` must be moved to the last axis. The expression `(¯1⌽⍳⍴⍴x)⍉x` has the effect of moving the first axis of `x` to the last, while leaving all other axes in their original order.

Analogous definitions hold for `⌈.+` and `⌊.+`.

As pointed out in "Operators and Derived Functions", Inner Product is not in the strictest sense an operator, but for most purposes it can be regarded as one.

Additional Error Reports

Each of the following reports is issued only if there is no parse or value error report (see "Common Error Reports") - the token report preceding the value report -, and none of the reports preceding it on this list applies:

- a token error is reported if `f` or `g` is a defined function name or a primitive function or operator symbol such that the derived function is not one of the three permitted forms;
- a valence error is reported if there is no left argument for the derived function (a missing right argument is a parse error);
- a rank error is reported if one of the arguments is a scalar;
- a length error is reported if the last dimension of `y` and the first dimension of `x` are not equal.

Example

```
      (⍳2 3)+.×(⍳3 4)
 20  23  26  29
 56  68  80  92
```

Outer Product[1] `y ∘.f x`

Arguments and Result

`f` can be one of `+`, `×`, `-`, `÷`, `⌈`, `⌊`, `|`, `<`, `≤`, `=`, `≥`, `>`, `≠`, and `⋆`. The arguments `y` and `x` are any arrays whose elements are suitable left and right arguments of `f`, respectively. The shape of the result is `(⍴y),(⍴x)`.

Definition

The function `f` is applied to all combinations of scalars from `y` and `x`. In particular, for vectors `x` and `y`:

```
      (y∘.f x)[i;j] ←→ y[i]f x[j]
```

for all scalar indices `i` and `j`. Similar relations hold for arrays of other ranks. In general, `y∘.f x` is equivalent to `y(f⍤0 0 0)x`.

As pointed out in the "Operators and Derived Functions", Outer Product is not in the strictest sense an operator, but for most purposes it can be regarded as one.

Additional Error Reports

If there is no error reported as a parse error, then a similar error may be reported:

- a token error is reported if *f* is a defined function name or a primitive function or operator symbol such that the derived function is not one of the permitted forms.

The following report is issued only if there is no parse, token, or value error (see "Common Error Reports"):

- a valence error is reported if there is no left argument for the derived function (a missing right argument is a parse error).

Example

```
      1 10 100∘.×1.2 ¯3 98.2 5
    1.2    ¯3      98.2     5
   12     ¯30     982      50
  120    ¯300    9820     500
```

Rank f@n x and y f@n x

(Just as for all operators, *f* cannot be Assignment.) Rank's derivation of monadic functions and its derivation of dyadic functions are covered in separate sections below. First, however, you should understand how the operator decomposes arrays, and the implications for its handling of empty arrays.

Rank Operator uses Frames and Cells

The Rank operator (`f@n x` and `y f@n x`) splits an array into equal-shaped cells held in a frame and applies its function to each cell. The frame's shape consists of zero or more leading dimensions of the array and the cells' shape consists of the remaining zero or more trailing dimensions. (Cf. "Dimension, Shape, and Rank".) Rank deriving dyadic splits each argument separately.

For example, a function applied to an array of shape 3 4 5 using the Rank operator may be applied to:

- cells of shape `⍳0` (scalar) in a frame of shape 3 4 5; or
- cells of shape 5 in a frame of shape 3 4; or
- cells of shape 4 5 in a 3-item frame; or
- a single cell with shape 3 4 5 in a frame with shape `⍳0` (scalar).

Rank allows specification of the rank of either the cells (a positive number in *n*) or the frame (a negative number in *n*, whose magnitude is used). For example, to have the Rank operator apply a function to rank-2 cells in a 3-dimensional array, you can use either 2 (for the rank of each cell) or -1 (for a frame rank of 1).

If `|n` exceeds the rank of an argument to which it is applied, it is treated as if it equaled that rank. Thus you can specify both extreme cases in a way independent of rank: to allot all the axes to the frame, you specify cell rank to be 0, of course; to give all axes to the cells, you specify cell rank to be 9, the limit on ranks.

The Rank Operator and Empty Arguments

The shape of the result of the derived function is always determined in the same way (with any additional axes on the right produced by the operand function), whether the arguments are empty or not. The type of the result, however, is a different matter. In the absence of an empty frame, the type is determined by the operand function, for the empty cells. On the other hand, if a frame is empty, the Rank operator never calls this function. For a primitive function, Rank uses its result type, or at least its default result type. For a nonprimitive function, it uses null. These examples show this type selection:

```
      y f x:x      ⍝ A simple function to be used as an operand.
      ∨2(f@1)⍳0    ⍝ Scalar frames, empty-vector cells (left argument is made
 `int              ⍝ to conform to right). An empty integer vector is returned
                   ⍝ to Rank by f.
      ∨2(f@0)⍳0    ⍝ An empty-vector frame (from right argument); no (scalar)
 `null             ⍝ cells for f, so no call. Rank makes the result type null.
      ∨2(+@1)⍳0    ⍝ Like the first example.
 `int              ⍝ +⍳0 is integer.
      ∨2(+@0)⍳0    ⍝ Rank doesn't apply Add to anything, but does use the type
 `int              ⍝ implied by Add: the Add result type for integers is
integer,
                   ⍝ if the result can be so represented.
      (⍳0) ÷@0 ⍳0  ⍝ Rank passes fake args (the usual zeros, unhappily)
 ÷: domain         ⍝ to the operand function to determine the size and
                   ⍝ type of the result. Divide is not amused.
```

Rank Deriving Monadic f@n x

The rank specification *n* is a one-element integer array. For conformity with the discussion of Rank deriving dyadic, assume *n* is a vector. Then if *n* is nonnegative, Rank applies *f* to cells of rank *j*←*n*[0]⌊⍴⍴*x* within a frame of shape (-*j*)↓⍴*x*.

If *n* is negative, Rank applies *f* to cells of rank *k*←0⌈(⍴⍴*x*)-|*n*[0] within a frame of shape (-*k*)↓⍴*x*.

Additional Error Reports

Each of the following reports is issued for Rank deriving monadic only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if *n* is not a simple integer array;
- a length error is reported if *n* has more than 3 elements (although only the first element is used);
- a nonfunction error is reported if *f* is not a function.

Examples

```
      (+/@1) 2 3⍴10 20 30 1 2 3 ⍝ The data operand must be separated
 60 6                         ⍝ from the left argument of Reshape: parentheses used.
```

```
      ⌽@¯1 ⍳2 3 4    ⍝ The frame is of rank 1, so the cells are of rank 2.
  8   9 10 11        ⍝ ⌽ is called twice, with matrices as arguments,
  4   5  6  7        ⍝ so the items whose order is reversed by
  0   1  2  3        ⍝ ⌽ are rows.



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

      ⌽@1  ⍳2 3 4    ⍝ The cells are of rank 1, so the frame is of rank 2.
  3   2  1  0        ⍝ ⌽ is called six times, with rows as arguments,
  7   6  5  4        ⍝ so the items whose order is reversed by
 11  10  9  8        ⍝ ⌽ are scalars.

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

      ⌽@3  ⍳2 3 4    ⍝ The cells are of rank 3, so the frame is scalar.
 12  13 14 15        ⍝ ⌽ is called just once, with a rank-3 array as arg,
 16  17 18 19        ⍝ so the items whose order is reversed by
 20  21 22 23        ⍝ ⌽ are matrices.
                     ⍝ ⌽@3  ⍳2 3 4 is equivalent to ⌽⍳2 3 4
  0   1  2  3
  4   5  6  7
  8   9 10 11
```

Rank Deriving Dyadic `y  f@n  x`

The rank specification `n` must be an integer array with one, two, or three elements. For convenience, assume `n` is a vector throughout this description.

There are three behaviors for Rank deriving dyadic, depending upon `n`;

- when `n` has one or two elements, Rank deriving dyadic bears some similarity to a scalar dyadic function: the resulting frames on `x` and `y` must be the same shape after any deficiency in the rank of one of them is made up by replication (a very permissive scalar extension), and the operand function is applied to pairs of corresponding cells;
- when `n` has a third element of zero, the operator bears a similarity to Outer Product: the operand function is applied to pairs of cells in all possible combinations;
- when `n` has a nonzero third element, the operator bears some similarity to a scalar dyadic function on the trailing `n[2]` axes of the frames and a similarity to Outer Product on the remaining (leading) axes of the frames (but see below for the subtleties glossed over here).

If `n` has one element and is nonnegative, Rank applies `f` to cells of rank `j←n[0]⌊⍴⍴x` within a frame of shape `(-j)↓⍴x` on `x` and to cells of rank `k←n[0]⌊⍴⍴y` within a frame of shape `(-k)↓⍴y` on `y`.

If `n` has one element and is negative , Rank applies `f` to cells of rank `j←0⌈(⍴⍴x)-|n[0]` within a frame of shape `(-j)↓⍴x` on `x` and to cells of rank `k←0⌈(⍴⍴y)-|n[0]` within a frame of shape `(-k)↓⍴y` on `y`. A common case is `f@¯1` (which pairs items from both arguments and applies the function `f` to each pair).

If `n` has two elements, then the first element of `n` is used for `y` and the second element is used for `x`, each interpreted in the same way as when `n` has one element.

When `n` has three elements, the first two are used in the manner just stated. The third element of `n` breaks the frame into two subframes, leading and trailing. It specifies the number of trailing axes in the frames of `x` and `y` in which cells will be paired with corresponding cells, and in which, therefore, mismatched dimensions are forbidden. Any leading dimensions present in one trailing subframe and absent in the other - which must in this case be a whole frame -, are supplied by replicating the latter subframe. Any remaining (leading) axes of the frames are used to create subframes whose members are paired in all possible ways, in the manner of an outer product. Again, `n` can be excessive; any excess is dealt with as described in the next paragraphs.

Assume, just for convenience, that the first two elements of `n` are positive. Set `cy←n[0]⌊ρρy` and `cx←n[1]⌊ρρx` (cell ranks), `ty←n[2]⌊(ρρy)-cy` and `tx←n[2]⌊(ρρx)-cx` (ranks of the trailing parts of the frames), and `ly←0⌈(ρρy)-cy+ty` and `lx←0⌈(ρρx)-cx+tx` (ranks of the leading parts of the frames).

That is, the cell specifications are honored to the extent possible, then the trailing subframe specification is honored to the extent possible, and the leading subframes are whatever is left. The function `f` is applied to pairs of cells of rank `cy` and `cx` taken from `y` and `x` respectively, using all combinations of subscripts for the first `ly` and `lx` axes and corresponding subscripts for the next `ty⌈tx` axes (after possible replication of one trailing subframe to make these subframes match).

Whether the cells match is the responsibility of the operand function. The two leading subframes have no compatibility requirement. The two trailing subframes must match only as far as they both exist: specifically,

```
    (-ty⌊tx)↑(-cy)↓ρy   ←→   (-ty⌊tx)↑(-cx)↓ρx.
```

Clearly, if `ty` is less than `tx` (and so the trailing subframe for `y` is replicated to match the shape of the trailing subframe for `x`), then `ly` is zero, and if `tx` is less than `ty`, then `lx` is zero.

```
Note that:
if n has only one element, y (f@n) x is equivalent to y (f@ n,n) x
        which is equivalent to y (f@ n,n,9) x
and if n has exactly two elements, y (f@n) x is equivalent to y (f@ n,9) x.
When n has fewer than three elements, the trailing subframes are simply the
frames.
```

Additional Error Reports

Each of the following reports is issued for Rank deriving dyadic only if there is no parse or value error (see "Common Error Reports") and none of the reports preceding it on this list applies:

- a type error is reported if `n` is not a simple integer array;
- a length error is reported if `n` has more than 3 elements;
- a mismatch error is reported if the shapes of the subframes do not match;
- a nonfunction error is reported if `f` is not a function.

Examples

```
    '-->' ,@1 ⊢4!'abcdABCD' ⍝ Rank 1 cell (rank 0 frame) vs. rank 1
 -->abcd                    ⍝ cell (rank 1 frame). The scalar left
 -->ABCD                    ⍝ frame is treated as a 2-element vector.
```

```
     '01' ,@¯1 ⊢4!'abcdABCD' ⍝ Rank 1 frame (rank 0 cell) vs. rank 1
 0abcd                       ⍝ frame (rank 1 cell).
 1ABCD
```

```
     (⍳3)(,@0 1)⍳2 3 4       ⍝ Rank 0 cell (rank 1 frame) vs. rank 1
   0  0  1  2  3             ⍝ cell (rank 2 frame). The vector left
   1  4  5  6  7             ⍝ frame is treated as a 2 by 3 matrix: the
   2  8  9 10 11             ⍝ "missing" dimension is supplied.

   0 12 13 14 15
   1 16 17 18 19
   2 20 21 22 23

     (1990+⍳2)(,@0 1 0)⍳2 3  ⍝ Both left argument scalars vs.
 1990    0    1    2         ⍝ both right argument vectors. Cells left
 1990    3    4    5         ⍝ rank 0, right rank 1; rank 0 trailing
                             ⍝ subframes and rank 1 leading subframes
 1991    0    1    2         ⍝ for both.
 1991    3    4    5

     (⍳2 3 4),@1 1 2(⍳2 3 9) ⍝ Vector vs. vector, rank 0 leading
   0  1  2  3  0  1  2  3  4  5  6  7  8  ⍝ subframes. The
   4  5  6  7  9 10 11 12 13 14 15 16 17  ⍝ 1 1 2 operand
   8  9 10 11 18 19 20 21 22 23 24 25 26  ⍝  could as well be
                                           ⍝ 1 1
  12 13 14 15 27 28 29 30 31 32 33 34 35
  16 17 18 19 36 37 38 39 40 41 42 43 44
  20 21 22 23 45 46 47 48 49 50 51 52 53
```

Now vector cells, from corresponding rows, both planes vs. both planes (rank 1 cells, rank 1 trailing subframes, and rank 1 leading subframes for both). The arguments are:

```
     2 3 3⍴"my you it  t hedog"
my
you
 it

  t
 he
dog
```

and

```
     2 3 4⍴"hat  did is win rs  's  "
hat
 did
 is

win
rs
's
```

Applying `,@1 1 1` to them yields

```
     (2 3 3⍴"my you it  t hedog")(,@1 1 1
*     )2 3 4⍴"hat  did is win rs  's  "
my hat
you did
 it is
my win
yours
 it's
```

```
  that
 he did
dog is

  twin
 hers
dog's
```

Errors

Errors may be reported by either the operator or an operand:

```
     (⍳2 4 4),@1 1(⍳2 3 9)  ⍝ One trailing subframe a vector of length
,@1: mismatch               ⍝ four, the other a vector of length three.
*     →                     ⍝ Rank notes the discrepancy in lengths.

     (⍳24)(+@0)⍳23
+@0: mismatch               ⍝ Rank can't pair scalar cells for Add.
*     →

     (⍳24)(+@1)⍳23          ⍝ Rank can pair vector cells for Add,
 +: length                  ⍝ but Add can't pair scalar items.
*      →
```

Inner and Outer Products

If there is no Outer Product for `f`, `f@0 0 0` may be used.
Sine, cosine, tangent of 0 45 90 ... 360 degrees, with the result doctored to show `Inf`:

```
     8.4⍕ 1 2 3 ○@0 0 0 ○.25×⍳9
  0.0000  0.7071  1.0000  0.7071  0.0000 -0.7071 -1.0000 -0.7071  0.0000
  1.0000  0.7071  0.0000 -0.7071 -1.0000 -0.7071  0.0000  0.7071  1.0000
  0.0000  1.0000     Inf -1.0000  0.0000  1.0000    ¯Inf -1.0000  0.0000
```

If there is no Inner Product for `f` and `g`, `f/@(⍴⍴y) x g@0 ¯1 y` may be used (if there is no Reduction for `f` a defined reduction operator may be used). For `(⍳2 12)∧.<⍳12 24`:

```
     ∧/@2 (⍳2 12) <@0 ¯1 ⍳12 24
 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
```

10. Control Statements

Unlike functions and operators, which take arguments and return results, control statements are primarily used for their side effects, although they always do have explicit results. Some control statements also have syntax that differs from that of functions taking arguments. The keywords are *reserved*: the words or names case, do, if, else, and while can appear only as parts of control statements as shown in this chapter.

Error Reports

Multiple errors elicit but one report before suspension. Because these control statements are complex, the order in which the reports are listed here is not necessarily A+'s order of reporting in any particular situation. The five kinds of reports are:

- parse;
- value: an argument has no value;
- wsfull: the workspace is currently not large enough for the execution of the statement; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;
- a report from an included function;
- interrupt (not an error): the user pressed **c** twice (once if A+ was started from a shell) while holding the **Control** key down.

Definitions of Control Statements

Case

Syntax

There are two forms of case statement:

```
case (target) {value0; expression0;
               value1; expression1;
                  * * *
               valueN; expressionN;
               default}
```

and

```
case (target) {value0; expression0;
               value1; expression1;
                  * * *
               valueN; expressionN}
```

where `case` is a keyword, and `target`, `value0`, ... , `valueN`, `expression0`, ... , `expressionN`, and `default` are each an expression or expression group.

Definition

The values of `target`, `value0`, ... , `valueN` are subject to the requirement that `target∊valueN` must be a valid expression for each `value0`, ... , `valueN`. The case statement is evaluated as follows: the test

```
    1∊target∊value0
```

is made. If the result of the test is 1, then `expression0` is evaluated and its result is the case statement result. No further evaluations within the case statement are made. Otherwise, the expression `value1` is evaluated and if the result of the test

```
    1∊target∊value1
```

is 1, then `expression1` is evaluated and its result is the case statement result. This continues until all the expressions up to `valueN` have been evaluated and tested. If all tests fail and the expression `default` is present, then it is evaluated, and its result is the case statement result. If all tests fail and `default` is not present, the case statement result is the Null.

Additional Error Report:

- An error report is issued by Find (⍳) if the results of `target` and some `valuek` are not a suitable pair of arguments for it.

Examples

```
    matrix x : {
         case (ρρx) {
              0 ; 1 1ρx;
              1 ; ((ρx),1)ρx;
              2 ; x;
              ↑`rank }
         }

    ρmatrix 4
1 1
    matrix ι3
0
1
2
    matrix ι2 3
0 1 2
3 4 5
    matrix ι3 4 5
.matrix: rank
*      →
```

Do - Monadic (Protected Execution)

Syntax

`do expression` ,   where `do` is a keyword and `expression` is either an expression or an expression group.

Definition

The do statement used monadically *normally* performs a *protected execute* of `expression`. That is, with one exception, execution of `expression` is always completed, or at least abandoned; it is never suspended. The exception: a suspension, with a `stop` error message, can be triggered by the Stop function (monadic ^) when `` `stop `` is 1, and by a *few* other operations, such as an attempt to give a bound variable an impermissible value. See the last example for treatment of input during a suspension within protected execution.

Execution of monadic do is affected by the Protected Execute Flag, `` `Gf ``, as described below. `` `Gf `` is set by the command `$Gf`.

The result of the do statement is usually a two-element vector of the form `(error_code;result)`. If there is no error, the do result is always of that form, `error_code` is 0, and `result` is the result of `expression`.

If an error was encountered when evaluating `expression`, there are three possibilities, depending upon the value of `` `Gf ``:

- If `` `Gf `` is 1 (the normal case), then `error_code` is nonzero, and `result` is a character string holding the error message. The table "Error codes for Protected Execution" lists the error codes and messages.
- If `` `Gf `` is 2, then the do result has three items, the first two as just described and a third containing the value of `$si` at the time of the error.
- If `` `Gf `` is 0, then the execution is not protected and the error message and suspension occur just as they would for `expression` alone.

Error codes for Protected Execution

If Error Code is not zero, Error Name is the second element of the do result

Error Code	**Error Name**	**Example**
0	(no error)	
-1	segv, bus, or other error not detected directly by the interpreter, or a name from Signal	`do ↑'error1'` Signal function causes and names the error
1	interrupt	`do while(1)1` Interrupt: hold **Control** down, press **c** twice **-** once if A+ was started from a shell.
2	wsfull	`do 1e9⍴1`

3	stack	`recurse x: recurse x`
		`do recurse 0`

4	value	`do 1+abcd`
5	valence	`do ∊3`
6	type	`do ÷"a"`
7	rank	`do 0#0`
8	length	`do 1 2 + 3 4 5`
9	domain	`do 0÷0`
10	index	`do 9#⍳9`
11	mismatch	`do 1 2 +@0 ⊢ 3 4 5`
12	nonce	`do 1 2↑⍳3 5`
13	maxrank	`do (10⍴1)⍴1`
14	nonfunction	`do x@1 x←1`
15	parse	`do ⍎'+f'` Where `(f x:x)`, say.

16	maxitems	`do mp[,]←item` Too many items.
17	invalid	`do {a←'abc'; `a .is `label; a←⍳3}`
18	nondata	`do +{519;+}`

Monadic do's can be traced using `$dbg do`.

Unlike Execute, do cannot successfully execute a command, a function definition, or, unless in braces, an expression that must be alone on a line. Thus `0⍎'$vers'` and `0⍎'f{x}:{2×x+1}'` and `0⍎'+@0'` and `0⍎'÷'` all give nonerror results (as does unprotected Execute); `do $vers` and `do f{x}:{2×x+1}` and `do +@0` and `do ÷` all yield error results; and `do {+@0}` and `do {÷}` both give nonerror results.

Messages that are not strictly A+ error messages will still appear in the log, e.g.
filename: No such file or directory
not an `a object
and many adap messages.

Warning! Within a Monadic Do - as in a Protected Execute - Result (←*x*) exits from Monadic Do only, with a 0 return code and the Result argument as result; it does not exit from the function containing the Monadic Do.

Typically, the monadic do statement is used in the form:

```
     if (0=0⊃z←do ⍎x) {
            ... Normal processing for result 1⊃z}
     else { ... Error processing for error 1⊃z}
```

For an A+ process started from a shell, the Unix command `kill -INT` *process-id* can be used to interrupt protected execution, like **Ctl-c Ctl-c** from Emacs (or a single **Ctl-c** within a session started from a shell), and execution will continue with the next expression. If, however, this `kill -INT` signal is sent from the shell when protected execution is not in progress, it will cause A+ execution to halt.

Examples

```
     y div x: {
          case (0⊃z←do{y÷x}) {
           0; 1⊃z;    ⍝ Divide worked, return result.
           9; 0;      ⍝ Domain error, replace 0÷0 by 0.
           ↑⍕1⊃z }    ⍝ Failed for another reason, so report.
          }

     2 3 4 div 5 2 1
 0.4 1.5 4

     2 2 ¯2 0 0 div 5 0 0 5 0
 0.4 Inf ¯Inf 0 0     ⍝ "Fixed" 0÷0, making it 0.

     2 3 4 div 5 2
 .div: length
*     →

     2 3 4 div 'abc'
 .div: type
*     →
```

```
     a div a←5e6ρ2
 .div: wsfull
*     →

⍝ Treatment of input for immediate execution when stopped within protected
execution.

     no no no         ⍝ An expression that cannot be parsed.
⍝[parse] .no: var?




     $stop 1
     do {∧a←,.0; (n←100)do a[,]←(¯200+5×n)÷n-40; a}
⍝[error] : stop       ⍝ Now stopped within immediate execution.
*      9 9 9
 9 9 9                ⍝ Result is displayed for any correct input.
*      nonono         ⍝ Value error ignored: no suspension and NO display.
*      do nonono      ⍝ The remedy is to use protected execution in input.
<  4                  ⍝ Protected do is correct input; result is error it
found.
< value
*      no no no       ⍝ Parse error ignored: no suspension and NO display.
*      do no no no    ⍝ Entire input cannot be parsed: do is ineffective.
*      do ⍎'no no no'
<  15                 ⍝ do result shows the parse error found by ⍎ in its
arg.
< parse
*      ()⍎'no no no'  ⍝ ⍎ result is code for parse error it found in its arg.
 15
*      ←              ⍝ Continue from the point where stop occurred.
<  9                  ⍝ Result reports domain error from 0÷0.
< domain
```

Do - Dyadic (Iterative Do)

Syntax

`count do expression`, where `do` is a keyword and `count` and `expression` are either expressions or expression groups. (Typically, `count` is an expression.)

Definition

The value of `count` must be a restricted whole number (a scalar or one-element array). The expression named `expression` is evaluated `count` times. There are the same limitations on `expression` as were stated above in the definition of Monadic Do.

If the expression `count` is a (perhaps parenthesized) variable name alone or a parenthesized expression that makes an assignment to a variable name, then within `expression` that variable is successively given the integer values `0,1,...,count-1` for the successive evaluations. In these cases, once execution of the do statement is complete, the value of that variable is an integer equal to the value of the expression `count.`

If `count` is `(*varname)` or `(*varname←...)` then execution proceeds as if the `*` were not present (including the value of `varname` after the do is completed) and for the successive evaluations of `expression` the variable is successively given the same integer values *except that these values are given in decreasing order*.

The explicit result of the do statement is the result of the last evaluation of `expression`.

Result (`←x`) causes an exit from the function that contains the Protected Do.

Additional Error Report

- A domain error report is issued if the result of `count` is not a restricted whole number.

Examples

```
     z←0⍴0
     n←10
     n do z←z,n                 ⍝ Variable name alone:
 0 1 2 3 4 5 6 7 8 9            ⍝ n is counted up from 0
     n                          ⍝ to its original setting
 10

     z←0⍴0
     (i←10) do z←z,i            ⍝ Assignment to a variable name:
 0 1 2 3 4 5 6 7 8 9            ⍝ similar treatment of i
     i
 10

     z←0⍴0
     v←`i
     ((%v)←10) do z←z,%v        ⍝ Not just a variable name:
 10 10 10 10 10 10 10 10 10 10  ⍝ %v remains 10 throughout
```

If

Syntax

`if (condition) expression`, where `if` is a keyword and `condition` and `expression` are either expressions or expression groups; typically, `condition` is an expression. If `condition` is either a single number or name, or it is an expression group and therefore in braces, the parentheses are not necessary (although perhaps still a good idea for clarity).

Definition

The result of `condition` is a restricted whole number (a scalar or one-element array). If that result is nonzero then `expression` is evaluated, and the result of the if statement is the result of `expression`. Otherwise the result is the Null: i.e., the explicit result is Null if `expression` is not evaluated.

A note on entry of this statement: A+ considers `if (condition)` entered alone on a line, with no pending punctuation, to be a complete statement, taking `expression` to be null.

Additional Error Report:

- A domain error report is issued if the result of `condition` is not a restricted whole number.

Example

```
    if (0=ρρx) x←,x
```

If - Else

Syntax

`if (condition) expression1 else expression0`, where `if` and `else` are keywords, and `condition`, `expression1`, and `expression0` are either expressions or expression groups. Typically, `condition` is an expression. The parentheses are not necessary (although perhaps still a good idea for clarity) if `condition` is either a single number or name, or an expression group and therefore in braces.

Definition

The result of `condition` is a restricted whole number (a scalar or one-element array). If that result is nonzero, then `expression1` is evaluated. Otherwise, `expression0` is evaluated. The result of the if-else statement is the result of whichever of `expression1` or `expression0` was evaluated.

Additional Error Report:

- A domain error report is issued if the result of `condition` is not a restricted whole number.

Example

```
    if (10≥|x) x else 10××x
```

While

Syntax

`while (condition) expression`, where `while` is a keyword and `condition` and `expression` are each either an expression or an expression group. Typically, `condition` is an expression. The parentheses are not necessary (although perhaps still a good idea for clarity) if `condition` is either a single name or an expression group and therefore in braces.

Definition

The result of `condition` is a restricted whole number (a scalar or one-element array). If the value of `condition` is nonzero, then `expression` is evaluated. `condition` is evaluated again, and if its value is nonzero, `expression` is evaluated again. This continues until the value of `condition` is 0. The result of the while statement is the result of the last evaluation of `expression`. If `expression` is never evaluated, the result is the Null.

A note on entry of this statement: A+ considers `while (condition)` entered alone on a line without pending punctuation to be a complete statement, taking `expression` to be null. If you inadvertently enter such a statement, you can recover by pressing **Control** and holding it while you press **c** twice.)

Additional Error Report:

- A domain error report is issued if the result of `condition` is not a restricted whole number.

Example

```
     penny x: {
          m←0.01;
          i←1;
          while (m<x) {
               ↓i,m;
               i←i+1;
               m←m×2 }; }



     penny 1
1 0.01
2 0.02
3 0.04
4 0.08
5 0.16
6 0.32
7 0.64
```

11. System Variables

Classification of System Variables

Although they are listed alphabetically by English name in this chapter, for convenient reference, the A+ system variables can be grouped, among many other ways, in five categories, dealing with:

- error handling and debugging: Bus Error Flag, Callback Flag, Core File Size Limit, Dependency Flag, Execution Suspension Flag, Floating Point Exception Flag, K Stack, Protected Execute Flag, Segv Error Flag, Stack Information, Stop, X Events Flag;
- versions: Compiler, Dynamic Environment, Language Level, Major Release Number, Minor Release Number, Phase of the Release, Release Code, Version;
- implicit arguments to primitive functions: Context, Printing Precision, Random Link;
- handling of input: Input Mode, Standard Input, Terminal Flag;
- files: File Being Loaded.

Setting and Referencing System Variables

System variables cannot be set or referenced directly. The system functions Set System Variable, `_ssv`, and Get System Variable, `_gsv`, are used for this purpose. Most system variables can be set and referenced in this way; exceptions are noted in the individual descriptions.

In A+, variables and system variables are in completely distinct domains, with overlapping name spaces; a name's reference is determined not only by the name itself but also sometimes by the situation in which it occurs. As an example, consider the printing precision, `` `pp: _gsv `` is willing to accept `'pp'` for it, whereas `%` considers `` `pp `` just a way of referring to `pp`:

```
      pp←5
      pp
5
      _gsv `pp
10
      %`pp
5
      (_gsv 'pp'),⍎'pp'
10 5
```

Definitions of System Variables

Bus Error Flag `` `busexit ``

This flag controls the action when a bus error occurs. If it is 0 (the default), then the usual error message is issued and execution is suspended. If it is 1 or 2, A+ exits and a core dump is taken. On Sun machines, 1 and 2 have the same effect. On AIX machines, 2 causes the FULLDUMP flag to be set, whereas 1 produces a smaller dump. If `` `corelim `` is too small to allow a dump, only the exit occurs. A Monadic Do or Protected Execute is overridden when this flag is 1 or 2. To set up an exit and core dump on bus and segv errors, execute

```
      ⊣ `busexit `segvexit `corelim _ssv¨ (1;1;Inf)
or
      ⊣ `busexit `segvexit `corelim _ssv¨ (2;2;Inf)
```

or the like, and to re-establish the defaults, execute

```
      ⊣ `busexit `segvexit `corelim _ssv¨ 0
```

All messages from users reporting core files should be sent to email id **aplusdev** and should contain:

- The A+ release number involved.
- The machine on which it happened, and the machine's domain.
- The location of the core file.
- The output of the "where" stack. See instructions below for obtaining this output.
- All other information you would normally include in a bug report: the application, $si, what you were doing at the time, repeatability, etc.

To get the "where" information:

- From an XTerm, issue the command
  `dbx `which a+`` *`corefile`*
  with `corefile` replaced by the actual filename.
- After a minute or so, you'll see a `(dbx)` prompt. Enter
  `(dbx)` **`where`**
  and grab the ensuing output, sometimes quite lengthy, and cut and paste it into your message.
- Issue the command `(dbx)` **`quit`** to exit from `dbx`.

Callback Flag `` `Sf ``

The value of `` `Sf `` is either 0 or 1 (the default). See "Callback Flag" (`$Sf`) for the meaning of these values.

Compiler `` `CCID ``

Corresponding to the MSDE convention, identifies the compiler used to build A+. This variable is primarily intended for use by the `tidyld{}` tool to distinguish future 32 and 64 bit dyload objects.

Context `` `cx ``

The value of `` `cx `` is a symbol holding the name of the current context.

Core File Size Limit `` `corelim ``

The value of `` `corelim `` is the size of the largest permissible core file created by a failing A+ process; if the file size would be larger than this value the core file will not be created. A core file will always be created if the value is `Inf`. The default is 0, so that a core file is never created when the default setting is in effect.

Dependency Flag `` `Df ``

The value of `` `Df `` is 0 or 1 (the default). See "Dependency Flag" (`$Df`) for the meaning of these values.

Dynamic Environment `` `dyme ``

`` `dyme `` has the value -1 if Dynamic Load, `_dyld`, is not available. Otherwise, it has the value 0 in a Sun OS environment, except as noted below, 1 in an AIX environment, and 2 in Solaris and Irix.

The value is 2 for sunos.4.1.3, reflecting the use of the new Lexa compiler and support for dynamically loaded code. For developers and maintainers of `_dylded` functions, this change means that shared library versions will need to be built to work with this version.

To ease the conversion, the `_dyld{}` code will attempt to create a temporary shared library using the "old" nonshared object files, so simple uses of _dyld{} should continue to work, e.g., `"xxx.o" _dyld ("_xxx";"xxx";0 0).` This automatic creation of shared libraries will most likely fail with a more complex left argument that contains libraries or other linker options. Please send aplusdev a mail message if you need any assistance with dynamically loaded code or creating shared libraries.

`` `dyme `` cannot be set.

Execution Suspension Flag `` `Ef ``

The value of `` `Ef `` is 0 or 1 (the default). See "Execution Suspension Flag" (`$Ef`) for the meaning of these values.

File Being Loaded `` `loadfile ``

`` `loadfile `` is a character string giving the name of the file currently being loaded. It is null if no file is in the process of being loaded. (Typing `` _gsv `loadfile `` in an Emacs session yields the Null, since the statement won't be executed while a `$load` is underway.)

Note that, in the case of nested loads, only the name of the innermost file is returned. Also, if the file is `$load`ed with a relative pathname, `_gsv `loadfile` returns a relative pathname.

Floating Point Exception Flag `` `Xfpef ``

The value of `` `Xfpef `` is 0 (the default) or 1. When it is 1, domain errors that result when external routines generate SIGPFEs (floating point exceptions) are suppressed. When it is 0, they are not suppressed.

Input Mode `` `mode ``

The value of `` `mode `` is either `` `apl `` (the default) or `` `ascii ``. See "Input Mode" (`$mode`) for their meaning.

K Stack `` `si ``

The value of `` `si `` is a nested vector. It cannot be set. It represents the K stack, which is used by the State Indicator command, `$si`, and whose contents are subject to change, to improve debugging; furthermore, the format of `` `si `` is likewise subject to change. This variable includes all suspensions, unlike State Indicator, which displays only the latest. At present, there are two elements for each level of suspension: an enclosed Null (as a separator between levels) and a nested vector whose (character) elements are indicators, like `'expr'`, and expressions, like `'0÷0'`.

Language Level `` `language ``

The value of `` `language `` is either `` `a `` or `` `aplus ``. It cannot be set. It is the recommended way for toolkits that are used by A+ to determine the environment.

Major Release Number `` `majorRelease ``

The value of `` `majorRelease `` is the major release number of the currently running version of A+. See "Invoking A+". It cannot be set.

Mapped Files Limit `` `maplim ``

The value of `` `maplim `` is the number of files that can be mapped concurrently; the default is 2000.

Minor Release Number `` `minorRelease ``

The value of `` `minorRelease `` is the minor release number of the currently running version of A+. See "Invoking A+". It cannot be set.

Phase of the Release `` `phaseOfRelease ``

The value of `` `phaseOfRelease `` is `` `alpha ``, `` `beta ``, or `` `prod ``, according to the currently running version of A+. See "Invoking A+ from the Shell". It cannot be set.

Printing Precision `` `pp ``

The value of `` `pp `` is a nonnegative integer less than 100; its default value is ten. It specifies the maximum total number of digits to be used in the display of a number, not counting the two digits following the e in exponential notation, with two exceptions:

If `` `pp `` is zero, then it is treated as if it were one.

If `` `pp `` is less than ten, integers are nevertheless displayed with up to ten digits.

See "Printing Precision", Format (⍕) "Examples", and Default Format "Examples".

Protected Execute Flag `` `Gf ``

The value of `` `Gf `` is 0 or 1 (the default). See "Protected Execute Flag" (`$Gf`) for the meaning of these values.

Random Link `` `rl ``

The value of `` `rl `` is a nonnegative integer. It cannot be referenced. See "Random Link" (`$rl`).

Release Code `` `releaseCode ``

The value of `` `releaseCode `` is a character vector composed of the characters following `a+_` in the value of ATREE for the currently running version of A+. See "Invoking A+". It cannot be set.

Segv Error Flag `` `segvexit ``

This flag is exactly like the Bus Error Flag, except that it controls the action when a segv error occurs.

Stack Information `` `doErrorStack ``

Enable (1) or disable(0) the stack information on error facility. See `_doErrorStack{}`.

Standard Input `` `stdin ``

The values of `` `stdin `` are 1 for normal terminal input mode (the default, of course), and 0 to get the effect of Terminal Flag (`$Tf`). Unlike `$Tf`, when `` `stdin `` is 0, keyboard entries are queued to be processed when `` `stdin `` is subsequently reset to 1. Cf. the Terminal Flag system variable, `` `Tf ``.

Stop `` `stop ``

The values of `` `stop `` are 0, 1, or 2 (the default). See "Stop" (`$stop`) for the meaning of these values.

Terminal Flag `` `Tf ``

The value of `` `Tf `` is either 0 or 1 (the default). If 1, terminal input is normal. If 0, terminal input is ignored; see the Terminal Flag command, `$Tf`. Cf. the system variable `` `stdin ``, Standard Input.

Version `` `vers ``

The value of `` `vers `` is a character vector describing the currently running version of A+; it is the text following the phrase `This version is` that appears when A+ is invoked. It cannot be set.

X Events Flag `` `Xf ``

The value of `` `Xf `` is either 0 (the default) or 1. See "X Events Flag" (`$Xf`) for the meaning of these values.

12. System Functions

Classification of System Functions

Although they are listed alphabetically by English name in this chapter, for convenient reference, the A+ system functions can be grouped, among many other ways, in ten categories, dealing with:

- system variables: Get System Variable, Set System Variable;
- names and references: Expunge, Expunge Context, Locals, Name, Name Class, Name List, Valence;
- attributes: All Attributes, Get Attribute, Set Attribute, Get Client Data, Set Client Data;
- dependencies: All Dependent Object Names, Dependency Definition, Dependent Object Names, Remove Dependency Definition;
- callbacks: Get Callback, Get Preset Callback, Set Callback, Set Preset Callback;
- files: Abort Loading of Script, Change Directory, Dynamic Load, Items of a Mapped File, Load a File, Load and Remove a File;
- format changes: Association List to Slotfiller, Comma Fix Input, Fix Input, Flatten, Format, Scalar Comma Fix Input, Scalar Fix Input, Screen Format, Get Format Symbols, Is a Slotfiller;
- text functions: General Search and Replace, Index Of, Name Search, Name Search and Replace, String Search, String Search and Replace;
- indexing: Permissive Indexing;
- debugging: Debug, NaN Find, Stack Information;
- A+ operation and space: Execution Profile, Exit, Hash Table Statistics, Work Area.

Querying Syntax

To inquire about the syntax of a system function or external function in an A+ session, enter its name alone on a line. For example:

```
      _nl
 _nl{any;any} returns any
```

Because some of the system functions take more than two arguments, they are all shown here in general form, for uniformity, and that is how a syntax query is answered. Nevertheless, infix notation can also, of course, be used to invoke the monadic and dyadic system functions.

Common Error Reports

Multiple errors elicit but one report. If an error report in the following list is issued, then the ones preceding it do not apply. Seven reports are common to all system functions:

- parse;
- value: an argument has no value;
- valence: the wrong number of arguments were given;
- nondata: the argument is a function or some other nondata object;
- type or domain: an argument is of the wrong type (nested when it should be simple, numeric when it should be a symbol, and so on) or the wrong class (e.g., a symbol, but not from some required set of symbols); the same error is called type by some functions and domain by others;
- length: a single number is required and the ravel of the argument does not have a length of one;
- wsfull: the workspace is currently not large enough to execute the function in; a bare left arrow (←), which dictates resumption of execution, causes the workspace to be enlarged if possible;

- interrupt (not an error): the user pressed **c** twice while holding the **Control** key down.

Definitions of System Functions

The Name Argument

Several system functions take names as arguments. A name argument is one of the following:

- a symbol holding a valid A+ name, e.g., `` `var `` or `` `ctx.var ``;
- a pair of symbols holding a valid A+ context name and a valid name, e.g., `` `ctx `var ``.

These names always refer to global objects. See "Symbols and Symbol Constants".

The definition of a system function refers to this section if it has an argument that conforms to this definition.

Abort Loading of Script `_abortload{}`

Definition

This system function allows the user to terminate the `$load`ing of a file. Executed in a script, it will terminate that script after the current line is processed. If one script loads a second, and the second script calls `_abortload{}`, the second script ends immediately, but the first script is not aborted, and continues to be loaded. While the loading of a script is suspended - by an interrupt, error, Stop, or Signal, for example -, entering `_abortload{}` in the (Emacs or XTerm) session log for execution will terminate its loading, without affecting the suspension.

All Attributes `_atts{y;x}`

Argument and Result

The arguments *x* and *y* are described in "The Name Argument". The result is a vector of symbols.

Definition

Lists all attributes set by `_set` on *y* with prefix *x*. E.g., `` _atts{`a;`appl} `` lists all attributes of the form `` `appl.* `` set on *a*. If *x* is Null, lists all attributes set by `_set` on *y*. The attributes are listed in random order. If the variable *y* does not exist, the result is empty. For a list of attributes set by s (which does not currently use `_set`), i.e., display attributes, use the settings attribute.

All Dependent Object Names `_alldep{x}`

Argument and Result

The argument *x* is described in "The Name Argument". The result is a vector of symbols.

Definition

The result is the transitive closure of `_dep{x}` (Dependent Object Names): it returns the dependency set of *x*, to wit, a list containing `_dep{x}`, `_dep¨_dep{x}`, etc., with duplicates removed.

Note that if *f* is a dependency and `` `f `` is contained in the vector `_alldep{`f}`, then the dependency set `_alldep{`f}` is cyclic. See "Cyclic Dependencies".

Association List to Slotfiller `_alsf{x}`

Argument and Result

The argument is an association list or a slotfiller. The result is a slotfiller (but see below).

Definition

This function converts an association list to the equivalent slotfiller. It also accepts slotfillers and returns them unchanged, so that it can be used to obtain a slotfiller from an array that might be either an association list or a slotfiller; no checking is necessary.

An argument that is an association list whose length is odd is accepted in two cases. If the last element is a symbol, a Null is appended before conversion. If the last element is a Null, it is dropped.

In one case the result is not actually a slotfiller: if there are duplicates among the symbols that are to be the elements of the first item of the result, the argument is nevertheless accepted and the duplicates are retained.

Example

```
     _alsf(`a;1;`b;2;`c;3 4 5)
<  `a `b `c
<  < 1
   < 2
   < 3 4 5
```

Change Directory `_cd{x}`

Argument and Result

The argument *x* is a symbol or character vector. The result is a nested vector with either one or two elements.

Definition

`_cd{x}` provides the same function as the `$cd` command. The argument *x* specifies the new current directory in Unix format. If the change to the new current directory is successful, the result is `1ρ<`ok`; otherwise it is a two-element array of the form `(`error;"error message")`, for example if there is no such directory. The PWD environment variable is set.

After a file is loaded, the current directory is the same as it was when the Load function or command was initiated: it is automatically restored if it was changed during execution of the lines of the file.

Comma Fix Input `_cfi{y;x}`

Arguments and Result

y is a simple character array of rank less than or equal to two, and *x* is a scalar restricted whole number. The result is a two-element nested array of the form `(bool;data)`, where `#bool` equals `#data` if *y* is a matrix and equals 1 otherwise, and `bool` is an integer array of rank `0⌈¯1+ρρy`, and `data` is a numeric array of rank `1⌈ρρy` with `¯1↑ρdata` equal to `x⌈0`.

Definition

See "Fix Input" (`_fi`). The definition of `_cfi` is the same as Fix Input, except that `_cfi` permits additional representations (and misrepresentations) of numbers. Specifically, commas and parentheses are permitted in number representations, as well as minus signs, low and the A+ high (¯). In producing the numeric values, commas are ignored no matter where they appear, and high and low minus signs and each set of enclosing parentheses is replaced by a high minus sign. In other words, commas are not treated as "thousands" separators (although they can be playing that role), and all parentheses and `-` and ¯ are treated as signifiers of negative quantities. For each result number, there can be at most one indication that it is negative: `'-¯16'`, `'(-16)'`, and the like are not accepted. The parentheses that indicate a number is negative must surround the entire number, those that indicate an exponent (in e-notation) is negative just the exponent itself. A left parenthesis may be followed by blanks. Note: blanks between numbers are not required, although their absence is obviously inadvisable; in other words, *missing blanks are not used as an indication of possible error any more than missing or extraneous commas are*. `'1e-3.5-7.6.5'` is translated as `1e-3 0.5 ¯7.6 0.5`.

Scalar Comma Fix Input, `_scfi`, is expected to be used in `` `in `` functions, validating user input. `_cfi`, on the other hand, is designed for larger arrays, such as text files; the assumption is that the data argument is likely to have been machine generated.

Additional Error Reports

- Nonce error if *x* is greater than 1000.
- Rank error if the rank of *y* exceeds two.

Example

```
     _cfi{'(1,200)';1}
<   1
<   ¯1200.
```

Debug `_dbg{x}`

Argument and Result

This function takes essentially the same arguments as `$dbg` and returns essentially the results displayed by it.

Definition

Provides the functionality of `$dbg` with different syntax: e.g., ``_dbg{`cxt;(`only;`)}`` to trace only the root context. Additionally:

(1) ``_dbg{`display;`beam}`` returns a slotfiller with information about all files currently beamed in. The six fields of the slot filler contain entries for each file:
`` `mode `` - 0, 1, or 2: the file is copy-on-write, read/write, or local write.
`` `arg `` - the righthand argument given to *I̲*.
`` `fname `` - the actual file name `` `arg `` was turned into.
`` `addr `` - the memory address where the file was mapped.
`` `refcnt `` - reference count: how many A-objects point to the file.
`` `bytes `` - the size of the file in bytes.

(2) `_dbg` allows for callbacks at the points where a tracing or tracking message would be displayed. Executing `_dbg{`cb;(fn;cd)}` sets up the callback function with its client data (static). This function must take five arguments: `fn{cd;event;arg1;arg2;arg3}`. The events are `` `func``, `` `xfs``, `` `sfs``, `` `load``, `` `beam``, and so on; `arg1` is the name of the function, file, or whatever involved; the possible values of `arg2` are `` `enter``, `` `exit``, `` `abort``, `` `in``, `` `out``, `` `unmap``, `` `dep``, `` `func``, and how many `Inf`s and how many NaNs; and `arg3` is usually null, but for beaming in it is the left argument of `I̲`.

(3) In the event callback for `` `inv`` events, the second argument is null for normal invalidation, and equals `` `cycle`` if the `` `inv`` event is the detection of a cyclic dependency.

Dependency Definition `_def{x}`

Argument and Result

The argument `x` is described in "The Name Argument". The result is a character vector.

Definition

The result holds the definition of the dependency named in `x`. See "Dependencies". If `x` does not name a dependency, the result is Null.

Dependent Object Names `_dep{x}`

Argument and Result

The argument `x` is described in "The Name Argument". The result is a vector of symbols.

Definition

The result contains the names of all dependencies whose definitions explicitly reference the object named in `x`. See "Dependencies".

Dynamic Load `_dyld{x;y}`

Arguments and Result

The argument `x` is a character vector and the argument `y` is a nested vector. The result is a scalar integer.

Definition

The effect of this system function is to load C functions into A+ so that they can be used like ordinary user functions. (But the fixed number of arguments for a C function being `_dyld`ed cannot exceed 8.) The details can be found in "Calling C Subroutines from A+". System error messages are also displayed. If this function is not available, the Dynamic Environment system variable, `` `dyme``, has the value -1. It is safe to `_dyld` an object already `_dyld`ed.

Warning! A cover function that takes care of the requirements of different architectures should be used rather than a direct invocation of `_dyld`, except when "hacking".

Warning! If the A+ name used in `_dyld` (in its right argument) begins with an underscore, then the function will be installed in the root context, no matter what the current context, and it will be listed by `$sfs` but not by `$xfs`.

Execution Profile `_profile{x}`

Argument and Result

A meaningful argument is a scalar symbol, one of `` `on `off `report `reset``. If the argument does not cause an error - i.e., if it is a one-element array - then the explicit result is the scalar integer 0.

Definition

This facility is enabled by executing `` _profile `on`` and disabled by `` _profile `off``. When not enabled, it adds absolutely no overhead to primitive function calls.

When it is enabled, executing `` _profile `report`` causes a report to be displayed that gives the number of primitive function executions by function, data type, and array size range (number of elements) since the last time `` _profile `on`` or `` _profile `reset`` was executed.

Exit `_exit{x}`

Argument and Result

The argument `x` is a scalar integer. There is no result.

Definition

The expression `_exit{x}` provides a similar function to the system command `$off`. The argument `x` is the value returned when the A+ process terminates. Proper values for `x` are 0 through 255, although any integer is accepted; by convention, the value 0 means success. Since execution of this system function causes the A+ process to exit, it has no meaningful result. See "Invoking A+".

Expunge `_ex{x}`

Argument and Result

The argument `x` is described in "The Name Argument". The result is an integer scalar.

Definition

`_ex` provides the same function as `$ex`. The object named in `x` is removed and its associated memory is freed. The result is 0 if the removal is successful, and 1 otherwise. In particular, the result is 1 if the object does not exist. It is also 1 if `x` is an object bound to a display class (see "is" and "free"); i.e., a bound object cannot be expunged.

Expunge Context `_excxt{x}`

Argument and Result

The argument `x` is described in "The Name Argument". The result is an integer scalar.

Definition

The context named in `x` must be empty. If it is, `_excxt{x}` removes it from the active workspace and returns 0. If the context is not empty, `_excxt` returns 1. The Name List system function with arguments `x` and `` `globs `` can be used to list all objects that would prevent the expunging of a context. See the "Name List" (`_nl{y;x}`) function. See also "Expunge Context" (`$excxt`).

The root context cannot be expunged; an attempt to expunge it yields a result of 1. For a nonexistent context, the result is 1.

Fix Input `_fi{y;x}`

Arguments and Result

`y` is a simple character array of rank less than or equal to two, and `x` is a scalar restricted whole number. The result is a two-element nested array of the form `(bool;data)`, where `#bool` equals `#data` if `y` is a matrix and equals 1 otherwise, and `bool` is an integer array of rank `0⌈¯1+ρρy`, and `data` is a numeric array of rank `1⌈ρρy` with `¯1↑ρdata` equal to `x⌈0`.

Definition

This function creates a numeric array with `x⌈0` elements derived from each row of `y` (or from `y` itself, if it is a scalar or vector). If a valid vector is created for the row `i#y`, it becomes `i#data`, and `i#bool` is assigned 1. If `i#y` does not contain the representations of exactly `x⌈0` numbers, then `i#bool` is assigned 0, and `i#data` is created by using the first `x⌈0` valid nonblank sequences in `i#y`, and using zeros as fillers if there are not enough valid nonblank sequences. (The term nonblank suggests how you should use the function, with blanks as separators, but see the note with which this definition ends.) If there is a nonblank sequence in `i#y` that does not represent a number, then `i#data` contains zeros from the point of invalidity on.

Each row of the argument `y` for which a valid numeric vector can be created must consist of `x` common representations of numbers. A common representation is one of the three general forms produced by the Default Format monadic primitive function `⍕`, namely integer, fixed-point, and exponential, except that low and high minuses are accepted indifferently: a low minus is accepted at the beginning of any number and in exponential form a high minus is accepted in the exponent. The expression

```
     bool/⍳#y
```

yields the row indices of `y` for which a valid numeric vector was created. Note: blanks between numbers are not required. `1e-3.5 7.6.5-2` is translated as `1e-3 0.5 7.6 0.5 ¯2`.

Additional Error Reports

- Nonce error if `x` is greater than 1000.
- Rank error if the rank of `y` exceeds two.

Examples

```
     ↓a←3 5ρ'1.34 2 3  2x50 '
1.34
2 3
2x50
     a _fi 1
< 1 0 0
<  1.34
   2
   2
     a _fi 2
```

```
< 0 1 0
<  1.34 0
   2    3
   2    0
```

Flatten _*flat*{*x*}

Argument and Result

The argument *x* is any array whose simple components are of one type, except that symbols and functions may be mixed and nulls are essentially ignored. The result is a simple vector whose length is the total number of elements in all the simple components of *x* and whose type is that of the first nonempty simple component of *x*, if *x* is not empty.

Definition

The simple components of *x* are all those simple objects obtainable by repeated selection and disclosure. If they are all empty, the result is an empty vector whose type is that of the nonnull ones; if they are all null, the result is the Null. If *x* is simple, the result is the ravel of *x*. Otherwise, the result is the catenation of the ravels of all the simple components of *x*.

Additional Error Report

- A domain error is reported if two simple components of *x* (including empty arrays whose types are not null) are of different types, except that symbol and function may both be present.

Examples

```
    _flat (ι3 5;23 34 42;0;ι0;;<<(<¯1 ¯2),<1 2ρ7 8)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 23 34 42 0 ¯1 ¯2 7 8
    _flat `a,<2 2ρ`b`c`d`e
 `a `b `c `d `e
    (`a,<{+})≡_flat(`a;+)
1
```

Format _*fmt*{*y*;*x*}

Arguments and Result

y is a character vector or scalar, and *x* is a simple character or numeric array or a nested array. The result is a simple character array of rank less than or equal to 2.

Definition

The function _*fmt* transforms character and numeric data in *x* into a character table according to specification *y*, which consist of a series of format phrases - see the "Format Phrases" table - separated by commas. For example, the string `'i6,f8.2,f6.1'` contains three phrases. Each phrase governs the appearance of a column in the result. The number of phrases need not match the number of items in *x*, the phrases being repeated cyclically as needed; furthermore, tab and constant specifications are in separate phrases.

Each phrase has a type, indicated by a letter, such as the letters `i` and `f` in the above example. A tab or constant field in `y` does not correspond to anything in `x`. A tab field merely contributes to the positioning of the next part of the output, and a constant field is reproduced in the output exactly as shown.

Within each format phrase, to the left of the letter that identifies its type, *qualifiers* or *decorators* may be inserted. A qualifier provides additional rules, such as "insert commas between triplets of digits" or "leave this field blank for zero". See the "Qualifiers" table. A decorator is a code letter immediately followed by a piece of text enclosed in angle brackets or whatnot; the text is to be attached to the representation of a number, usually to indicate its sign. See the "Decorators" table.

When rounding is required, IEEE rules are used: see "Format". When a number cannot be formatted in the specified manner, its field is filled with asterisks. A field whose format is valid but for a different type from the data is filled with question marks. In particular, symbols in `x` do not cause an error message, but the field in which a symbol is to be formatted is filled with question marks.

Normally, `x` is a matrix, and the application of the formatting phrases is straightforward. It can be of other ranks and of any depth, however. In that case, the columns to be formatted are obtained, conceptually, by the following process. When a nested array is encountered, it is ravelled, and its elements are successively disclosed and treated in the manner being described. When a simple array is encountered, it is transformed into a matrix if it is not already one: a scalar becomes a 1 by 1 matrix, an n-element vector becomes an n by 1 matrix, and any other array `a` is reshaped as `((×/¯1↓ρa),¯1↑ρa)ρa`. This produces a series of matrices, and thus a series of columns, to which the formatting phrases are applied as stated above. Each formatted column can be viewed as a character matrix; the result has the same number of rows as the one with the most rows. Conceptually, blank rows are appended as necessary to each of these matrices to make them the same length as the result, and they are catenated side by side (`,@0`) to form the result.

In the "Format Phrases" table, *j* is an optional repetition factor; *q* represents an optional qualifier and decorators; *w* is field width; *d* denotes significant digits or decimal digits; *p* is displacement; `a`, `e`, `f`, `i`, `t`, and `x` are literals giving phrase types.

Format Phrases

Phrase	Application	Description
j`a`*w*	Character	Display each character in a field *w* positions wide. A *w* greater than 1 produces blanks between adjacent characters.
jq`e`*w*`.`*d*	Exponential	Display each number in a field *w* positions wide, with *d* significant digits. Align *E*'s and decimal points in fields that are not left justified.
jq`f`*w*`.`*d*	Fixed point	Display in a field *w* positions wide, with *d* decimal digits, including trailing zeros. Leading zeros are shown as blanks unless the `z` qualifier is used.
jq`i`*w*	Integer	Display in a field *w* positions wide. Leading zeros are shown as blanks unless the `z` qualifier is used.

jqg<*txt*>	Picture Format, rounding arguments to integers	In *txt*, `9` and `z` are "special characters"; the rest are "ordinary." Round each number to an integer, after scaling if `k` is present. If `s` is present, modify *txt* as required. Then format each number with as many digits as there are special characters in *txt*, so each digit corresponds to a special character. Display a digit corresponding to a `9` unconditionally. If a digit corresponding to a `z` is in a sequence of leading or trailing zeros, display a blank; otherwise, display the digit. Display ordinary characters as is, with one exception, intended to exclude superfluous thousands separators but include decimal points: any ordinary character surrounded by special characters is displayed as blank if in the display a blank is next to it, unless it is the last ordinary character and is followed by at most two characters. The only valid decorators and qualifiers are `m` and `p` (but only the first character of the text of each), `s`, and `k`.
`jt`*p*	Absolute tab	Start the next field at position *p*+1. If *p* is too small, the next field may overwrite earlier ones.
jxp	Relative tab	Start the next field at a position displaced *p* positions from the end of the preceding field. If *p* is negative, the next field may overwrite earlier ones.
<*text*>	Constant: for all	Insert *text* in every row of the result.

Qualifiers

Qualifier	**Summary**	**Description**
`b`	blanks	Make the entire field blank when the value is zero.
`c`	commas	Insert commas between successive triplets of digits in integer and (before the decimal point) fixed-point fields.
`k`*e*	scale	Scale the result by displaying a value that is 10^e times the value.
`l`	justify	Left justify. The default is right justification.
`z`	zeros	Display leading zeroes in integer and fixed-point fields.

In the "Decorators" table, *text* may also be delimited by ≤ ≥, ⊂ ⊃, ¨¨, / /, ⎕⎕, ⍞⍞.

Decorators

Decorator	**Summary**	**Description**
`m`<*text*>	Negative-Left	Insert *text* to the left of a negative value, not APL high minus. *text* can be empty. `m` and `n` can be used together.

n<*text*>	Negative-Right	Insert *text* to the right of a negative value rather than use APL high minus. *text* can be empty. *m* and *n* can be used together.
o<*text*>	Zero replacement	When the data as formatted is zero, replace the result by *text*. Valid for the *e*, *f*, and *i* phrase types.
p<*text*>	Positive-Left	Insert *text* to the left of a positive value. *p* and *q* can be used together.
q<*text*>	Positive-Right	Insert *text* to the right of a positive value. *p* and *q* can be used together.
r<*text*>	Background	Fill the field cyclically with *text* where normally blanks would appear, including blanks from a *b* qualifier. For *e*, *f*, and *i* phrase types.
s<*text*>	Symbol Substitution	For each pair of characters in *text*, the second indicates a character to replace the first. In *e*, *f*, and *i* phrase types, punctuation, sign, and error characters in the result can be replaced; in a *g* phrase type, any characters in the picture text and error characters in the result can be replaced; no other replacements can be made. For example, *s*<,..,> in an *f* phrase type produces periods where commas usually appear and commas where periods usually appear.

Examples

```
     'e9.4' _fmt 200 .987
 2.000E 2               ⍝ Internal blank used to align decimal points and
E's.
 9.870E¯1


     n←123456789 0 ¯123456789 123456
     'bcm<->k-2f13.2' _fmt n
 1,234,567.89

-1,234,567.89
     1,234.56


     'm<(> n<)> q< > b c k-2 f16.2' _fmt n
   1,234,567.89                ⍝ Blanks are okay in the left argument.

  (1,234,567.89)
       1,234.56


     'i 6' _fmt `wrong `type
??????
??????


     'g<99:99:99>' _fmt 100⊥3↑3↓sys.ts{}
07:19:30


     'g<zz9>' _fmt .49 .51 99.47
  0
  1
 99


     rfnd← 1.23 10.00 12345.67 332 1234.56
     'k2 g<Your refund is: $z,zz9.99>' _fmt rfnd
Your refund is: $    1.23 ⍝ k2 applied to 1.23 is 123, and the
Your refund is: $   10.00 ⍝ decimal point in 1.23 is from g.
************************* ⍝ 12,345.67 is too long; if we had s<*?>
Your refund is: $  332.00 ⍝ in the left arg, *...* would be ?...?.
Your refund is: $1,234.56 ⍝ The comma is needed here, and it appears.


     'k2 g<Your refund is: $z,zz9.zz>' _fmt 1.23 10.00
Your refund is: $    1.23
Your refund is: $   10.   ⍝ The decimal point is the last ordinary
                          ⍝ character and it has only two characters
                          ⍝ after it,so it's shown even when the
                          ⍝ trailing zeros are dropped.

     's<9#z$>g<$$$# units of product z9>' _fmt 73
  73 units of product z9  ⍝ s allows 9 and z to be included literally
                          ⍝ in the output.
```

General Search and Replace `_gsr{y;x;r;n}`

Arguments and Result

`y`, `x`, `r`, and `n` are all simple character arrays, normally vectors, except that `r` and `n` can also be null. The result is either a simple character array or an integer vector or matrix.

Definition

If `x`, `r`, or `n` is not a vector, its ravel is used. There are eight cases, as determined by `n` and `r`:

- If `r` is Null, then:
  - If `n` is Null, then the function is the same as String Search (`_ss`).
  - If `n` is not Null, then the function is the same as Name Search (`_ns`), except that `n` affects the definition of "namelike" as follows:
    - if `n` is the empty character vector (`''`), then the definition is exactly the same as for `_ns`;
    - if `n` is nonempty and `0#n` is not `'⋄'`, then `n` replaces `'._'`, and `` '`¯' `` are not considered;
    - if `n` is `'⋄',c`, then `c` replaces the alphabetic characters, digits, and `'._'`, and `` '`¯' `` are not considered.

- if `r` is not Null (but possibly `''`), then:
  - If `n` is Null, then the function is the same as String Search and Replace (`_ssr`).
  - If `n` is not Null, then the function is the same as Name Search and Replace (`_nsr`), except that `n` affects the definition of "namelike" as follows:
    - if `n` is the empty character vector (`''`), then the definition is exactly the same as for `_nsr`;
    - if `n` is nonempty and `0#n` is not `'⋄'`, then `n` replaces `'._'`, and `` '`¯' `` are not considered;
    - if `n` is `'⋄',c`, then `c` replaces the alphabetic characters, digits, and `'._'`, and `` '`¯' `` are not considered.

Examples

Name delimiters are okay within target and replacement:

```
     _gsr{'⎕IO←1';'⎕IO';'∆IO';'⎕∆⍙'}
∆IO←1

     _gsr{'⎕⎕IO←1,⎕IO←2,B⎕IO←3';'⎕IO';'∆IO';'⎕∆⍙'}
⎕⎕IO←1,∆IO←2,B⎕IO←3

     _gsr{'⎕⎕IO←1,⎕IO←2,B⎕IO←3';'⎕IO';'∆IO';'⋄⎕∆⍙'}
⎕⎕IO←1,∆IO←2,B∆IO←3


     _gsr{'Mississippi is';'is';'IS';''}
Mississippi IS

     _gsr{'Mississippi is';'is';'IS';}
MISsISsippi IS
```

Get Attribute `_get{y;x}`

Argument and Result

The arguments *x* and *y* are described in "The Name Argument". The result is an array.

Definition

If `_set{y;(x;z)}` (Set Attribute) has been executed previously, then the result of this function is *z*, the value for attribute *x* on variable *y*. Otherwise, the result is Null, including the case when *y* does not exist.

Get Callback `_gcb{y}`

Argument and Result

The argument *y* is described in "The Name Argument". The result is a two-element nested array.

Definition

If `_scb{y;x}` (Set Callback) has been executed previously, then the value of `_gcb{y}` is the value of *x* in the last such setting: function scalar, static data. Although the name displayed for the callback function may now have a new reference, `⍕0⊃_gcb{y}` gives the definition of the callback function. Otherwise the value is Null.

Get Client Data `_gcd{y}`

Argument and Result

The argument *y* is described in "The Name Argument". The result is an array.

Definition

This function is like `_get` (Get Attribute), but for an implicit, unnamed attribute. It gets the value that was set by the last execution of `_scd{y}` (Set Client Data); if none was set, it returns the Null. It has been superseded by Get Attribute and is retained only for compatibility.

Get Format Symbols `_gfmtsym{}`

Result

The result is a vector of symbols.

Definition

This function lists the symbols acceptable as values for the out attribute. See the "out Attribute Format Samples" table.

Get Preset Callback `_gpcb{y}`

Argument and Result

The argument `y` is described in "The Name Argument". The result is a two-element nested array.

Definition

If `_spcb{y;x}` (Set Preset Callback) has been executed previously, then the value of `_gpcb{y}` is the value of `x` in the last such setting: function scalar, static data. Although the name displayed for the preset callback function may now have a new reference, you can obtain the definition of the preset callback function by entering `⍕0⊃_gpcb{y}`.

Otherwise the value is Null.

Get System Variable `_gsv{x}`

Argument

The argument `x` is a symbol scalar or a quoted string naming a system variable whose value the user is allowed to access. The result is the value of that system variable.

Definition

The value is the current value of the system variable named in `x`. A domain error is reported if `x` does not name a system variable or access to the system variable named in `x` is forbidden, as it is for `` `rl``. See "System Variables" for descriptions of the system variables and their values and interdictions.

Additional Error Report

- A domain error is reported if `x` does not name a system variable whose value is accessible.

Hash Table Statistics `_hashstat{x}`

Argument and Result

The argument is Null or a symbol naming a context. The result is an integer vector.

Definition

The result gives the number of entries in each bucket of a hash table. If the argument is Null, the statistics are for the symbol table and the result length is 1024. If the argument names a context, the statistics are for the hash table for that context and the result length is 512. The function `+/` applied to the result indicates the number of names that have ever been defined, while `⌈/` gives a rough indication of collisions and hence possibly reduced execution speed (although for this purpose the whole vector should be examined for clumping).

Index of `_index_of{y;x}`

Arguments and Result

y and *x* must be nested vectors of the same length. For each of them, its elements must all be nested vectors of the same length, or all be enclosed scalars. For *y*, a scalar is treated as a one-element vector. The result is a simple integer vector or scalar whose shape is the same as an element of *x*.

Definition

Call *y*'s contained vectors `y1...yn` and *x*'s contained vectors `x1...xn`. The result `r` is an integer vector of length `⍴x1`; `r[i]` is the least `j` for which `xk[i]` is equal to `yk[j]` for all `k`; if there is no such `j`, then `r[i]` is `⍴y1`. This function is a high performance version of the expression `(⍉>y⍳¨y)⍳⍉>y⍳¨x`.

Regarding the arguments as matrices whose columns are the contained vectors, so that the matrices have the same number of columns, `_index_of` yields a vector that gives, for each row of the right argument, the index of the first row of the left argument that it matches; for any row of the right argument that does not have a matching row in the left argument, the corresponding element of the result is the number of rows in the left argument, i.e., the common length of its contained vectors.

Example

Consider a tiny table with three fields: styles, colors, and prices. The style and color fields can be used as key fields with `_index_of` to find row indices of the table, and these can be used with Pick to give prices:

```
   Inventory←(("Oxford";"Oxford";"Loafer";"Loafer");
*             ("Brown"; "Tan";   "Black"; "Brown" );
*             (128.98;  175.98;  112.50;  112.50))
   (Inventory[0 1]_index_of(<"Oxford";<"Tan"))⊃2⊃Inventory
 175.98
```

Additional Error Reports

- A length error is reported if the arguments are not the same length.
- A domain error is reported if the contained vectors of an argument are not of the same length.

Is a Slotfiller `_issf{x}`

Argument and Result

`x` is any array. The result is a scalar boolean integer.

Definition

If `x` is a slotfiller, the result is 1. Otherwise the result is 0, e.g., if there are repetitions among the items of `0⊃x`.

Items of a Mapped File `_items{y;x}`

Arguments and Result

The argument `y` is a one-element numeric array whose element is a restricted whole number, while the argument `x` is a symbol or character scalar or character vector. The result is a scalar integer.

Definition

The argument `x` holds the name of a mapped file (see "Mapped Files"). A mapped file can be processed within A+ as if it were an ordinary array. In particular, it makes sense to speak of the items of a mapped file. If `y` is -1, the result is the number of currently allocated items. If `y` is nonnegative and less than the current number of items, then the effect is to truncate the file. Otherwise, if `y` is nonnegative the effect of this function is to allocate space (if necessary) on the end of the file so that the file can hold `y` items. See "Work Area" (`_wa`) regarding units of storage allocation. The result is the current number of items before the new allocation. See "Examples", and note that the "Operation would block" message and the delay caused by trying to avoid that message can occur for `_items`.

When a file is written out as a mapped file, the file is made only large enough to hold the actual data.

After a file length is changed by `_items`, any variables that were associated with it (by Map or Map In) must be remapped (by a new application of Map or Map In), because `_items` has rewritten the file.

Load `_load{x}`

Argument and Result

The argument `x` is a symbol scalar or a character vector. The result is a two-element nested array.

Definition

The expression `_load{x}` provides the same function as `$load x` (Load command). The argument `x` names the A+ script file to be loaded. The result is `(`ok;`*file*`)` if the file is found and can be read, where *file* is a character vector giving the fully qualified name of the file that is loaded, i.e., path/filename. After a file is loaded, the current context and the current directory are the same as they were when the Load function was invoked: each is automatically restored if it was changed during execution of the lines of the file.

If the file is not found the result has the form `(`error;"`*error message*`")`. Note that a result `(`ok;`*file*`)` does not mean that the file was loaded without error, just that it was found and can be read.

Example

```
     _load 's'
< `ok
< /usr/local/lib/s.+
```

Load and Remove `_loadrm{`delete;x}`

Argument and Result

The right argument `x` is a symbol scalar or a character vector. The result is a two-element nested array.

Definition

`_loadrm{`delete;x}` provides the same function as `$loadrm x` (Load and Remove command). The argument `x` holds the name of the file to be loaded. The file is loaded and then deleted (this is a special function for the Emacs environment). The result is the same as that of `_load`. The left argument must be `` `delete `` or `'delete'` or an error message is issued.

Locals `_locals{x}`

Argument and Result

`x` is a symbol, function expression, or function scalar representing a defined or external function; it cannot be a primitive or derived function. The result is a three-element nested array.

Definition

The first element of the result is the name of the function, as a symbol. If `x` represents a defined function, the second element is a vector of the names of the arguments to the function and the third element lists its local variables. If `x` represents an external function, the second element is an integer vector of its argument types (see table in "Calling C Subroutines from A+"), and the third element is null.

Name `_name{x}`

Argument and Result

The argument `x` is a function scalar. The result is a scalar symbol.

Definition

`x` is of the form `<{fn}`, where `fn` is a defined, primitive, or system function. For a defined function, the only case that really makes sense, the result holds the fully qualified name of `fn`, i.e., the name of the current context, which will be empty for the root context, followed by a dot as separator, followed by `fn`. If `fn` is a system or primitive function, the result is just `fn` preceded by a backquote.

The expression `_name<{&}`, either inside the definition of a function `fn` or entered when a defined function `fn` is suspended, will produce the fully qualified name of `fn`.

If in context `cx` you enter, literally, `_name<{f}`, you will get either the uninteresting result `` `cx.f `` or a value error: the only unqualified name you can use in the argument is one defined in the current context. A more useful case is when you have an expression that yields a function that may or may not be in the current context, an expression that uses `_gcb` or `of` or the like.

Additional Error Report

- A nonce error report is issued if `x` is a symbol.

Example

```
    $cx sys
    _name <{import}
`sys.import
```

Name Class `_nc{y;x}`

Arguments and Result

Both arguments and the result are symbol scalars; *y* can also be Null.

Definition

If the symbol *x* holds an unqualified user name, i.e., a name without an explicit reference to its context, and if the symbol *y* holds a context name, the result is the symbol form of the name class of the user name *x*, in the context *y* (see the "Name Classes" table). If *x* holds a fully qualified name of the form `cxt.nm`, the result refers to *nm* in the context `cxt`, and the value of *y* is ignored.

The root context is denoted by the empty symbol (`` ` ``). The current context is denoted by Null: if `_nc` is referenced in the form `_nc{;x}`, the current context is assumed.

Name Classes

Name Class	Description
`` `vars ``	User variables
`` `fns ``	Defined functions
`` `xfs ``	External functions
`` `globs ``	Global objects
`` `deps ``	Dependencies
`` `ops ``	Defined operators
`` `null ``	Not in use, not a valid name, or not one of the above

Name List `_nl{y;x}`

Arguments and Result

Both *y* and *x* are symbols, except that *x* can be ○ and *y* can be Null. The result is a vector of symbols.

Definition

The result is a vector of symbols holding the names of objects of class *x* (see the "Name Classes" table) in the context identified by *y*. The empty symbol (`` ` ``) denotes the root context, and Null denotes the current context. Dependencies are listed only when they have saved values, which need not be valid. See the table "Name List Expressions vs. The Kind of Objects in the Result" for examples of these and other uses of Name List. For the arguments `` `apl `` `` `ascii ``, and `` `uni ``, the order of the symbols is consistent between them (the symbols in the same positions representing the same primitive) within the same version of A+, but the order may change from version to version, and new functions may cause insertion of symbols; any such changes will not be considered upwardly incompatible.

Name List Expressions vs. The Kind of Objects in the Result

Expression (see Note below)	**Kind of Object Named in the Result**
``_nl{`ctx;`vars}``	Variables in the context `` `ctx ``
``_nl{`ctx;`fns}``	Functions in the context `` `ctx ``
``_nl{`ctx;`deps}``	Dependencies in the context `` `ctx ``
``_nl{`ctx;`ops}``	Operators in the context `` `ctx ``
``_nl{`ctx;`globs}``	Global objects in the context `` `ctx ``
``_nl{`ctx;`xfs}``	External functions in the context `` `ctx ``
``_nl{`;`cxs}``	Contexts
``_nl{`;`svs}``	System variables
``_nl{`;`sfs}``	System functions
``_nl{`;`cmds}``	System commands
``_nl{`;`apl}``	All keywords and primitive symbols in APL mode
``_nl{`;`ascii}``	All keywords and primitive symbols in ASCII mode
``_nl{`;`uni}``	All keywords and primitive symbols in UNI mode
``_nl{`;`circle}`` or `_nl{;○}`	List of symbolic left arguments for Circle function. (Although `` ` _nl `circle `` works fine, the parser is confused by `` ` _nl ○ ``. Left arg can be null, with correct result in any current context.)
``_nl{`;`errors}``	Names (as symbols) of errors defined in A+
``_nl{`;`nl}``	Valid right arguments to `_nl`
``_nl{`;`wa}``	Valid symbol arguments to `_wa`

Note:

- Where the left argument is shown as `` `ctx ``, it can also be Null (absent), indicating the current context.
- Where the left argument is shown as `` ` `` (the root context), it can also be Null (absent), indicating the current context.

Name Search _ns{y;x}

Arguments and Result

Both y and x are character arrays, normally vectors. The result is an integer vector or matrix.

Definition

If x is not a vector, its ravel is used. If y is a vector, the result is a vector of indices of nonoverlapping *namelike* occurrences of the vector x in y. Each index marks the beginning of an occurrence in y. An empty result signifies that there are no occurrences. A namelike occurrence is one that satisfies the following restrictions:

- not part of a comment;
- not in quotes;
- not immediately preceded by a backquote or a high minus (`` ` ``¯) or preceded or followed by an alphabetic character (in upper or lower case), a digit, or a dot or an underscore (._).

If y is a matrix or higher-rank array, then an occurrence is required to be entirely on one line; occurrences spanning rows are not accepted. (An unbalanced quotation mark, perhaps intended as an apostrophe, will block name recognition in later lines, however.) The result is an r by n matrix, where r is the rank of y and n is the number of occurrences found. A column of the result gives the indices of the corresponding occurrence, with the highest-order index at the top and the column index at the bottom. Thus 0# applied to the result gives a list of the items in which occurrences were found.

Examples

```
      _ns{'This is ''is''';'is'}
 5
     4 3ρ'abc abc  abc'
abc
 ab
c
abc
     _ns{4 3ρ'abc abc  abc';'abc'}
 0 3
 0 0
```

Name Search and Replace _nsr{y;x;r}

Arguments and Result

y, x, r, and the result are all character arrays, normally vectors.

Definition

If *x* or *r* is not a vector, its ravel is used. Occurrences of *x* are sought in exactly the same way as they are in Name Search, `_ns{y;x}`, but in two different objects, depending upon *y*, *x*, and *r*:

- If *y* is a vector or the ravels of *x* and *r* are the same length, then the result is *y* with *r* in place of every occurrence of *x* found in *y* itself.
- Otherwise, the result is the ravel of *y* with *r* in place of every occurrence of *x* found in the ravel of *y*; if no occurrences were found, the result is simply the ravel of *y*.

See also the examples in "String Search and Replace".

Example

```
     _nsr{'This is ''is''';'is';'are'}
This are 'is'
```

NaN Find `_nanfind{x}`

Argument and Result

The argument is any array. The result is an integer vector.

Definition

If the type of *x* is floating point (so *x* is simple), then the result is a vector consisting of the indices of those elements of the ravel of *x* that are NaN's. Otherwise the result is `⍳0`.

NaN's are used in some systems to represent indeterminate numbers. They are like `Inf`s, but more general. This function can be used to locate NaN's occurring in loaded files or resulting from dynamically loaded code. The `nan` subcommand of the `$dbg` command can be used to screen files being beamed in for NaN's. (NaN's are not generated by A+ primitive functions.)

Permissive Indexing `_index{i;x;d}`

Arguments and Result

i is a simple array of restricted whole numbers, *x* is any array, and *d* conforms to the items of *x*, i.e., its general type is the same as theirs, and either it is a scalar or its shape is the same as theirs.

Definition

This function is like `i#x`, with three exceptions: (1) Only a simple *i* is allowed, so only items of *x* are chosen. (2) If *i* is Null, the result is no items of *x*, not all items of *x*. (3) If an element of *i* is out of range, no index error is reported; instead, *d* is placed in the corresponding position in the result, replicated if necessary so that it matches the items of *x*.

Examples

```
     _index{2 19 14 1;
         ('Mar 06';'Jul 18';'May 15';'Nov 26');<'n.a.'}
< May 15
< n.a.
< n.a.
< Jul 18

     _index{10 11;⍳11 4;¯1}
 40 41 42 43
 ¯1 ¯1 ¯1 ¯1
```

Remove Dependency Definition `_undef{name}`

Argument and Result

The argument `name` is described in "The Name Argument". The result is a boolean integer.

Definition

The function removes the dependency definition for `name`, if any, and leaves everything else, such as value and callback function, unchanged. After this function is executed, `name` is an ordinary, not a dependent, variable. It may not have a value, since Remove Dependency Definition does not cause an evaluation. You may sometimes want to force an evaluation before invoking this function. The result is 0 if a definition was removed, and 1 otherwise - `name` does not name a dependency. Cf. "Remove Dependency Definition" (`$undef`).

Scalar Comma Fix Input `_scfi{s}`

Argument and Result

The argument `s` is a character array, normally a vector, and the result is a numeric scalar.

Definition

This function is like Comma Fix Input (`_cfi`), except that: it handles only a single number (so there must be no internal blanks); it has a simple scalar result; it has no boolean part to its result, an invalidity being reported as an error instead; and it treats commas properly, requiring that they be internal to the integer portion of the number, properly placed every three digits leftward from the decimal point (or right end, if there is no decimal point), and that there be either no commas or the full set. I.e., if `s` is a valid argument for `_scfi`, `_scfi{s}` is exactly equivalent to `''ρ1⊃_cfi{s;1}`.

Additional Error Report

- A domain error is reported if the first item of `_cfi{s;1}` would be zero.

Scalar Fix Input `_sfi{s}`

Argument and Result

The argument `s` is a character array, normally a vector, and the result is a numeric scalar.

Definition

This function is like Fix Input (`_fi`), except that it handles only a single number, has a scalar result, and has no boolean part to its result, an invalidity being reported as an error instead. I.e., if `s` is a valid argument, `_sfi{s}` is exactly equivalent to `''ρ1⊃_fi{s;1}`.

Additional Error Report

- A domain error is reported if the first item of `_fi{s;1}` would be zero.

Screen Format `_sfmt{f;d}`

Arguments and Result

The left argument is a format specification, either one shown in the "out Attribute Format Samples" table, or one acceptable as a left argument for `_fmt` (Format), or ⍕ (Format). The right argument is a simple numeric scalar. The result is a character vector.

Definition

If the left argument is a symbol, then it and the result are as described in the "out Attribute Format Samples" table. If the left argument is character, then it and the result are as described for `_fmt` (Format). Otherwise, the left argument is numeric and the result is as described for ⍕ (Format).

Formats containing `y2` are restricted to the years 1950 through 2049, errors in dates return `0`'s except that an erroneous `` `y2 `` is set to "`**`", and the length of the result returned for a bad date is the same as for a good date.

Warning! If the right argument is numeric but nonscalar, only the first element is used.

Set Attribute `_set{y;x}`

Argument and Result

The argument *x* is a nested vector `(w;z)`, where *z* is any array. Both *w* and the argument *y* are described in "The Name Argument". The result is a scalar integer.

Definition

Sets the value for attribute *w* on global variable *y* to the value *z*. If *z* is Null the attribute is removed. The result is 1 if the value was not previously set or the attribute was removed or *y* does not exist, and 0 otherwise. The attribute *w*, which is commonly a name, can be whatever you want; it simply allows you to associate data with *y* in an easily retrievable way.

Set Callback `_scb{y;x}`

Arguments and Result

The argument *y* is described in "The Name Argument". The argument *x* is a two-element nested array. The result is Null.

Definition

The argument *x* is a pair of the form `(f;s)`, where *s* is any array and *f* is a function expression. Thus the first element of *x* is a function scalar and not the name of a function, so this assignment is unaffected by any later changes to the meaning of the name of the function *f*. Executing `_scb{y;x}` causes the "callback" function *f* to be called each time the value of the global variable (or, indeed, function) named in *y* is modified through certain assignments, *after* the modification has been made. The array *s* is known as the static data, and is passed as a parameter to *f* on every call that results from an assignment to the global variable named in *y*.

If *s* is omitted the static data is Null. If *f* is omitted then any callback previously established for the global variable named in *y* is removed.

The general form of the syntax of *f* is

`f{s;d;i;p;c;v}` where

s is the static data
d is the new value
i is an index, nested if for y and simple if for , y
p is a path
c is a symbol naming the context in which y is defined, and
v is `y

such that

```
(i#p⊃c%v)←d   ⍝ if i is nested
(i#,p⊃c%v)←d  ⍝ if i is simple
     or
(p⊃c%v)[,]←d  ⍝ if i is out of range for c%v before this assignment
```

describes the change that caused f to be called.

A callback function can have any number of arguments from to zero to six, but the meaning of the arguments is always as above, reading from the left. For example, if it has three arguments then the first is static data, the second is new data, and the third is an index. Any explicit result of a callback function is ignored by A+.

Dependencies are global variables, and callbacks can be set on them. However, such a callback function is called only when the dependent variable is explicitly specified using ordinary or selective assignment, and in the latter case only as described above. A callback does not occur when a dependency is marked for evaluation, or when an evaluation occurs (although a *preset* callback *is* called in this latter case). Furthermore, during a callback on a dependency, the dependency's value is not marked invalid by any change the function makes in a variable on which it depends. (Cf. "Set Preset Callback".)

Only one callback can be set on a global variable. For example, if `_scb{`a;(fn;)}` is executed, and then `_scb{`a;(gn;)}`, the first callback function fn is removed when the second one gn is established. Since it is the callback function itself, and not a name, that is given to _scb, if you redefine the function and want the new definition to apply to the callback, you must call _scb again to set it.

For examples and more detail, see "Callback Functions".

Set Client Data _scd{y;x}

Arguments and Result

The argument x is any array. The argument y is described in "The Name Argument". The result is an array.

Definition

This function is like _set (Set Attribute), except that there is no explicit, named attribute and the result is the previous client data, i.e., what the result of _gcd{y}, Get Client Data, would be if it were executed immediately before _scd{y}. This function has been superseded by Set Attribute and is retained only for compatibility.

Set Preset Callback _spcb{y;x}

Arguments and Result

The argument `y` is described in "The Name Argument". The argument `x` is a two-element nested array, as described for Set Callback. The result is Null.

Definition

The differences between callback functions established by `_spcb` and `_scb` are:

- preset callback functions on dependencies are called when either a dependent variable is assigned a value or a dependency is evaluated (after the evaluation of the dependency definition is completed), whereas callback functions are called only when the dependent variable is assigned a value;
- `_spcb{y;x}` causes the callback function specified in `x` to be called *just before* the value of the global variable (or, indeed, function) named in `y` is changed, but `_scb` is called *just after*;
- preset callback functions are used to validate new values for global variables and they therefore return meaningful results (namely, the validated data), whereas callback functions do not;
- for the Append form of Selective Assignment, the indices `i` are not valid for `_spcb`, since the appending has not yet been done, but they are valid for `_scb`.

Since preset callbacks are used for validating new values of global variables, the following rules should be followed in their definitions:

- if a new value is valid, set the result of the preset callback function to that value;
- if a new value is invalid, signal an error (see "Signal"); the value of the global variable will remain unchanged.

For examples and more detail, see "Callback Functions".

Set System Variable `_ssv{y;x}`

Arguments and Result

The argument `y` is a symbol scalar or a quoted string. The result is always Null.

Definition

This function sets the value of the system variable named in `y` to the value of the right argument `x`. A domain error is reported or a correction made (e.g., `100|x` for `` `pp ``) if the right argument is not a valid value of the system variable named in `y` or if that system variable is not allowed to be set. See "System Variables" for descriptions of the various system variables and their values and interdictions.

Additional Error Report

- A domain error is reported if `y` does not name a system variable that can be set or `x` is an impermissible setting for `y`. Although `$Df 2` is a no-op, ``_ssv{`Df;2}`` is a domain error.

Stack Information `_doErrorStack{}`

Returns the stack information for the last error that occurred under a Protected Execute while `` `doErrorStack `` was enabled (1).

This facility should enable developers to mail error information without having to exit the application.

String Search `_ss{y;x}`

Arguments and Result

Both `y` and `x` are character arrays, normally vectors. The result is an integer vector or matrix.

Definition

If `x` is not a vector, its ravel is used. If `y` is a vector, the result is a vector of indices of nonoverlapping occurrences of `x` in `y`. Each index marks the beginning of an occurrence.

If `y` is a matrix or higher-rank array, then an occurrence is required to be entirely on one line; occurrences spanning rows are not accepted. The result is an r by n matrix, where r is the rank of `y` and n is the number of occurrences found. A column of the result gives the indices of the corresponding occurrence, with the highest-order index at the top and the column index at the bottom. Applying `0#` to the result gives the indices of the items in which occurrences were found. See also the examples for Name Search.

Examples

```
     _ss{'Mississippi';'is'}
 1 4
     _ss{'Mississippi';'issi'}
 1     ⍝ The instances must be nonoverlapping.
```

String Search and Replace `_ssr{y;x;r}`

Arguments and Result

`y`, `x`, `r` and the result are all character arrays, normally vectors.

Definition

If `x` or `r` is not a vector, its ravel is used. Occurrences of `x` are sought in exactly the same way as they are in String Search, `_ss{y;x}`, but in two different objects, depending upon `y`, `x`, and `r`:

- If `y` is a vector or the ravels of `x` and `r` are the same length, then the result is `y` with `r` in place of every occurrence of `x` found in `y` itself.
- Otherwise, the result is the ravel of `y` with `r` in place of every occurrence of `x` found in the ravel of `y`; if no occurrences were found, the result is simply the ravel of `y`.

Examples

```
     _ssr{'Mississippi';'is';'IS'}
MISsISsippi
```

```
     3 6⍴'abc abc defgh abcd'
abc ab
c defg
h abcd

     _ssr{3 6⍴'abc abc defgh abcd';'abc';'xyz'}
xyz ab
c defg
h xyzd

     _ssr{3 6⍴'abc abc defgh abcd';'abc';'wxyz'}
wxyz wxyz defgh wxyzd
```

Valence `_valence{x}`

Argument and Result

The argument `x` is either the name of a defined or external function, as described in "The Name Argument", or a function expression or scalar holding a defined, external, or system function. The result is an integer scalar.

Definition

The result is the number of arguments taken by the function specified in the argument `x`.

Additional Error Reports

- A nonce error is reported if `x` is a function scalar that is a primitive function;
- a domain error is reported if `x` is a symbol naming a system function, but not if `x` is a function scalar that is a system function.

Work Area `_wa{x}`

Argument and Result

The argument `x` is a symbol or integer scalar.

Definition

The expression `_wa n`, for `n` greater than or equal to 1, increases the workspace size by approximately `n` megabytes; see the first table entry regarding IBM machines. Other values of the argument `x` will produce information about the amount of available storage in the workspace, as well as the amount and distribution of storage in use, and in one case cause the areas of unused storage to be coalesced. See the "`_wa` Arguments and Results" table.

Memory allocation for A+ objects is based on an increasing set of integers called *fragment sizes*. The storage area for a particular object will be a *fragment* of sufficiently large fragment size to hold the object, but perhaps with extra, unused room. (It always uses a fragment of smallest possible size. Fragments can be split off larger ones as needed.) As noted in "`_wa` Arguments and Results", `_wa` can be employed to find out how many fragments of each size are currently in use.

`_wa` Arguments and Results

Argument	Effect and Result

`n`, for `n≥1`	The effect is to increase the workspace size, `` `size ``, by `n` megabytes (a megabyte is 2^{20} bytes). The effect of giving a noninteger argument is undefined. The result is 0 if successful. On IBM machines, the workspace size cannot be increased when a mapped file exists for an A+ process, because A+ requires that the entire workspace be contiguous and AIX requires that all space be taken from one end of the address space; thus, the mapped file blocks enlargement of the workspace.
`` `fragsizes ``	The result is a vector of integers of fragment sizes, in words. This vector is fixed for every A+ session, but may vary with the release number and the machine on which you are running.
`` `fragcounts `` or `0`	The result is a vector of integers giving the number of free, or available, storage fragments of each fragment size. (Note that these numbers are computed after the argument to `_wa` is allocated.)
`` `coalesce `` or `¯1`	The effect is to combine contiguous free fragments into larger fragments. The result is a vector of the new fragment counts. If there is no writable `/var/atmp` (see the `-m` invocation flag, below) nothing is done and the result is all zeros.
`` `size ``	The result is the workspace size, i.e., its current maximum permitted size, in bytes. It can be increased by `_wa n`, or by a naked left arrow (←, indicating resumption) with a pending wsfull error.
`` `avail ``	The result is the amount of available storage in the workspace. (Note that the amount is computed after the argument to `_wa` is allocated.) It is the difference between the workspace size, `` `size ``, and the space actually being used, and is almost equal to the result of the command `$wa`. Usually, some of `` `avail `` is in `` `atmp `` and some is not.
`` `atmp ``	The result is the current total size of the files used to hold the workspace. It includes both space that is in use and space that is available. See `` `avail ``. `$df /var/atmp` tells more about available atmp.
`` `info ``	The result is a combination of the above information in the following arrangement: ``(`size `atmp `avail; `fragcounts; `fragsizes)``.

A+ data objects (including function arrays but not functions) are stored as 56-byte headers and bodies of varying size. If the number of elements of an object is `n`, then its body will take, in bytes, depending on its type:

`` `char ``	`n+1`
`` `int ``	`4×n⌈1`
`` `float ``	`8×n⌈1`
`` `sym ``	`4×n⌈1`
`` `box ``	`4×n⌈1` plus the storage for each of its elements
`` `func ``	`4×n⌈1`
`` `null ``	usually no space, but sometimes the same formula as `` `box ``.

A+ looks for its atmp space in `/var/atmp/*`. If several A+ processes are running on the same machine, they will compete for the same atmp space. The atmp space is used as both workspace area and mapped file area, so if large mapped files are used it may even be necessary to *reduce* the workspace size to get an application to work. Usually, it is best to keep the workspace about twice as large as the amount of memory you expect to use; 256Meg is the recommended safe limit, beyond which you may run out of address space.

Modifying Memory Mapping Characteristics

The memory mapping characteristics of atmp can be modified by means of the invocation flag `-m` with the value `ws_atmp_shared`, `ws_atmp_noreserve`, `ws_atmp_private`, `ws_atmp_heap`, or `ws_malloc`. The same settings can be given to `APLUS_ATMP_MODEL`, an environment variable, with the same effect.

`ws_atmp_shared`, `ws_atmp_noreserve`, and `ws_atmp_private` control the flags used for mmap and atmp (see the man page for mmap); `ws_atmp_noreserve` is implemented as `MAP_AUTORESRV` on sgi. `ws_atmp_heap` will put atmp in memory. `ws_malloc` will just use `malloc` and `free` as needed. If there is no `/var/atmp` then the default will be `ws_malloc`. When `/var/atmp` does exist, the default is `ws_atmp_shared` except on UltraSparc and all Solaris machines, which have a default of `ws_atmp_noreserve`. Alternatively and temporarily, the memory mapping characteristics of atmp can be modified by setting the environment variable `APLUS_ATMP`. This variable is intended for use by system administrators to configure new architectures. The possible settings for `APLUS_ATMP` are: `MAP_PRIVATE`, `MAP_NORESERVE`, and `FROM_HEAP`.

The Heap

There is another area of storage in A+, aside from atmp space, namely, the *heap*. The heap is used for A+ internal memory needs, either temporary or permanent, such as in creating contexts or global names, parsing nested arrays, and buffering outgoing messages in adap. Subsequent uses of a symbol or name within a process do not use more memory, because expunging a name does not free the heap space used for it.

A long-running A+ process can eventually use up its heap, although the heap is pretty large and A+'s demands on it are fairly small. A `brealloc()` error can then occur. The obvious way of controlling this potential problem is to not use a tremendous number of names. The heap size can be changed at A+ invocation by using the `-h` flag; caution: do not use this flag unless it is specifically needed to solve a memory-related problem. The size can be measured using `pmon`, remotely, from an XTerm, but not from within an A+ process, since `pmon` is interactive, so `$pmon` cannot be used.

Another way to monitor heap space use is to run in an XTerm, while A+ is running,
```
while [ true ]; do pstat -s; sleep 1; done
```

Handling of Symbols

Memory allocation and hashing algorithms are intended to provide high performance even for applications using 10,000 or more symbols. Symbols are kept in the atmp area rather than in the heap, preventing the bus errors that could occur when an application used hundreds of thousands of symbols, exhausting the heap. Such an application may need a relatively large `_wa` allocation.

The address space for an A+ process is fixed at two gigabytes. Four things use address space:

- the actual code of the A+ process, and dynamically loaded routines;
- dynamic memory (heap space);
- atmp space;
- mapped files.

See also "Memory Allocation - What to Do and What Not to Do" and "Memory Allocation in A+ - a Closer Look" in "Calling C Subroutines from A+".

13. System Commands

All system commands begin with a dollar sign (`$`) as the first nonblank character on the line. They cannot appear directly in defined functions or expression groups, but can appear in character vectors executed by Execute, but not by Value, with `$` the first nonblank character. The result of Execute for a command is Null. The result is not the text that appears as a result of executing the system command directly in an A+ session.

If A+ recognizes a command as clearly erroneous, it gives *`incorrect`* as a response. It simply ignores (correct) A+ commands that have missing or wrong arguments, like `$rl` or `$rl 'a'`.

Any command that is not recognized by A+ as either one of those described below or clearly erroneous is assumed to be a Unix command and is executed in a child process forked from the A+ process, and responded to (including error responses) by that process. A+ invokes `sh` by default, not `ksh`, `csh`, or whatever; in particular, this means that `~` in path names is not interpreted as a "magic" character. If you need this or some other property of one of these shells, invoke that shell explicitly in the command, as in
`$ksh -c "ls ~user/dirnm/*.+"`.
(The quotes in the example are necessary: `ksh -c` takes only one argument.)

Warning: The result of a Unix command depends upon the particular system under which it is executed - `$unixcommand` is ***highly nonportable***. If you need advice about the use of such a command, you should usually ask someone who knows a lot about Unix, and not necessarily someone who knows a lot about A+.

Classification of System Commands

Although they are listed alphabetically by English name in this chapter, for convenient reference, the A+ system commands can be grouped, among many other ways, in seven categories, dealing with:

- names and references: Commands, Context, Contexts, Expunge, Expunge Context, External Functions, Functions, Global Objects, Operators, System Functions, Variables;
- dependencies: Dependencies, Dependency Definition, Dependent Object Names, Remove Dependency Definition;
- implicit arguments of primitive functions: Printing Precision, Random Link;
- files and Unix: Change Directory, Load, Load and Remove, Pipe, Pipe In, Pipe Out, Pipe Out Append;
- versions: Mode, Version;

- error handling and debugging: Debugging State, Dependency Flag, Execution Suspension Flag, Protected Execute Flag, Reset, Set Callback Flag, Stack Information, State Indicator, Stop, X Events Flag;
- A+ operation and space: Off, Terminal Flag, Workspace Available.

Definitions of System Commands

Callback Flag `$Sf [n]`

This flag is a debugging aid that enables (1) or disables (0) callback functions. Normally, with callbacks enabled (`$Sf 1`), a callback function is invoked when an associated variable changes. With callbacks disabled (`$Sf 0`), no callback function is invoked. The default setting is 1. `$Sf` alone displays the current setting. This flag can be used to inhibit callbacks while clearing suspensions: set it to 0, clear, set it back to 1, and continue execution. Note that the screen management system uses callback functions, as may any A+ utility, and therefore setting this flag to 0 can have unpredictable effects. Caution is advised when using `$Sf`.

Change Directory `$cd` **[*path*]**

If *path* is present, it must be a valid Unix directory path, and the effect of the command is to change the current directory to the one named in *path*. If *path* is not present, the user's home directory becomes the current directory. After a file is loaded, the current directory is the same as it was when the Load function or command was issued: it is automatically restored if it was changed during execution of the lines of the file. The PWD environment variable is set. See "Change Directory" (`_cd`).

Commands `$cmds`

Displays a list of the system command names. See "Name List" (`_nl`).

Context `$cx` **[*name*]**

If *name* is present, it must be a valid context name (see "User Names"), and the effect of the command is to make that context the current context. If the context does not exist it will be created. The root context is specified by a dot (`.`). If *name* is not present, the name of the current context is displayed. If the context is changed when a function is suspended, resumption of execution will automatically change the context back. After a file is loaded, the current context is the same as it was when the Load function or command was initiated: it is automatically restored if it was changed during execution of the lines of the file.

Contexts `$cxs`

Displays a list of all the context names in the active workspace. See "Name List" (`_nl`) and "Expunge Context" (`$excxt`).

Debugging State `$dbg [subcommand list]`

Several subcommands (with any parameters for them) can be given as arguments to `$dbg`, separated by blanks, as in

```
$dbg +load +beam +dyld +pack
```

and

```
$dbg indent 4 char >.
```

If no argument is given, the current settings are displayed. A listing of the various arguments together with their meanings is displayed in response to the subcommand `help`.

There are some additional messages. For example, when load tracking is on and a load command cites a nonexistent file, a debugging message mentioning the failed load command is issued.

The two subcommands `char x` and `indent n` are used to control the formatting of debugging messages. For example, if *x* is `|` and *n* is 2, a display when tracing is on might look like this:

```
⍝     .f entered
⍝     | .g entered
⍝     | | .h entered
⍝     | | .h exited
⍝     | .g exited
⍝     .f exited
```

whereas if *x* were `>` and *n* were 4, the third line, for instance, would look like this:

```
⍝     >   >   .h entered
```

The table below shows all the subcommands. The arguments that are not entered literally as shown are *b*, which is 0 or 1 (if *b*, `+`, and `-` are omitted, the command acts as a toggle); *c*, which is a context name or names; *x*, which is a character; and *n*, which is a nonnegative integer. You need enter only initial segments of the literal parts of the arguments just long enough to make them unambiguous.

`$dbg` Subcommands (Arguments)

Subcommand	Significance
(none)	Display all current settings.
`1` or `0`	Turn all tracing and tracking on (`1`) or off (`0`).

`beam b`
`+beam`
`-beam`
	Turn beam tracking (mapping and unmapping) on (`+beam` or *b*=`1`) or off (`-beam` or *b*=`0`), or toggle (just `beam`).

`bitwise b`
`+bitwise`
`-bitwise`
	Turn checking for nonboolean And and Or arguments on (`+bitwise` or *b*=`1`) or off (`-bitwise` or *b*=`0`), or toggle (just `bitwise`). The default is to check.

`char x`	Set character to be used in dbg messages (see example above).

`+cxt c`
`-cxt c`
	Limit tracing to the contexts listed in *c* (`+`) or all but the contexts listed in *c* (`-`). If *c* is absent, trace all contexts (whether `+` or `-`). Message indentation always corresponds to execution depth, regardless of omissions.

`def b`
`+def`
	Turn tracking of function and dependency definition on (`+def` or *b*=`1`) or off (`-def` or *b*=`0`), or toggle (just `def`).

`-def`	

`dep b`
`+dep`
`-dep`
	Turn dependency evaluation tracing on (`+dep` or `b=1`) or off (`-dep` or `b=0`), or toggle state (just `dep`).

`disp map`	Display the limit on the number of concurrently mapped files; a list of files currently mapped, with reference counts; and some internal index information.
`display beam`	Display all current mappings of files.
`display flags`	Display the value of several key system variables; highlight nondefault values.

`do b`
`+do`
`-do`
	Turn Monadic-Do tracing on (`+do` or `b=1`) or off (`-do` or `b=0`), or toggle state (just `do`).

`dyld b`
`+dyld`
`-dyld`
	Turn dynamic load tracking on (`+dyld` or `b=1`) or off (`-dyld` or `b=0`), or toggle state (just `dyld`).

`fmt b`
`+fmt`
`-fmt`
	Turn messages from `_fmt` on (`+fmt` or `b=1`) or off (`-fmt` or `b=0`), or toggle state (just `fmt`). The default is no message display.

`func b`
`+func`
`-func`
	Turn defined function and operator tracing on (`+func` or `b=1`) or off (`-func` or `b=0`), or toggle state (just `func`).

`help`	Display all permissible settings and their significance.
`indent n`	Set dbg message indentation per level of execution depth.

`inv b`
`+inv`
`-inv`
	Turn on (`+inv` or `b=1`) or off (`-inv` or `b=0`) or toggle (just `inv`) the tracing of dependency invalidations and the triggering of a callback (if a callback function is defined, using `_dbg`). Messages are indented to make tracing of dependency trees easier. A warning is issued when `inv` is on and a cyclic dependency is encountered during invalidation.

`levels n`	Maximum (execution) depth of tracing. `0`: off. No `n`: unlimited.

`load b`
`+load`
`-load`
	Turn load tracking on (`+load` or `b=1`) - showing full path names - or off (`-load` or `b=0`), or toggle state (just `load`).

`nan b`
`+nan`
`-nan`
	Turn on (`+nan` or `b=1`) or off (`-nan` or `b=0`) or toggle (just `nan`) the checking of files as they are beamed in for NaN and Inf. If checking is on and such an undesirable is found in a file being beamed in, an error message is sent to stdout. Cf. `_nanfind`.

`pack b`	Turn packfile command tracing on (`+pcb` or `b=1`) or off (`-pcb` or

`+pack` `-pack`	*b*=0), or toggle state (just `pcb`).

`pcb b`
`+pcb`
`-pcb`
	Turn preset-callback tracing on (`+pcb` or *b*=1) or off (`-pcb` or *b*=0), or toggle state (just `pcb`).

`_prcb b`
`+_prcb`
`-_prcb`
	Turn prereference-callback tracing on (`+_prcb` or *b*=1) or off (`-_prcb` or *b*=0), or toggle state (just `_prcb`).

`print b`
`+print`
`-print`
	Display tracing messages (`+print` or *b*=1) or not (`-print` or *b*=0), or toggle (just `print`). The default is to display them. Designed to be used in conjunction with callbacks, which are set using `_dbg{`cb;(fn;cd)}`.

`_rcb b`
`+_rcb`
`-_rcb`
	Turn reference-callback tracing on (`+_rcb` or *b*=1) or off (`-_rcb` or *b*=0), or toggle state (just `_rcb`).

`scb b`
`+scb`
`-scb`
	Turn set-callback tracing on (`+scb` or *b*=1) or off (`-scb` or *b*=0), or toggle state (just `scb`).

`sfs b`
`+sfs`
`-sfs`
	Turn system function tracing on (`+sfs` or *b*=1) or off (`-sfs` or *b*=0), or toggle state (just `sfs`).

`tkerr b`
`+tkerr`
`-tkerr`
	Turn reporting of otherwise silent execution errors that do not cause execution suspension on (`+tkerr` or *b*=1) or off (`-tkerr` or *b*=0), or toggle state (just `tkerr`). ("tk" is for toolkit.) The default is 0.

`wa b`	Turn tracing of `$wa` on or off, or toggle. Will also trace `_wa` and workspace size changes by ← soon.

`xeq b`
`+xeq`
`-xeq`
	Turn Execute (Protected or not) tracing on (`+xeq` or *b*=1) or off (`-xeq` or *b*=0), or toggle state (just `xeq`).

`xfs b`
`+xfs`
`-xfs`
	Turn external function tracing on (`+xfs` or *b*=1) or off (`-xfs` or *b*=0), or toggle state (just `xfs`).

Examples:

```
     2 1 2 ∧ 1 1 0    ⍝ $dbg +bitwise is the default.
⍝    Warning: found 2 nonbooleans in left argument of ∧.
 1 1 0

     2 1 2 ∧ 7 1 8
⍝    Warning: found 4 nonbooleans in both arguments of ∧.
 1 1 1

    f{}:c.g{}
    c.g{}:'c.g'
    $dbg func 1      ⍝ Trace function calls.
```

```
     f{}
⍝    .f entered
⍝      c.g entered
⍝      c.g exited
⍝    .f exited
c.g
     $dbg levels 1    ⍝ Limit the levels traced to 1.
     f{}
⍝    .f entered
⍝    .f exited
c.g
```

```
     $dbg levels      ⍝ Trace all levels.
     $dbg +cxt c      ⍝ Limit tracing to the c context.
     f{}
⍝      c.g entered    ⍝ Note the indentation, which indicates the level.
⍝      c.g exited
c.g

     $dbg 0    ⍝ Reset debugging; default for fmt is 1: issue _fmt msgs.
     do line←'i3,7c1'_fmt(n←185;1 7⍴' spread')
⍝ _fmt: Missing edit specification
⍝ _fmt: i3,7c1
⍝ _fmt:      ^^
<   9
< domain
     $dbg -fmt        ⍝ Turn off messages from _fmt.
     do line←'i3,7c1'_fmt(n;1 7⍴' spread')
<   9
< domain
```

Dependencies `$deps [name]`

If `name` is present it must be a valid context name (see "User Names"), and the effect of the command is to display a list of dependency names in that context. If `name` is not present the names of dependencies in the current context are displayed. See "Name List" with `` `deps `` right argument.

Dependency Definition `$def name`

This command displays the definition of the dependency named in `name`. See "Dependency Definition" (`_def`), and "Dependencies".

Dependency Flag `$Df` [*n*]

This command enables (1) or disables (0) dependencies, by setting `` `Df ``. Normally, with dependencies enabled (`$Df 1`), dependency definitions are evaluated when dependencies are referenced, and their definitions are marked for evaluation. With dependencies disabled (`$Df 0`) dependency definitions are never evaluated, which has the effect that dependencies act like normal global variables. However, they are still marked for evaluation and will be evaluated if so marked when referenced after `$Df` is reset to 1. Display mode (`$Df 2`) no longer exists, its function having been incorporated in `$dbg`; the argument 2 is a no-op. If no parameter is given to `$Df` its current value is displayed. The default setting is 1.

Warning! Disabling dependencies can cause confusion. Be careful.

Dependent Object Names `$dep name`

This command displays the names of all dependencies whose definitions explicitly reference `name`. See "Dependent Object Names" (`_dep`).

Execution Suspension Flag `$Ef [n]`

Execution suspensions due to A+ errors occur normally when this flag is set to its default value 1. When it is set to 0, these suspensions do not occur: an A+ error causes abandonment of execution instead, without any message. If no argument is given to `$Ef`, its current value is displayed. Note that errors include signals generated by the primitive function Signal. See also the in attribute. This definition is experimental and may be refined in the future, or replaced by a different mechanism.

Expunge `$ex names`

This command removes the global objects whose names are listed in `names` from the active workspace. These objects can be variables, functions, dependencies, and operators. Global variables and dependencies bound to display classes will not be removed (see the functions `is` and `free`); in these cases the command will display the message `name: is bound.` See "Expunge" (`_ex`).

Expunge Context `$excxt context`

This command removes the context whose name is listed in `context` from the active workspace. The context must be empty, else the command displays the message `context: not empty.` The Global Objects command `$globs name` can be used to list all objects that would prevent the expunging of a context. See "Expunge Context" (`_excxt`).

The root context cannot be expunged. Any attempt to expunge it or a nonexistent context is ignored.

External Functions `$xfs [name]`

The command lists external compiled C program names in the current context if `name` is not present, else in the context `name`.

Functions `$fns [name]`

If `name` is present, it must be a valid context name (see "User Names"), and the effect of the command is to display a list of the user function names in that context. If `name` is not present, the names of functions in the current context are displayed. See "Name List" (`_nl`).

Global Objects `$globs [context]`

This command lists the names of all global objects of any kind that appear in the given context. The list of names includes names for which attributes have been set (using `_set`), even if they have not been given values and consequently are not listed by `$vars`. If `context` is omitted, the current context is assumed. This command is especially useful in connection with the Expunge Context command, `$excxt`. See "Name List" (`_nl`).

Input Mode `$mode [apl | ascii | uni]`

If `apl` or `ascii` or `uni` is present, the keyboard mode is set to the one specified. Otherwise, the current mode is displayed. APL mode means that the special graphic characters that denote A+ primitive functions are to be entered, using the APL union keyboard. ASCII mode means that instead of each of these special graphics a specified sequence of one or more standard ASCII characters is to be entered; see the table below. There is also a mode like ASCII but more consistent and presumably easier to read and remember, namely UNI; see the second table below (which, unlike the first table, includes characters that are the same as for APL mode). It is important to understand that the latter two modes involve only simple token substitution.

Session output is in the specified mode, but expressions being executed, in functions or directly from the session input, are never translated or otherwise reconstructed. Derived functions, however, are re-created when displayed. `s.box`, incidentally, seems to live at some sort of halfway house.

UNI tokens are:

1. ASCII characters;
2. sequences of two ASCII characters in which the second character is =, /, or \ (used because such sequences are not otherwise legal A+ expressions); and
3. strings consisting of an uppercase letter, a period, and a token as in 1. or 2. (again, not otherwise an A+ expression). The uppercase letters are used to group the symbols:
    - M for mathematical;
    - I for indexing;
    - S for shape;
    - O for outer products, but Q where an M.y is involved, resulting in Q.y;
    - P for inner product;
    - B for bitwise;
    - A for argument;
    - F for file;
    - E for evaluate and inverse evaluate (Format); and
    - Y for symbols.

Examples:

```
      $mode apl
      q←×¨
      f{}:0÷0
      3÷4
 0.75
      3⌈4
 4
      q
(×¨)

      $mode uni
      q
(*~)
      3%4
 0.75
      3 M.+ 4
 4
      -3 4
 -3 4
      1 + -3 4
 -2 5
      - 3 4
 -3 -4
      1 + - 3 4
 -2 -3
      g{}:0%0
      g{}
//[error]  %: domain
*      $si
g{}
.g: 0%0
```

```
*     →
     f{}
//[error]  %: domain
*     $si
f{}
.f: 0÷0

*     →
     s.box -3 4
-3 4
     s.box - 3 4
-3 -4
     s.box 2 3 4
2+⍳3
     $mode apl
     s.box ¯3 4
¯3 4
     s.box -3 4
¯3 ¯4
     s.box 2 3 4
2+⍳3
     g{}
⍝[error]  ÷: domain
*     $si
g{}
.g: 0%0
```

ASCII Mode Equivalents To APL Graphics

APL	ASCII	APL	ASCII	APL	ASCII	APL	ASCII
←	:=	¨	each	∧	&	∨	?
×	*	÷	%	⍟	log	≡	==
⌈	max	∘.⌈	max. see below	⌊	min	∊	in
≠	~=	≤	<=	≥	>=	⋆	^
⌽	rot	⍉	flip	⍳	iota	⍴	rho
↑	take	↓	drop	⍋	upg	⍒	dng
⊥	pack	⊤	unpack	⊂	bag	⊃	pick
⍎	eval	⍕	form	⊢	rtack	⊣	where
○	pi		mdiv	?	rand	⌶	beam
%	ref	∪	dot	⍝	//	~⍨	bwnot
∧⍨	bwand	∨⍨	bwor	<⍨	bwlt	≤⍨	bwle
=⍨	bweq	≥⍨	bwge	>⍨	bwgt	≠⍨	bwne

Note to table: In ASCII mode, every valid outer product `∘.f` is replaced by `s.` (read "s-dot"), where *s* is the string of symbols that replaces *f*. For example, `>=.` replaces `∘.≥`.

UNI Mode Equivalents To APL Graphics

APL	UNI	APL	UNI	APL	UNI	APL	UNI
+	+	− (minus)	- (and perhaps space)	¯ (high minus)	-	×	*
÷	%	⋆	M.*	⍟	M.&	∣	M.|
	M.#	⌈	M.+	⌊	M.-	⊥	M.<
⊤	M.>	?	M.?	○	M.^	∧	&
∨	|	∼	!	<	<	≤	<=
=	=	≥	>=	>	>	≠	!=
≡	==	⍕	E.%	⍎	E.*	%	^
∪	Y.&	/	/	\	\	⊣	A.<
⊢	A.>	←	:=	⌶	F.!	#	#
⊃	I.>	⍳	I.#	⍋	I.+	⍒	I.-
∊	I.?	⊂	I.<	↑	S.+	↓	S.-
,	,	⍴	S.?	⌽	S.|	⍉	S.\
!	S.!	+/	+/	×/	*/	⌈/	M.+/
⌊/	M.-/	∧/	&/	∨/	|/	+\	+\
×\	*\	⌈\	M.+\	⌊\	M.-\	∧\	&\
∨\	|\	∘.+	O.+	∘.−	O.-	∘.×	O.*
∘.÷	O.%	∘.⋆	Q.*	∘.∣	Q.|	∘.⌈	Q.+
∘.⌊	Q.-	∘.<	O.<	∘.≤	O.<=	∘.=	O.=
∘.≥	O.>=	∘.>	O.>	∘.≠	O.!=	+.×	P.*
⌈.+	P.+	⌊.+	P.-	∧⍨	B.&	∨⍨	B.|
∼⍨	B.!	<⍨	B.<	≤⍨	B.<=	=⍨	B.=
≥⍨	B.>=	>⍨	B.>	≠⍨	B.!=	@	@
¨	~	*time*	time	*case*	case	*do*	do
else	else	*if*	if	*while*	while	⍝	//
→	+=	*e*	e (e-notation)	.	.	$	$
&	? (stack vars)	`	` (back-quote)	'	'	"	"
:	:	;	;	(	(	)	)
[	[	]	]	{	{	}	}

Note to table: A UNI minus represents an APL high minus if that is syntactically possible. To force it to be an APL minus, use a blank after a UNI minus when necessary, as shown in the examples.

Load `$load script`

The argument `script` is a filename. In response to this command, the directories specified in the environment variable APATH are searched for the file named `script`. If that file is not found and `script` does not end with `.+` or `.a`, these directories are searched again for `script.+`, and if that file is not found they are searched for `script.a`. Once a file is found, it is loaded into the active workspace by interpreting every line of the file, starting at the top, essentially as if the lines had been entered directly in the active workspace. The difference is that after a file is loaded, the current context and the current directory are the same as they were when the Load command was issued: each is automatically restored if it was changed during execution of the lines of the file. The PWD environment variable is set and restored during this operation.

Load and Remove `$loadrm script`

This command is the same as the Load command, except that the file is deleted after it has been loaded.

Mapped Files Limit `$maplim` [`n`]

The value of `$maplim` is the number of files that can be mapped concurrently. `$maplim n` sets the limit to `n`; the default is 2000. Reducing the limit saves space, since it reduces the size of an internal table.

Off `$off`

The effect of this command is to terminate the active A+ process. See "Exit" (`_exit`).

Operators `$ops` [`name`]

If `name` is present, it must be a valid context name (see "User Names"), and the effect of the command is to display a list of the user operator names in that context. If `name` is not present, the names of operators in the current context are displayed. See "Name List".

Pipe `$|var cmd`

The effect of this command is to pipe the default display of the global variable named in `var` to the Unix command named in `cmd`. The argument `var` can be a fully qualified name, specifying a context. This is actually a Unix command: see a Unix manual for details. Example:

```
    txt←'Allow for 3-day settlement.'
    $|txt cat >>note.asc
```

Pipe In `$<var cmd`

The `$<`*var cmd* command runs the Unix command *cmd* and puts the result (the standard output) in the global variable *var* as a character vector. There must be no space between the `$` and `<` symbols and *cmd* must not be an interactive command.

To read ASCII file `myfile` into variable `myvar`, you can use `$<var cat myfile`.

Pipe Out `$>obj filename`

The `$>`*obj nm.asc* command puts an ASCII version of the A+ global object *obj* in the file *nm.asc*, replacing the previous contents if the file already exists. It is equivalent to displaying *obj* and cutting and pasting it into *nm.asc* after emptying *nm.asc* if necessary, and basically equivalent to `$|`*obj* `cat >`*nm.asc*. If *obj* is a function, its definition is written in the file. There must be no space between the `$` and `>` symbols.

Pipe Out Append `$>>obj filename`

The `$>>`*obj nm.asc* command is just like `$>`*obj nm.asc* except that if the file *nm.asc* exists the ASCII text version of the default display of the A+ object is *appended* to it, instead of replacing the previous contents. There must be no space within the `$>>` sequence.

Printing Precision `$pp` [*d*]

If *d* is present, then, in the usual case, the effect is, loosely speaking, to set the Printing Precision to *d*. More precisely, the relevant characters of *d* are the first nonblank character and the character immediately following it, if that character exists and is nonblank. If the relevant characters are digits, the effect of the command is to set the precision to the number they represent.

If *d* is omitted (or a relevant character is not a digit), the current setting is displayed.

The Printing Precision `` `pp `` normally specifies the maximum total number of digits to be used in the display of a number, not counting the two digits following the `e` in exponential notation. There are two exceptions:

- If `` `pp `` is zero, then it is treated as if it were one.
- If `` `pp `` is less than ten, integers are nevertheless displayed with up to ten digits.

The maximum meaningful value for $pp is 16. Digits displayed on output beyond the 16th digit are not valid.

See "Printing Precision" (`` `pp ``), the examples for Format (⍕), and the examples for Default Format.

Protected Execute Flag `$Gf [n]`

This flag is a debugging aid that enables (1) or disables (0) protected execution. When disabled, a Protected Execute or a Monadic Do produces the same error message as would the A+ expression it executes. The default setting is 1. See "Execute in Context or Protected Execute" (⍎) and "Do - Monadic (Protected Execution)".

Examples

```
     $Gf 0
     do{⍎234}
 ⍎: type
*      →
     $Gf 1
     do{⍎234}
<   6
<  type
```

Random Link `$rl n`

The random link is set to positive integer `n`. The random link is used as a *seed* by the primitive functions Roll and Deal. Setting the random link allows the user to produce repeatable sequences of pseudorandom numbers. See "Random Link" (`` `rl ``).

Remove Dependency Definition `$undef name`

This command removes the dependency definition for `name`, if any. It leaves everything else, such as value and callback function, unchanged. After this command is executed, `name` is an ordinary, not a dependent, variable (if it was previously a variable). It may not have a value, since Remove Dependency Definition does not cause an evaluation. You may sometimes want to force an evaluation before invoking this command. Cf. "Remove Dependency Definition" (`_undef`).

Reset `$reset [n]`

The `$reset n` command clears *n* suspensions, and `$reset` alone on a line clears all suspensions. A negative or otherwise invalid argument is taken as `0`, a no-op, and all arguments equal to or greater than the number of suspensions are equivalent. `$reset` does not clear a multiline entry with unbalanced punctuation and, indeed, is blocked by it; to abort the expression, use either → or `$` alone.

In terms of asterisks, → (or `$`) always removes one asterisk and may remove more than one if the innermost asterisks signal unbalanced punctuation, whereas `$reset` alone on a line removes either all or none (when the innermost asterisks signal unbalanced punctuation) and `$reset n` removes either *n* or none.

Stack Information `$doErrorStack`

Set the system variable `` `doErrorStack `` to 0 or 1. See `_doErrorStack{}`.

State Indicator `$si`

The name and current line of the currently suspended function are displayed. Any functions pendent, that is, waiting for the currently suspended function to be completed, are displayed on lines above the display line of the suspended function. Any uncleared previously suspended functions and their pendent functions are not shown. Lines displayed in the state indicator are:

- Functions or operators, e.g., `.sqrt: x*0.5`
- Immediate execution line, e.g., `x*0.5`
  or `sqrt ¯1`
- File name and failed line number: `statfns.+[23]  x*0.5`

The user is reminded of the suspension by the appearance of a `*` in the prompt. To clear the most recent suspension enter `$` or right arrow (→) and press the Enter key. To clear all suspensions, clear each individual suspension in turn. `$reset [n]` allows you to clear as many as you want with one command.

Stop `$stop [n]`

If `n` is present, the effect of the command is to set the *stop debugging state* to `n`. The valid settings for `n` are 0 (none), 1 (stop), and 2 (display). See "Stop" (`^`) for the meaning of the three states. If `n` is not present, the current value of the stop debugging state is displayed.

Warning: If you are running A+ from Emacs and you set this state to 1, then if the Stop function (^) is executed while a menu's contents are displayed but before a selection is made, the machine is hosed until the Emacs session is killed.

System Functions `$sfs`

This command displays a list of the system function names. See "Name List" (`_nl`). (`$_sfs` lists system functions which are for the use of A+ implementers only.)

Terminal Flag `$Tf`

Turns terminal input (stdin) off. When programming an A+ application to be run from a Unix shell script, put `$Tf` at the end of the application file; otherwise A+ will be in open keyboard mode after the application file has been read, and the Unix shell script will not be executed past the point where A+ was invoked. This mechanism will be replaced in the future. See "Standard Input": `` `stdin`` when 0 has the effect of `$Tf` but queues keyboard entries for processing when it is subsequently reset to 1.

Variables `$vars` **[***`name`***]**

If `name` is present, it must be a valid context name (see "User Names"), and the effect of the command is to display a list of the variable names in that context. If `name` is not present, the names of variables in the current context are displayed. Dependencies are listed only if they have saved values, which do not need to be valid. Names that have been given attributes using `_set` but that have not been given values are not listed; see "Global Objects" (`$globs`). Also see "Name List" (`_nl`).

Version `$vers`

The version of the active A+ system is displayed. See "Version" (`` `vers``).

Workspace Available `$wa` **[***`n`***]**

If `n` is a positive integer, `$wa n` adds `n` megabytes to the active workspace.[1] `$wa 0` is almost equal to `` (_wa `fragcounts),(_wa `size),(_wa `atmp),_wa `avail``. `$wa` (with no argument) is almost equal to `` _wa `avail``.

See "Work Area" (`_wa`). (The slight differences are accounted for by the system requirements for running the command and the function.)

X Events Flag `$Xf [n]`

Enable (1) or disable (0) X events. When X events are enabled (`$Xf 1`), X events that do not have A+ callbacks will be processed even when A+ programs are running. This gives the appearance of a multithreaded environment. For example, scrolling a window will work even while a long-executing program is running, as long as there is no A+ callback attached to the scrolling event. This setting is the default.

If X events are disabled (`$Xf 0`), then no X events will be processed when an A+ program is running. Instead, they are queued and dispatched after the program execution is completed.

This command has no use in running applications, but the 0 setting may be useful in debug mode.

14. Defined Functions and Operators

Programs (user-defined functions and operators) are defined when they are entered in an A+ session. They may be entered at the keyboard, effected when a character vector containing a definition is executed, brought in from a script during the execution of a `$load` command, or, in an Emacs A+ session, brought in from any Emacs buffer, including the A+ one. In any case, the same rules apply regarding the form in which programs must be entered.

In Emacs, to bring in a program all at once, place the cursor anywhere in it and press the **F3** key. To bring it in a line at a time, or just to retrieve individual lines, place the cursor on each line successively and press the **F2** key. After either of these keys has been pressed, if the source buffer is not the A+ buffer, there will be two buffers in the window, A+ and source. Caution: The **F3** key causes the program to be read exactly as it appears in the file, so if there are asterisks indicating continuation of the definition (as in a multiline definition entered in an A+ session), these asterisks will either cause a parse error or result in a presumably unexpected new definition.

(Cf. "The Syntax and Semantics of A+" and "Workspaces and Scripts".)

A program consists of a header and a body separated by a colon (:). The header consists of the name of the program along with its argument names. The body is an expression or expression group. The result of a program is the result of the expression group that forms the body. See "Function Definitions" and "Operator Definitions" for detailed descriptions of headers and bodies. A simple example of a function definition is

```
    r f p:¯1+(1+r÷p)*p     ⍝ Simple to compound interest, with p periods.
```

Scope of Names

The value of a local variable in a program (defined function or operator) is strictly local: it can be seen only within that program, and not by calling programs or called programs. This rule for local variables is sometimes called static scoping, or lexical scoping. (Note, however, that if the local variable is a mapped file, the file value can be seen elsewhere; see "Mapped Files".)

The arguments to a program are local to that program.

The name of the program is global, and furthermore the unqualified form of that name is also considered global within the program, and therefore any occurrence of it refers to the program.

Fully qualified names within the program - ones that explicitly refer to contexts, like `ctx.var` and `.fn` - are always global, and so are the names (in symbol form) appearing in Value and Value in Context expressions. If possible, these are the preferred ways to indicate that a name is global.

Names in arguments for Execute are not seen during parsing of the program and play no role in determining whether unqualified names are local or global.

Any unqualified name not covered above is local if, anywhere in the program, it is the target, without extra parentheses, of an ordinary or Strand Assignment, for example `a←b+c` or `(a;b)←(c;d)`.

Otherwise any such name is global: either it is only referenced, not specified, in the program, or wherever it appears as a target it is parenthesized, for example `(a)←b+c` or `((a);(b))←(c;d)`. Because of the way this aspect of Strand Assignment is implemented, at least currently such an assignment does not make a name local if there is more than one right parenthesis between the name and the left arrow; thus the assignments `((a;b))←(x;y)` and `(c;(d);e)←(x;y;z)` force only `e` to be local.

The context for a visible unqualified global name is the context of the program, i.e., the context explicitly given in the program name in the header, or, if that name is unqualified, the current context when the program was defined. A statement that may later be executed as a Context command, `⍎"$cx ..."`, does not change the context during program definition. See "Assignment" and "Contexts". Within a program, the program's context can be obtained by `0#∪_name<{%}`.

A character string or substring or symbol that may later be interpreted as an unqualified global name, using Execute or Value, is not a visible unqualified global name (when the program is defined). The context that is used for such text is determined only when it emerges as a name, i.e., when it is executed, and it will be whatever context is current at that time. That current context could be the context when the program was invoked or one established by the execution of a Context command within the program or a program called by it.

Function Syntax

A function is a program that takes from zero to nine data arguments and returns a result. Examples of functions that take 0 (niladic), 1 (monadic), 2 (dyadic), and 3 arguments are shown in Table 4.3, "Function Call Expressions and Function Header Formats". (Although a dependency is a kind of niladic defined function, it is different enough in form and operation that it is treated separately, in "Dependencies", and ignored here.) For monadic and dyadic functions, for both definition and use, the two possible forms are shown: infix and general. No matter which form is used when the function is defined, either form can appear when it is used.

Operator Syntax

An operator definition is a program that takes

1. one function operand, two function operands, or a function operand and a data operand, and
2. one or two data arguments,

and returns a result. It is distinguished from a function definition by parentheses in the header that surround the expression consisting of the operator name and its operands. Examples of the possible headers are shown in "Operator Header Formats".

In each form in that table, the left operand, `f`, must be a function. When is the right operand, `h`, a data variable and when is it a function? If in the operator definition it has the name `g`, it is a function. If it has any name there other than `g`, it is a data variable *unless* in *every* occurrence of it in the body of the definition it can only be a function. Examples of such occurrences are `... h¨ ...` and `... h{...} ... .`

To repeat: the left operand is always a function. If the right operand is not named `g` in the definition, it is a data variable if there is an occurrence of it in the body of the definition where it either must (e.g., `... h[...] ...`) or could possibly (e.g., `... h÷2 3 4 ...` or `... h/...`) be a data variable. Otherwise the right operand is a function.

Operators have higher priority than functions. The arguments defined within the parentheses are bound to the operators before the derived function is applied to its data arguments.

Operators have long left scope and short right scope. This is the mirror image of functions, which have long right scope and short left scope. For example, the expression `100 200 ,@1@¯1 ⍳2 3 4` is the same as the expression with redundant parentheses `100 200((,@1)@¯1)⍳2 3 4`.

Entry of Programs

Single-statement programs are normally entered on a single line, as in

```
mean x: (+/x)÷#x
```

Multiple-statement programs may be entered on one or more lines. The definition is not complete until any open quotation mark and all open braces and parentheses are closed. Be sure to see "Workspaces and Scripts" regarding continuation rules during entry of statements and functions, and asterisks that A+ supplies to indicate depth of punctuation. These asterisks are not shown in this chapter.

Here is a sample multistatement multiline function definition:

```
total m : {
   ⍝ Append row, column, grand totals to matrix.
   c←+/m;           ⍝ Column totals.
   r←+/⍉m;          ⍝ Row totals.
   (⍉(⍉m),r),c,+/c
   }
```

This function could be defined more efficiently using the Rank operator rather than Transpose:

```
total m:{
   ⍝ Append row, column, grand totals to matrix.
   c←+/m;  r←+/@1 m;     ⍝ Column and row totals.
   (m ,@1 0 r),c,+/c
   }
```

Notice that the comments are comments to lines, not to statements. They do not appear before the statement semicolons and they do not have their own semicolons. Even if there are several statements on a line, there can still be only one comment.

Defined functions can of course appear in the definitions of defined functions, as in

```
     y f x:0⌈g{y}÷g{x}
```

Notice that *g* appears in general form, not in infix form. Although either form is valid, it is a good practice, especially in scripts, to use the always unambiguous general form within the body of a definition. Then it doesn't matter whether *g* is defined before or after *f*; when it appears in the definition of *f* A+ will surely be able to determine that it is a function and not a variable.

Here is a sample operator definition, showing how the right operand can be a data value:

```
u(f p a)v:{
     case(a){
               1 ; u f  (ρu)↑v;
               ¯1; u f (-ρu)↑v;
               0 ; u f     0#v;
               ↑'Incorrect right operand'}
     }
```

In the following operator definition, both operands must be functions, as shown by the syntax. Defined operators can be used in the definitions of defined operators, like `inv` in this example:

```
     y (f iso g) x:(g inv){f{g{y};g{x}}}
```

Indentation, Blanks and Tabs

When the name of a program is entered alone on a line, without arguments or braces, A+ displays the program definition. It is shown just as it was entered, with the spacing preserved, with two exceptions:

1. Any invisible characters (blanks, tabs, ...) before the first visible character in the function are eliminated. If you want to see on entry what will be displayed later, start the header at the left margin, eliminating the blanks supplied at the beginning of each input line in an A+ session.
2. If the function was entered in an A+ session (rather than in a script or other file), the spacing within each line is preserved but the indentation is generally not: the first nonblank character of each line after the first is shown at the left margin. In these lines, it is specifically the blank character that is dropped, however; other invisible characters are kept. You can force the original indentation to be honored in successor lines by using tabs. All tabs and any blanks that *follow* a tab are kept.

Thus, if in an Emacs A+ session you enter

```
     f1{x;y}:{
              if (x=0) {
                     ...;
                     ...};
                 else  {
                     ...}
              }
```

using just blanks to obtain the indentation, you get this display

```
     f1
f1{x;y}:{
if (x=0) {
...;
...};
else  {
...}
}
```

whereas if you enter

```
f1{x;y}:{
        if (x=0) {
           ...;
           ...};
           else  {
           ...}
       }
```

using tabs, at least as the first character on each successor line, you get a display exactly matching your input.

Formatting a Defined Function or Operator

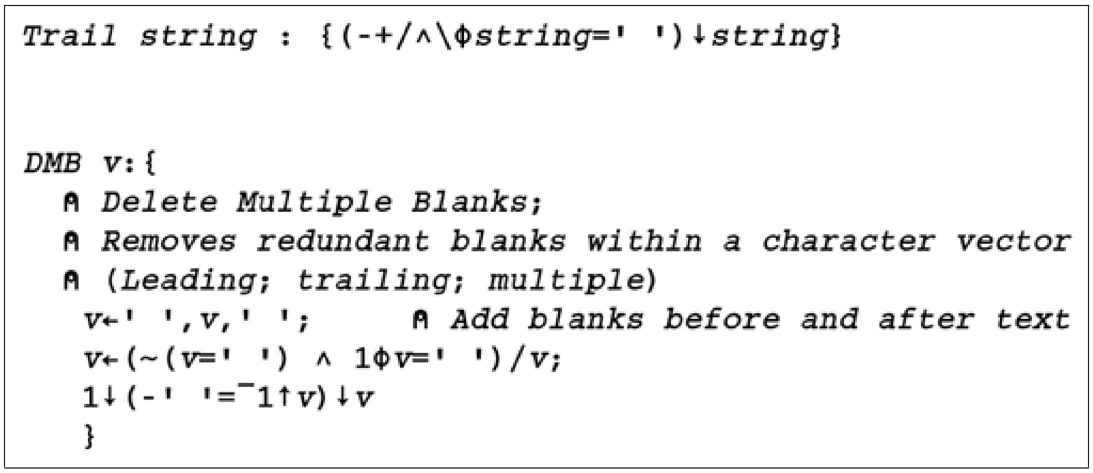
At times it may be convenient to convert a defined function or operator to text. Examples of the need for this would include printing formatted reports of code from the workspace or dynamic inspection of the code. However, it may not be obvious just how to format these objects into text. It is done like this:

```
     avg
avg v : {(+/v)÷#v}

     mat←⍕<{avg}      ⍝ here is the technique

     mat
avg v : {(+/v)÷#v}

     ⍴mat
 18
```

This is the format (⍕) of a function scalar.

15. A+ Function Definition Mode Tips: "Do"s and "Don't"s

Here is a sample showing a couple of small defined functions, with some discussion regarding their respective anatomies:



```
Trail string : {(-+/∧\⌽string=' ')↓string}


DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

A. A unit of code that may be called a "program" or a "subroutine" in other languages is called a "**function**" in A+. (This term betrays some of the mathematical origins of the A+ language.) A function takes in data (in the form of its one or more arguments), processes it (with the code within its definition), and returns new data (in the form of its one or more results).

   Here are two small sample functions, named "`Trail`" and "`DMB`".

   A function may be defined entirely on one line (as shown with the "`Trail`" function), or on multiple lines (as shown with the "`DMB`" function).

   Although functions can be created within the A+ environment, they cannot be edited or saved from within the A+ environment; it is therefore most common to enter functions from within the Emacs or XEmacs editor.

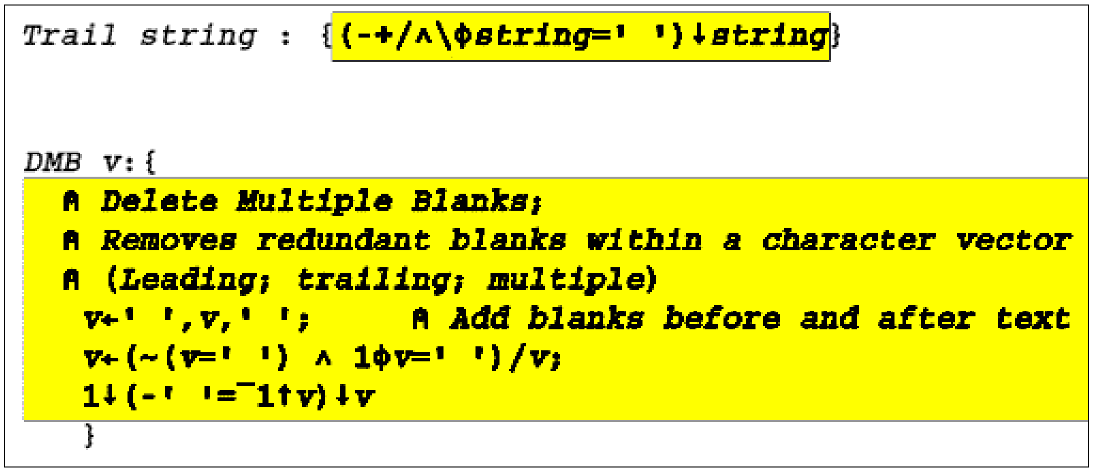
```
Trail string : {(-+/∧\⌽string=' ')↓string}


DMB v:{
   ⍝ Delete Multiple Blanks;
   ⍝ Removes redundant blanks within a character vector
   ⍝ (Leading; trailing; multiple)
    v←' ',v,' ';      ⍝ Add blanks before and after text
    v←(~(v=' ') ∧ 1⌽v=' ')/v;
    1↓(-' '=¯1↑v)↓v
    }
```

B. This is the "**header line**" of the function definition. It establishes the name of the function, and the number of arguments that the function may use. See "Types of Headers."

```
Trail string : {(-+/∧\⌽string=' ')↓string}


DMB v:{
   ⍝ Delete Multiple Blanks;
   ⍝ Removes redundant blanks within a character vector
   ⍝ (Leading; trailing; multiple)
    v←' ',v,' ';      ⍝ Add blanks before and after text
    v←(~(v=' ') ∧ 1⌽v=' ')/v;
    1↓(-' '=¯1↑v)↓v
    }
```

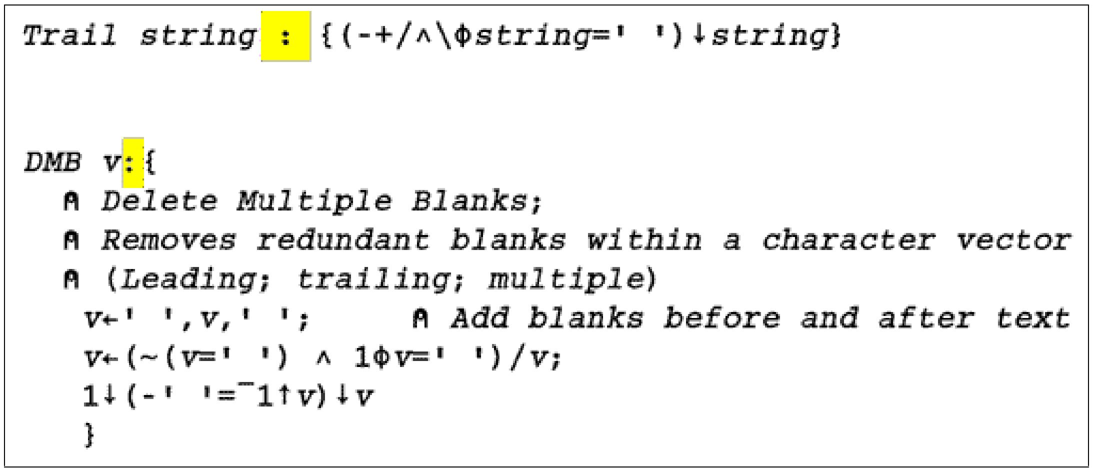
C. An **opening brace** ("{") begins the definition of a multi-line function. Braces are *optional* for a function that is defined entirely on one line.

You may type the definition in on the same line as the header, or you may press the Enter key and put the code that defines the function on successive lines under the heading. The definition is ended when you enter a **closing brace** ("}") which is paired with the opening brace.



```
Trail string : {(-+/∧\⌽string=' ')↓string}


DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

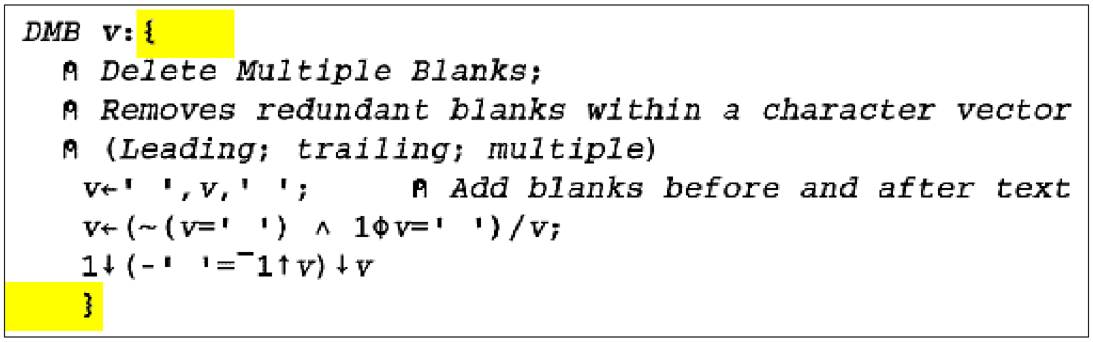
D. Everything between the outermost braces constitutes the "**body**" of the function. (The braces are optional for a function defined on one line.)



```
Trail string : {(-+/∧\⌽string=' ')↓string}


DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

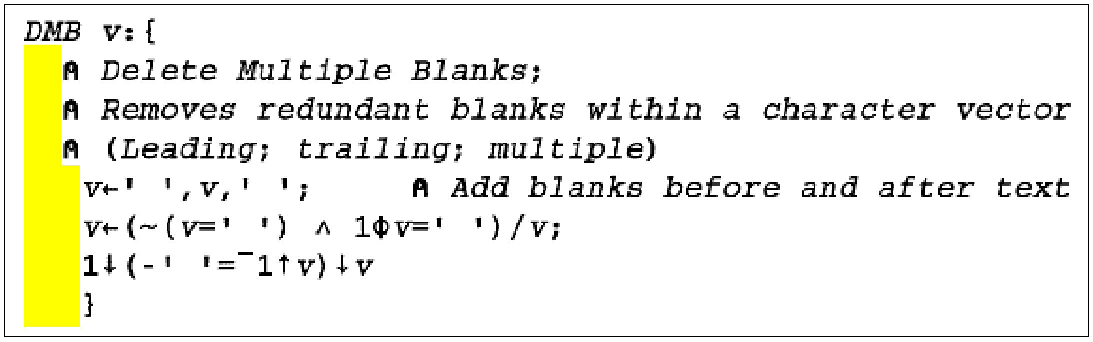
E. A **colon** separates the header line from the body of the function. Spaces may optionally be placed before and after the colon, but are not required.

In general, **spaces** may be inserted between names and symbols in a function in order to aid visual clarity. These spaces have no effect on the operation of the code.



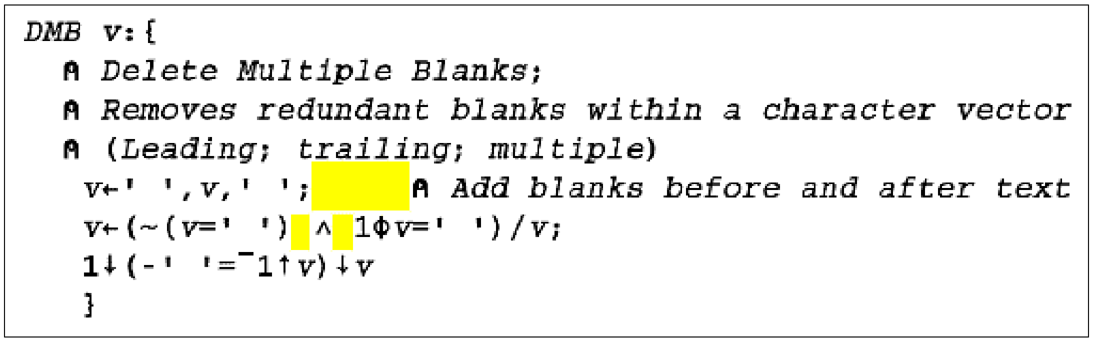
```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';     ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

F. By convention (only), it is common to have the heading and opening brace on a line by itself, and the closing brace also on a separate line.



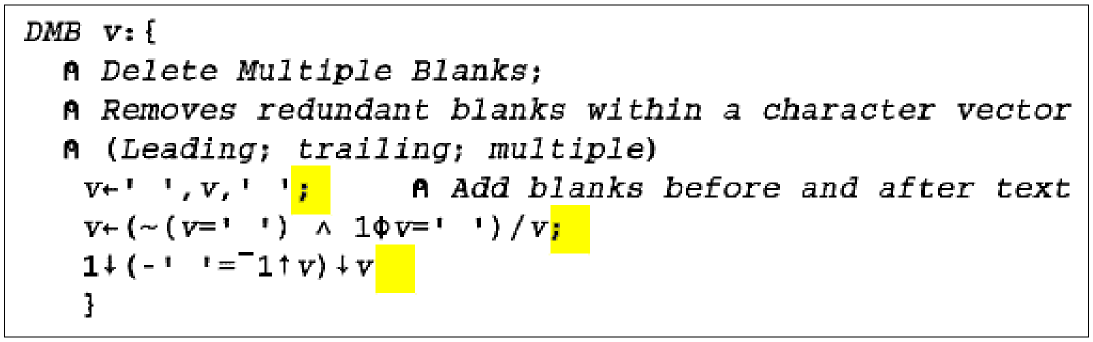
```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';     ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

G. The **indentation** of the lines must be entered manually; lines are not automatically indented. The system, however, will retain the indentation that you provide. It may be desirable to use indentation for clarity. How much each line is indented is up to you. You can enhance the readability of your functions through the judicious use of indentation, and in particular, by being *consistent* with your indentation style.



```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';     ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

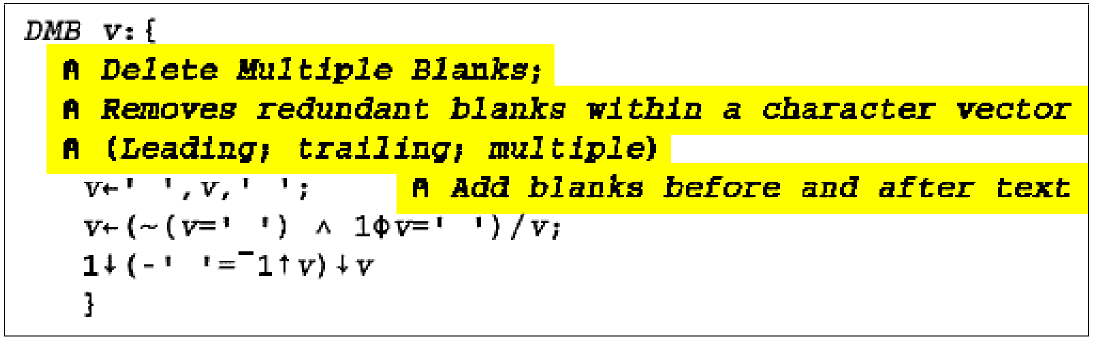
H. **Extra blanks** may optionally be used between A+ symbols. This may be desirable for clarity (although it has no effect at all on the execution of the code). Blanks may help to visually separate blocks of code within a line.



```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

I. A **semicolon** *must* be used at the end of each line - except for the last line of code in the function: the last line *must not* end with a semicolon. (Comment lines do not need semicolons; see below.)

If semicolons are not used properly, a common symptom is that the function might execute with no error reports, but may not return a result to you.



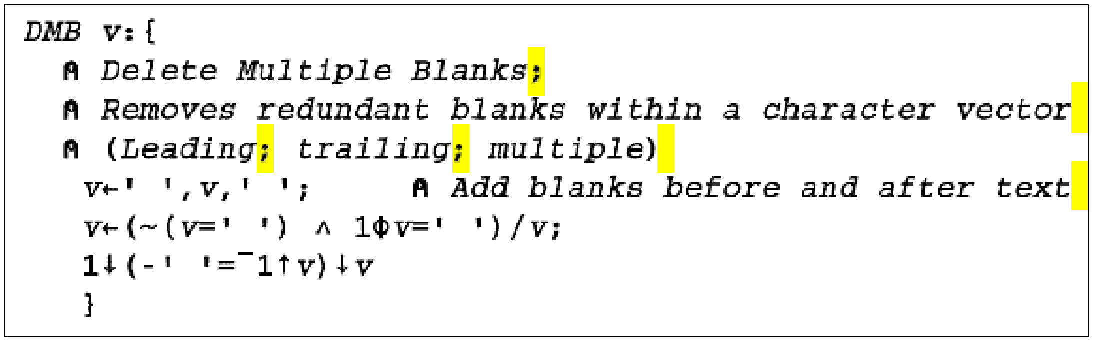
```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

J. Any line which begins with a lamp symbol ("⍝") is a **comment line**. Nothing else on a comment line is evaluated or executed (...a lamp is used for illumination only...).

It is *very* desirable to include comments with your code. Many people commonly start a function definition with one or more comment lines which explain the overall purpose of this particular unit of code and describe its arguments and result.

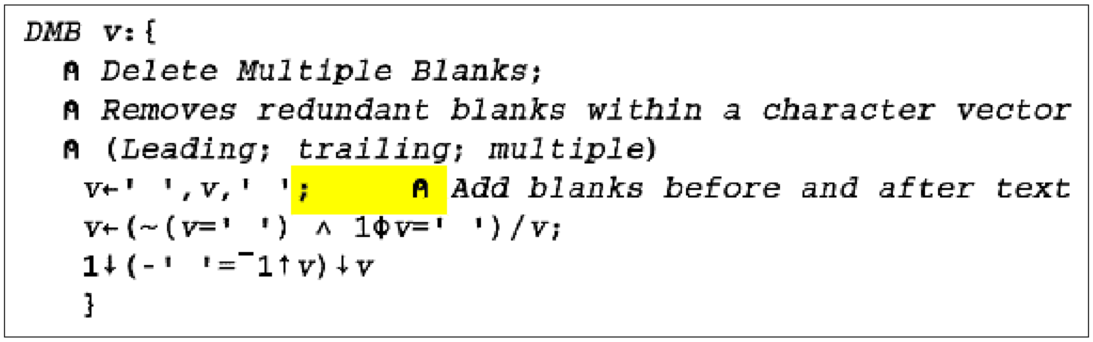
Comments may appear anywhere within the body of a function. Everything to the right of a comment symbol, up to the end of the line, is considered to be part of the comment. Therefore, whether a comment line ends with a semicolon or not is immaterial; if a semicolon character is present at the end of a comment line, it is just part of the comment.

You cannot put an ending brace on a comment line, because the brace will just be considered to be part of the comment.



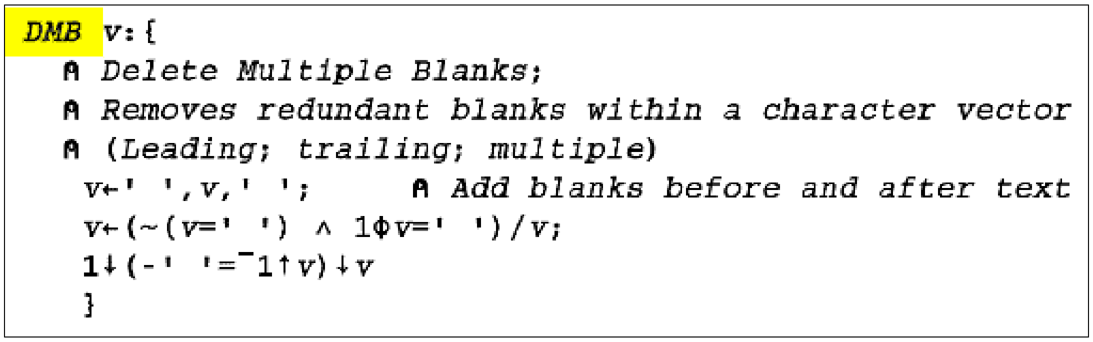
```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

K. If there is a **semicolon** character *within* the **comment**, it is simply part of the comment. A semicolon to the right of a comment symbol does *not* end the comment; a comment may only be ended by the end of the line (i.e., by pressing the Enter key).



```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

L. A **comment** which appears to the right of a line of code *must follow* that line's ending semicolon.



```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';      ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
   1↓(-' '=¯1↑v)↓v
   }
```

M. The **header** of a function does not indicate an assignment of a result (as it does in other APL implementations), because *all* A+ functions return results.

```
DMB v:{
  ⍝ Delete Multiple Blanks;
  ⍝ Removes redundant blanks within a character vector
  ⍝ (Leading; trailing; multiple)
   v←' ',v,' ';       ⍝ Add blanks before and after text
   v←(~(v=' ') ∧ 1⌽v=' ')/v;
  1↓(-' '=¯1↑v)↓v
   }
```

N. An **assignment of a result variable** is also *not needed* within the body of a function. The result of the function is simply the result of the *last operation*, whether that was an assignment to a variable (of any arbitrary name) or the result of an operation with no assignment.

16. Dependencies

A dependency is a global variable (the dependent variable) and an associated definition that is like a function with no arguments. Values can be explicitly set and referenced in exactly the same ways as for a global variable, but they can also be set through the associated definition.

In this chapter the basic characteristics of dependencies are developed through a series of examples, and are then collected in a definition of dependencies. Then, itemwise dependencies are defined and exemplified. Finally, cyclic dependencies are discussed.

Creation and Deletion

A dependency is created by specifying its definition, in the form `name:expression`. The variable name can be either new or pre-existing. The body of the definition (which follows the colon) is entered in the same way that the body of any defined function is entered, with the same syntax rules, and the variable name can appear within it. The description of Execute in Context shows how you can define dependencies of the same form in several contexts at once. Because a dependency definition contains a colon, it cannot be a statement in a defined function or another dependency definition. To create a dependency within such a definition, use Execute in Context or the like.

A dependency, both variable and definition, can be deleted, and its storage freed, by the Expunge function (`_ex`) or command (`$ex`). The Remove Dependency Definition function, (`_undef`) and command (`$undef`), as their names imply, delete only the definition, leaving an ordinary variable with all its other properties intact, e.g., its value and any callback function set on it (see "Callback Functions"). They do not cause an evaluation before they remove the definition.

Evaluation

The first time a dependency is referenced after it is created, (assuming it has not been explicitly specified in the meantime) its definition is evaluated and the result is both saved and returned. When the dependency is referenced again, the saved value is returned if it has not been marked invalid in the meantime; otherwise the definition is evaluated and the new result is saved and returned. The saved value can also be set through ordinary Assignment, and this value will be returned until it is marked invalid. For example:

```
     a←3        ⍝ Define a global variable a
     b:a*2      ⍝ Define a dependency b
     b
 9              ⍝ The definition of b is evaluated.
     b
 9              ⍝ The saved value of b is returned.
     a←4        ⍝ The saved value of b is marked invalid.
     b
 16             ⍝ The definition of b is evaluated.
     b←13       ⍝ Specify a value for b
     b
 13             ⍝ The saved value of b is returned.
     a←5        ⍝ The saved value of b is marked invalid.
     b
 25      ⍝ The definition of b is evaluated.
```

The rules of localization within the definition of a dependency are the same as those for ordinary functions.

When a dependency is defined, the value, if any, of the dependent variable is marked invalid - not erased, just marked invalid. After that, the saved value (which continues to be retained) is marked invalid (evaluation needed) whenever the source of a value or the dependency itself is changed - i.e.,

- the value of a visibly used global variable is changed;
- another dependency visibly used in the definition has its value marked invalid or explicitly set;
- the definition of a visibly used function or operator is modified;
- the dependency is redefined.

A visible use of a name occurs when the name appears in A+ code directly, and not in a character string or a symbol. A use that is not visible is an implicit reference, which occurs through the use of Execute (⍎) or Value (%).

For example, if

```
     a←100
     b:a*2
     f x:3+x
     df:a+b+f 2000
     df
 12103            ⍝ The definition of df is evaluated.
```

then any one of the following will cause the value of `df` (saved by the last input line) to be marked invalid:

```
     a←50
     b:a*3
     b←625
     f x:4×x
     df:a+b+f 3000
```

Only changes to global variables, functions, and dependencies that have a visible use will cause the saved value of a dependency to be marked invalid: implicit references will not. Moreover, changes to global variables that are assigned but not referenced in a dependency definition will not cause its saved value to be marked invalid. For example, in the dependency `df` defined below, changing `x` will cause the saved value of `df` to be marked invalid, but changing `y` or `z` will not, because there are no *visible* uses of them, and `c` is only set, not referenced (it is the right argument of the specification, `(⍎'y')+%`z`, that is referenced). Entering a new definition of `df` will of course mark any saved value invalid.

```
     df:x+.c←(⍎'y')+%`z  ⍝ The saved value of df is marked invalid.
     x←10
     y←100
     z←1000
     df
 1110
     x←20      ⍝ The saved value of df is marked invalid.
     df
 1120
     y←200     ⍝ df not marked: no visible use in the definition.
     df
 1120
     z←2000    ⍝ df not marked: no visible use in the definition.
     df
 1120
     c←¯50  ⍝ df not marked: in the definition c is only assigned, not
referenced.
     df
 1120
```

When a dependency definition is executed, the dependent variable is first marked valid. This validation allows the variable to be referenced during evaluation, either within the definition or in asynchronous execution such as a callback, and it provides a fallback value, which may possibly be of some use.

If execution of a dependency fails, a suspension occurs, as here, where `n` has no value:

```
     m:3×n
     m
 .n: value
```

When the suspension is cleared, one of two things happens; if the dependency does not have a saved value, as in this example, then a value error on its name occurs:

```
*     →
 .m: value
*
```

If, however, it has a saved value then that value has been marked valid. It is returned when the suspension is cleared. Continuing the example:

```
*     →    ⍝ Clear the previous suspension in this example.
     m←5   ⍝ Now m has a saved value that is not marked invalid.
     n←'a' ⍝ Marks m's saved value invalid and makes its definition
erroneous.
     m
 ×: type   ⍝ The saved value has now been marked valid.
*     m
* 5
*     →    ⍝ Clear the suspension.
 5         ⍝ The saved value is returned.
```

More examples of dependencies are presented in the chapters that follow.

Dependencies Defined

A dependency is a *global variable*, the dependent variable, with an associated *niladic definition*. It is established by

```
     name:definition
```

or

```
     name[indexname]:definition
```

The latter form is an *itemwise* dependency and is treated separately below.

The rules of localization for a dependency definition are the same as those for ordinary functions, and indeed it is interpreted exactly as the body of a niladic function would be. `name` may appear within `definition`.

A dependency has a *saved value* once its *current definition* has been evaluated or a value has been explicitly assigned. If the variable had a value before the dependency was established, the dependency has a saved value immediately after establishment, but that value is marked invalid.

Evaluation of Dependencies

When referenced, the value of a dependency is determined as follows:

- If it has no valid saved value, its definition is evaluated and the result of the evaluation is the value, which is also saved for future references. If it has an invalid saved value, the invalidation is removed as the first step of evaluation, and if the evaluation fails this saved value is returned when the execution suspension is cleared. If the dependent variable is bound to a display class and the definition yields an improper value for that class, then the saved value will be retained (no longer marked invalid, of course), in some cases with no warning or error message.
- If the dependency has a saved value that is not marked invalid, the saved value is the value.

A dependency that is displayed by s in any sector or workspace and is not iconized is referenced each time A+ goes through its main loop, and likewise if it is bound to the `` `reference`` class.

Note that if a dependency has a stored value that is marked invalid, a reference triggers an evaluation not only when the reference requires the value, as in `` `dep is `table``, but also when the reference does not require the value, as in `` `class of `dep`` (supposing `` `dep`` not shown, so that its invalidation did not trigger an immediate evaluation).

Invalidation of Dependencies

The saved value of a dependency is marked invalid:

- when it is first defined (if it has a saved value), and when its definition is modified;
- when a change occurs to a global object that is visibly referenced (see below) in its definition, and the definition is not currently being evaluated, nor a callback on the dependency currently being executed.

A global object is visibly referenced if its name occurs as the source of a value (not just as a specification target) outside an argument to Execute or Value. A change to such an object occurs when:

- a global variable is explicitly modified (a mapped-file global variable can change value without itself being explicitly changed);
- the saved value of a dependency is marked invalid or its saved value or definition is explicitly modified;
- the definition of a function or operator is changed.

Note that a global variable that is set in a dependency but never referenced is not a visibly referenced object.

The value of a dependency is marked valid just before its definition is evaluated (if there *is* a saved value) and whenever a value is explicitly Assigned to the dependent variable - including, of course, at the end of a successful execution of its definition.

A change to a visibly referenced global variable made during evaluation does not cause the dependent variable's value to be marked invalid. Consider

```
      m:{m←m+n; (n)←10×n; m+n}
      m←100
      n←1
      m
 111
```

`n←1` marks the saved value of `m` invalid, so the next statement triggers evaluation of its definition. The invalidation mark is changed to an under-evaluation mark, and `m←m+n` uses the previous saved value to produce a new saved value (101). Respecifying `n` (as 10) at this point does not invalidate this new saved value, since it is marked as under evaluation, and it is used in the final `m+n`.

On the other hand, a new definition unconditionally invalidates a saved value, as in

```
      m:{m←m+n; ⍎"m:n"; m+n}
      m←100
      n←1
      m
 2
```

where in the final `m+n` the new definition (`m:n`) is executed to evaluate `m`.

Code in callbacks is treated in this respect like code shown and called explicitly in the body of the dependency. When a dependency that is being evaluated triggers a callback function on another variable, any change made to a variable that is visibly referenced in the dependency does not mark the saved value of the dependency invalid. E.g.,

```
      {a←0; b←⍳3;}
      a:(c)←10×b      ⍝ Marks the saved value of a invalid.
      f{}:(b)←10×a    ⍝ Set b in a callback function.
      `c _scb (f;)    ⍝ Trigger a callback from c in the dependency definition.
      a               ⍝ Dependency is evaluated.
 0 10 20
      b               ⍝ See that b was set in the callback.
 0                    ⍝ Yes, from a's original stored value.
      a
 0 10 20              ⍝ The value from the evaluation.
```

Note that an evaluation is not completed until all callbacks triggered by it have been finished.

If a variable is set during its *own* callback, its dependents are invalidated (again).

Itemwise Dependencies

Itemwise dependencies are intended to reduce redundant computation by allowing the *items* of a dependent variable to be marked invalid separately. Only the first axis is singled out in this way. The form is the same as for ordinary dependencies except that Bracket Indexing and an index name are used in the definition:

```
    name[indexname]:body_of_definition
```

The variable named by the index name is local and can be used for any purpose within the definition. Since this variable is local, the index name must be unqualified.

A simple example of an itemwise dependency is

```
    y[i]:f{m[i];c}
```

Whenever `c` is changed - even just one of its items -, all of the saved value of `y` is marked invalid, but when items of `m` are appropriately changed, then only the corresponding items of the saved value of `y` are marked invalid.

Recognition of Itemwise Changes

For itemwise invalidation of the dependent variable, any change in a variable on which the dependency depends itemwise must be made by Bracket Indexing, Choose, or Append Assignment. In the example just shown (leaving aside for the moment an incompatibility between Append and the others), either

```
   m[3756]←expression
```

or

```
   (3756#m)←expression
```

marks just `y[3756]` invalid, and, more generally, each of

```
   ((expr1)#m)←expr2
```

and

```
   m[expr3]←expr4
```

and

```
   m[,]←expr5
```

marks items of `y` invalid.

All of `y`, however, is marked invalid by

```
   ((3756=⍳#m)/m)←expression
```

and, of course,

```
   m[;5]←expression
```

Moreover, either pair of statements

```
   m[3756]←expression
   m[,]←expr5
```

or

```
   m[,]←expr5
   m[3756]←expression
```

mark all of y invalid (assuming no updating of y between the statements). In effect, when y is updated, it is updated by just one of a Bracket Assignment, an Append Assignment, or an ordinary Assignment. Therefore, itemwise invalidations caused by a Bracket Assignment and an Append Assignment cannot be pending at the same time. (Itemwise invalidations from several Bracket Assignments or several Append Assignments *can* be pending simultaneously, of course.)

If a is a large itemwise dependency and you know at some point in the code that you may be about to cause a total invalidation in one of the two manners described just above, you might consider avoiding the total invalidation by forcing an itemwise evaluation with a trivial statement such as

```
{ρa;};
```

In `y[i]:m[i]` *all* of y is marked invalid by `m[;1]←expr` . As discussed above, Append Assignment is recognized as an itemwise change, except that a total invalidation occurs when otherwise an itemwise change from an Append Assignment would be pending at the same time as an itemwise change from a Bracket Indexing or Choose.

Recognition of Itemwise Dependence

In an itemwise dependency definition, only the form shown, Bracket Indexing with just an index name and no semicolons, is allowed on the left. If any other form appears, the dependency is total, no matter what the rest of the definition is like.

In the body of the definition, to the right of the colon, a similar, but slightly less stringent, rule holds for each variable. To have the dependency depend itemwise on a variable, every referencing of that variable must be a Bracket Indexing using just that same index name for the first axis. None of the following definitions will allow updating of less than all of ns:

```
ns[i]:(sr+nw)[i]         ⍝ An expression, not a variable, is being indexed.
ns[i]:{j←i;sr[j]+nw[j]}  ⍝ The index has same value, but not same name.
ns[i]:(i#sr)+i#nw        ⍝ Choose, not Bracket Indexing, used in the def.
ns[i]:sr[i+1]+nw[i+1]    ⍝ Indices consist of more than just index name.
ns[i]:sr[i]-sr[i-1]      ⍝ The bad occurrence negates the good one.
```

On the other hand, indexing along other axes does not interfere with itemwise invalidation:

```
     ns[i]:sr[i;⍳50],@1 nw[i;50+⍳50]
```

allows updating of only the affected items of ns. Obviously, any dependence on a function or operator is total.

Note: At the present time, the parser may confuse the dependent variable with a function when parsing the body of an itemwise dependency definition. If such a parsing error occurs, replace the dependent variable, `b`, say, in the body by its fully qualified form, `cxt.b`, say, if the context is known and otherwise by `%`b` (or more likely `(%`b)`) to give the parser the hint it needs.

The Time and Form of Evaluation of an Itemwise Dependency

When an itemwise dependency is defined or its definition is changed, its saved value, if any, is marked invalid, just as is done for any dependency. If the next event for it is a reference, its definition is evaluated, with the index name being given the value Null, to indicate all items (see next section). If at some later time any change occurs in a function, operator, or variable on which it depends, and that change is not recognized as itemwise, then the saved value is again marked invalid. If still later the definition is evaluated, the value of the index name is again the vector Null (again, see next section). Just as for an ordinary dependency, an evaluation with Null index for a variable bound to a screen display class can lead, sometimes with no warning or error message, to a retention of the saved value (no longer marked invalid).

Now suppose a dependent variable is referenced after a series of changes that are recognized as itemwise to variables on which it depends itemwise. (Indices modified by a callback may not be recognized.) Suppose further that no total invalidation has occurred. Then the dependency definition is evaluated with an index vector consisting of the indices for which the changes were recognized, listed in the order in which the changes took place, but without duplication. This vector may contain all indices of the dependent variable; it will nevertheless not be transformed into the Null. Just as for a total invalidation, an evaluation for a variable bound to a screen display class can lead, sometimes with no warning or error message, to a retention of the saved value (no longer marked invalid). Furthermore, for any dependent variable, evaluation with a value that is impermissible for Bracket Indexing leads to validation of the saved value; e.g., if `int[i]:fl[i]` and `fl` is set to a floating-point vector, `int` is set to an integer vector, `fl[2]` is set to a value that cannot be coerced to an integer, and then `int` is referenced, the evaluation fails and the saved value of `int` (no longer marked invalid) is returned, without any error or warning message.

If a Selective Assignment is explicitly made to a dependent variable and there are pending itemwise assignments, the definition is evaluated, for all the pending indices, before the Selective Assignment is made.

When any reference is made to an itemwise dependency, including one by Bracket Indexing, if the saved value is marked invalid, even just itemwise, its definition is evaluated before the reference is executed. Indices are not compared to see whether only valid items are being referenced, so that an evaluation is not actually needed.

If a dependency is defined and the dependent variable is explicitly assigned a value before any reference, then a reference will not trigger an evaluation until one of the variables upon which the dependency depends is changed. In particular, itemwise dependency on a particular variable may only gradually be reflected in the value of the dependency, as illustrated in this example:

```
     b←10+ι10
     a[i]:{↓i;↓'---';b[i]}  ⍝ Entire saved value of a marked invalid.
     a←ι10                  ⍝ Now it is not marked invalid.
     a
 0 1 2 3 4 5 6 7 8 9        ⍝ Uses saved value.
     b[3 5]←103 105         ⍝ Pending change for a[3 5]
     a[0]                   ⍝ Trigger an evaluation.
 3 5                        ⍝ Just the changes since a←ι10
---
 0
```

```
      a
 0  1  2  103  4  105  6  7  8  9      ⍝ Old a, with a little new b
      b
 10  11  12  103  14  105  16  17  18  19
```

You usually want all the dependency to be honored from the beginning, and usually there is no problem, because you do not give an explicit value to a dependency that you have just defined. You should be aware, however, of the use of the saved value in the circumstances just illustrated.

Evaluation When the Index is the Null

For any vector of indices `i` that is not the Null, the itemwise dependency definition `a[i]:...` can be thought of as being executed in the form `a[i]←...`. When the index vector is the Null, however, this view would be incorrect. It is, rather, executed in the form `a←...`. The shape of `a` can be changed by a total invalidation.

To repeat, for emphasis: when there is a recognized *total* evaluation of an itemwise dependency:

- The Null is used to index the variables for which the dependency is itemwise.
- The result of executing the body of the dependency definition *is* the new saved value of the dependent variable. The Assignment to the dependent variable is not an Indexed Assignment, but rather an ordinary Assignment, and the shape of the dependent variable can be changed by it.

Cyclic Dependencies

So far only *acyclic dependencies* have been discussed, i.e., sets of dependencies with no recursive references, where no dependency depends on itself. The Debugging State system command (`$dbg`) provides a useful tool for analyzing recursive, or *cyclic* dependencies.

Example 1. Evaluating Cyclic Dependencies

The first example illustrates an important property of cyclic dependencies: if in the course of evaluating a dependency, its name is recursively referenced, that reference will be satisfied with the previously saved value (if there is no saved value, a value error occurs). Unlike a recursive function, a cyclic dependency cannot cause an infinite recursion. For example:

```
      $dbg dep 1     ⍝ Show dependency evaluations.
      a:b+2          ⍝ Marks the saved value of a invalid.
      b:a+g+2
      a←12           ⍝ Assign a and b values to avoid value errors.
      b←5            ⍝ They are now not marked invalid.
      g←10           ⍝ Saved value of b is now marked invalid, so a is also.
      a
⍝     Dependency .a evaluation entered
⍝       Dependency .b evaluation entered
⍝       Dependency .b evaluation exited
⍝     Dependency .a evaluation exited
 26
```

The new value of `a` is 26, which means the new value of `b` is 24. Since `g` is 10, `a` must have been 12 when `b` was evaluated, which was its saved value at the time.

Example 2. Interrelated quantities (rate, yield, spread)

Cyclic dependencies arise quite naturally in applications as sets of interrelated variables. For example, consider the following mutual relationships among underlying rate `u`, yield `y`, and spread `s`:

```
     y:u+s
     u:y-s
     s:y-u
```

For initialization set any two of these quantities. After that, if any one of the quantities is set the other two will be automatically updated when referenced. For example:

```
     $dbg dep 1
     (u;s)←(0.08;0.005) ⍝ Set both at once else one invalidates the other.
     y             ⍝ Defining dependency marked any saved value of y invalid.
⍝    Dependency .y evaluation entered
⍝    Dependency .y evaluation exited
 0.085
     y←0.09        ⍝ Marks the saved values of both u and s invalid.
     u
⍝    Dependency .u evaluation entered
⍝      Dependency .s evaluation entered
⍝      Dependency .s evaluation exited
⍝    Dependency .u evaluation exited
 0.08
     s
 0.01             ⍝ The saved value is returned.
```

When `y` was reassigned, the saved values of `u` and `s` were marked invalid. When `u` was then referenced its definition was evaluated, causing `s` and `y` to be referenced. The reference to `s` caused it to be evaluated, using the new value for `y` and the old value for `u`. This new value for `s` was then used with the new value of `y` to produce a new value of `u`. Because of the selfconsistency of the mathematical expressions, the new value of `u` is the same as the old one. However, these computed values would not be the same if `u` and `s` had been referenced in the opposite order. To verify this, reproduce the example up to the references of `u` and `s`, and then reference them in the opposite order:

```
     (u;s)←(0.08;0.005)
     y
⍝    Dependency .y evaluation entered
⍝    Dependency .y evaluation exited
 0.085

     y←0.09
     s
⍝    Dependency .s evaluation entered
⍝      Dependency .u evaluation entered
⍝      Dependency .u evaluation exited
⍝    Dependency .s evaluation exited
 0.005

     u
 0.085
```

Reasoning as before, `u` now has a different value from the one originally specified, but `s` does not. Of course it is not always possible to know the order in which dependencies will reevaluated, which means that one cannot be certain of the new values in cases like this. The only consistent way to use n cyclic dependencies like these is to explicitly set any n-1 of the quantities and use the dependent definition only for the remaining one.

How can the values of several cyclic dependencies be set independently without causing one another's saved values to be marked invalid? In the above example, if `y` is first specified and then `u` is specified, the specification of `u` will cause the saved value of `y` to be marked invalid, and therefore when y is referenced it will not necessarily return the value to which it was set. The answer to this problem is to set them in a strand assignment, because at the end it marks as valid the values that it has just saved for all targets that are dependencies. Continuing the above example:

```
     (y;s)←(0.09;0.005)
     y
 0.09
     s
 0.005
     u
⍝    Dependency .u evaluation entered
⍝    Dependency .u evaluation exited
 0.85
```

In practice, the values for y and s may not be conveniently available at the same time, so the information on the right of the strand assignment should be maintained in a set of auxiliary global variables. For this example these variables will be denoted by yA, uA, and sA.

```
     yA←0.09           ⍝ At some point in an application.
     sA←0.005          ⍝ Probably at some other point.
     (y;s)←(yA;sA)     ⍝ Later, and before referencing u
     u
⍝    Dependency .u evaluation entered
⍝    Dependency .u evaluation exited
 0.085
     uA←u              ⍝ Keep the auxiliary of u current.
```

By keeping the auxiliary variables current, it is not necessary for an application to respecify all n-1 dependencies in a cyclic set before evaluating the n-th: those not respecified will simply use their current saved values. Continuing the above example:

```
     yA←0.095          ⍝ A subsequent respecifying of yA
     (y;u)←(yA;uA)     ⍝ Later, before referencing s. Old value of uA used.
     s
⍝    Dependency .s evaluation entered
⍝    Dependency .s evaluation exited
 0.01
     sA←s              ⍝ Keep the auxiliary of s current.
```

The easiest way to keep the auxiliary variables current with the dependency values is to make them into dependencies as well:

```
     yA:y
     sA:s
     uA:u
```

The strand assignments can be incorporated in a function to be called when the current set of assignments to the auxiliary variables has been completed, and before the uncommitted dependency is referenced:

```
     commit x:(%¨0⊃x)←%¨1⊃x
```

For example, the above strand assignment is equivalent to `commit(`y`u;`yA`uA)`.

The use of auxiliary dependencies also makes it easy to cancel, or back out, changes simply by resetting the values of these dependencies to those when the last commitment was made. In the above example, if:

```
     cancel x:(%¨1⊃x)←%¨0⊃x
```

then the values of the auxiliary variables can be reset to their values at the time of the last commitment by:

```
     cancel(`y`u`s;`yA`uA`sA)
```

System Functions and System Commands for Dependencies

The system functions and commands discussed here are described in "System Functions", and "System Commands".

The following example will be used to illustrate the system functions and system commands that apply to dependencies.

```
     p←2 3⍴1.23 4.5 20 5.6 7 8.95
     p
 1.23  4.5  20
 5.6   7     8.95
     n←2 3⍴10 1 2 5 3  1
     n
 10   1   2
  5   3   1
     fn{x}:⌊x
```

In the definitions that follow, `m` is dependent on `p`, `n`, and `fn`; `ct` is dependent on `m`; and `gt` is dependent on `ct`.

```
     m:p×fn{n}            ⍝ Price times number.
     ct:+/m               ⍝ Column totals.
     gt:+/ct              ⍝ Grand total.
```

The system command `$deps` lists the names of the dependencies, while the system function `_nl` provides such a list as a vector of symbols when given `` `deps `` as an argument:

```
     $deps
 m ct gt
     list←_nl{;`deps}
     list
 `m `ct `gt
     $vars
 p n
     _nl{;`vars}
 `p `n
```

Dependencies are global variables, and when they have saved values, their names appear in variable lists.

```
     ct
 40.3 25.5 48.95
     $vars
 p n m ct   ⍝ The evaluation of ct caused evaluation of m, but not of gt.
```

The system command `$def dep` displays the definition of the dependency `dep`, while the system function `` _def `dep `` returns that definition as a character vector:

```
     $def gt
gt:+/ct
     def←_def `gt

     def
gt:+/ct
```

The system command `$dep name` lists all dependencies in which `name` is explicitly referenced, where `name` can be any name, but the only meaningful names are those of global variables, dependencies, or functions. The corresponding system function is `_dep`:

```
     $dep n
m
     $dep fn
m
     _dep `ct
 `.gt
```

The system function `_alldep` is the transitive closure of `_dep` with duplicates removed. The value of `_alldep `name` is a list consisting of the unique names in `_dep `name`, `_dep¨_dep `name`, etc.

```
     _alldep `m
 `.ct `.gt
```

The command `$undef v` and function `_undef v` remove the dependency definition for `v` while leaving all the other properties of `v` intact. Forcing evaluation of `v` just before such a removal will save its latest value.

Finally, there is the Debugging State system command `$dbg`. The complete definition is given above, in the chapter on system commands. The use illustrated here is `$dbg dep 1` for tracing dependency evaluation. Whenever a dependency is referenced, an "entered" and an "exited" message is displayed for each dependency whose definition had to be evaluated in order to satisfy the reference. In the above example, `ct` has already been referenced. If it is referenced again, its saved value is returned, so no dependencies are evaluated. If `gt` is referenced, however, its definition will be evaluated.

```
     $dbg dep 1
     ct
 40.3 25.5 48.95
     gt
⍝    Dependency .gt evaluation entered
⍝    Dependency .gt evaluation exited
 114.75
```

If one of the underlying variables is changed, all the dependencies are marked for evaluation, so all their names appear when `gt` is subsequently referenced:

```
     n[1;1]←4
     gt
⍝    Dependency .gt evaluation entered
⍝      Dependency .ct evaluation entered
⍝        Dependency .m evaluation entered
⍝        Dependency .m evaluation exited
⍝      Dependency .ct evaluation exited
⍝    Dependency .gt evaluation exited
 121.75
```

Note that had `m`, `ct`, and `gt` been defined as niladic functions, the evaluated results illustrated in this example would have been the same. However, niladic functions have no saved values and are therefore always evaluated when referenced. Consequently dependencies provide a generally more efficient evaluation scheme.

17. Appendix: Some A+ and APL Differences

For the most part, A+ is very similar to other dialects of APL, but there are some significant differences that are important to know about.

This is not meant to be an exhaustive list of differences, but rather, a starting point for an APL programmer, to help to prevent confusion about A+ conventions. This list does not, for instance, include references to the many extensions over most of the other APL implementations.

Object Names, System Commands, Reserved Names

1. A+ has upper- and lowercase alphabetic characters, but no underscored characters. Both cases can be used in names.
2. There are no quad-names. System function names begin with _ and are not distinguished names, and system variable names begin with `` ` ``. System variables are set and referenced only through system functions or system commands.
3. There are no local lists in function headers; names are automatically localized. All unparenthesized explicitly-assigned unqualified names are strictly local within a function. Global names can be assigned by `(global)←expression` or `.global←expression` or `cxt.global←expression` or the like. There are no semi-globals: all variables are lexically scoped and are either local or global.
4. System commands begin with "`$`" instead of "`)`".
5. `$load` does not initially clear the active workspace, acting instead like `)copy`.
6. `$si` shows the state indicator with the suspended function at the bottom.
7. The following words are reserved: `if`, `else`, `do`, `while`, `case`, `time`, `Inf`, and those names reported by the `$sfs` system command.

Environmental

8. Comparison tolerance is always `1e-13`.
9. There is no del-editor; use XEmacs in an A+ session log. In other files, use XEmacs and copy into an A+ session, or use the **F2** key to execute a line or the **F3** key to load a program into A+. Use `$load` in an A+ session to load a script, executing it in the process.
10. All expressions yield a result. Where a result is absent in APL it is often the Null in A+.

11. A+ provides an unambiguous function call syntax `f{...;...;...}` in addition to infix notation.

Language

12. Index origin is always 0.
13. `0÷0` produces a domain error.
14. `n÷0` does not produce an error when `n` is nonzero: it produces `Inf` when `n` is positive and `¯Inf` when `n` is negative.
15. Leading axis default in Reduce, Scan, Catenate, Take, Drop, Reverse, Rotate, Compress, Replicate, Expand.
16. Nor, Nand, Factorial, and Binomial are not implemented.
17. Reduce and Scan are restricted to the functions: `+×⌊⌈∨∧`. `⌊/⍳0` produces `Inf` and `⌈/⍳0` produces `¯Inf`.
18. Inner Product is restricted to the following cases: `+.×`, `⌊.+`, and `⌈.+`.
19. Outer Product is restricted to the following functions: `+×-÷*⌊⌈<≤=≥>≠|`.
20. Take and Drop are restricted to a single-element left argument.
21. Member (`L∊R`) does not ignore the rank of its right argument: it searches the items of the right argument for cells of the left argument that are the same shape as the items of the right argument. To get the traditional APL effect, ravel the right argument.
22. Find (`L⍳R`) does not ignore the rank of its left argument: it searches the items of the left argument for cells of the right argument that are the same shape as the items of the left argument. To get the traditional APL effect, ravel the left argument.
23. The arguments to dyadic `∊` and `⍳` must be the same type or both numeric.
24. Reshape uses fill elements for `L⍴R` when `L` does not contain zeros and `R` is empty.

18. Appendix: Quick Reference

List of Primitive Functions and Operators, with References

The symbols are ordered here by type of function (arithmetic, structuring, etc.) and are followed by the dyadic and monadic names:

Symbol	Dyadic Function	Monadic Function
+	Add	Identity
−	Subtract	Negate
×	Multiply	Sign
÷	Divide	Reciprocal
⋆	Power	Exponential
⍟	Log	Natural log
|	Residue	Absolute value
⊥	Decode	Pack
⊤	Encode	Unpack
?	Deal	Roll
○	Circle (sin, cos, ...)	Pi times
	Solve	Matrix Inverse
⌈	Max	Ceiling
⌊	Min	Floor
<	Less than	Enclose
≤	Less than or Equal to	(no monadic)
=	Equal to	(no monadic)
>	Greater than	Disclose
≥	Greater than or Equal to	(no monadic)
≠	Not equal to	(no monadic)
⍋	Bins	Grade up
⍒	(no dyadic)	Grade down
≡	Match	Depth
⍳	Find	Interval
∊	Member	Rake

∧	And	Stop
∨	Or or Cast	Type
⍕	Format	Default Format
⍎	Execute in Context or Protected Execute	Execute
%	Value in Context	Value
⌶	Map (see "Files in A+")	Map In (see "Files in A+")
←	Assignment or Selective Assignment	Result
⊃	Pick	Raze
#	Choose	Count
[;]	Bracket Indexing (varying number of arguments)	
↑	Take	Signal
↓	Drop	Print
⌽	Rotate	Reverse
⍉	Transpose Axes	Transpose
!	Restructure	Item Ravel
⊂	Partition	Partition Count
⍴	Reshape	Shape
⊣	Left	Null
⊢	(no dyadic)	Right
∪	Combine Symbols	Separate Symbols
,	Catenate	Ravel
~	Laminate	Not
/	Replicate	Reduce (+ × ⌈ ⌊ ∧ ∨)
\	Expand	Scan (+ × ⌈ ⌊ ∧ ∨)
.	Inner Product (+.× ⌈.+ ⌊.+)	(no monadic)

∘.	Outer Product (+ − × ÷ | ⌈ ⌊ < ≤ = ≥ > ≠)	(no monadic)
@	Rank	(no monadic)
¨	(no dyadic)	Each or Apply
⍨	(no dyadic)	Bitwise

Lists of System Functions, Variables, and Commands, with References

System Function Names and References

_alldep	All Dependent Object Names
_alsf	Association List to Slotfiller
_atts	All Attributes
_cd	Change Directory
_cfi	Comma Fix Input
_dbg	Debug
_def	Dependency Definition
_dep	Dependent Object Names
_dyld	Dynamic Load
_ex	Expunge
_excxt	Expunge Context
_exit	Exit
_fi	Fix Input
_flat	Flatten
_fmt	Format
_gcb	Get Callback
_gcd	Get Client Data
_get	Get Attribute

`_gfmtsym`	Get Format Symbols
`_gpcb`	Get Preset Callback
`_gsr`	General Search and Replace
`_gsv`	Get System Variable
`_hashstat`	Hash Table Statistics
`_index`	Permissive Indexing
`_index_of`	Index of
`_issf`	Is a Slotfiller
`_items`	Items of a Mapped File
`_load`	Load
`_loadrm`	Load and Remove
`_locals`	Locals
`_name`	Name
`_nanfind`	NaN Find
`_nc`	Name Class
`_nl`	Name List
`_ns`	Name Search
`_nsr`	Name Search and Replace
`_scb`	Set Callback
`_scd`	Set Client Data
`_scfi`	Scalar Comma Fix Input
`_set`	Set Attribute
`_sfi`	Scalar Fix Input
`_sfmt`	Screen Format
`_spcb`	Set Preset Callback
`_ss`	String Search

_ssr	String Search and Replace
_ssv	Set System Variable
_undef	Remove Dependency Definition
_valence	Valence
_wa	Work Area

System Variable Names and References

`busexit	Bus Error Flag
`corelim	Core File Size Limit
`cx	Context
`Df	Dependency Flag
`dyme	Dynamic Environment
`Ef	Execution Suspension Flag
`Gf	Protected Execute Flag
`language	Language Level
`loadfile	File Being Loaded
`majorRelease	Major Release Number
`minorRelease	Minor Release Number
`mode	Input Mode
`phaseOfRelease	Phase of the Release
`pp	Printing Precision
`releaseCode	Release Code
`rl	Random Link
`segvexit	Segv Error Flag
`Sf	Callback Flag
`si	K Stack

`stdin	Standard Input
`stop	Stop
`Tf	Terminal Flag
`vers	Version
`Xf	X Events Flag

System Command Names and References

$|	Pipe
$<	Pipe In
$>	Pipe Out
$>>	Pipe Out Append
$cd	Change Directory
$cmds	Commands
$cx	Context
$cxs	Contexts
$dbg	Debugging State
$def	Dependency Definition
$dep	Dependent Object Names
$deps	Dependencies
$Df	Dependency Flag
$Ef	Execution Suspension Flag
$ex	Expunge
$excxt	Expunge Context
$fns	Functions
$Gf	Protected Execute Flag
$globs	Global Objects

`$load`	Load
`$loadrm`	Load and Remove
`$mode`	Input Mode
`$off`	Off
`$ops`	Operators
`$pp`	Printing Precision
`$reset`	Reset
`$rl`	Random Link
`$Sf`	Callback Flag
`$sfs`	System Functions
`$si`	State Indicator
`$stop`	Stop
`$Tf`	Terminal Flag
`$undef`	Remove Dependency Definition
`$vars`	Variables
`$vers`	Version
`$wa`	Workspace Available
`$Xf`	X Events Flag
`$xfs`	External Functions

Two Tables of Examples of Operators

Examples of Operators

Expression	Description	Example
`fn@n`	Rank: `fn` applied to all rank-`n` subarrays	`⍋@1 vect +@1 mat`
`fn¨`	Each: for each scalar `s`: `< fn >s`	`i +¨ ⍳¨ n`
`fd¨`	Apply enclosed function `fd`	`f←(+;-); x f[0]¨ y`

Examples of Operators

Operator	Expression	Remarks
Rank	,@0 v←?4ρ9	Ravel applied to scalars
Rank	v ,@0 m←?3 4ρ9	Catenate applied scalar by scalar
Rank	m +@1 0 v←?3ρ9	vector-scalar (also m +@¯1 v)
Rank	m +.×@1 1 0 m	vector-vector (all other axes are all versus all)
Inner Product	x f.g y: f/x(g@0 ¯1 1)y	
Outer Product	x ∘.g y: x (g@0 0 0) y	

Data Types

Table of Data Types

Data Type	A+ Symbol	Examples
character	\`char	'per cent' and '' and "net"
integer	\`int	0 and 4 14 59 86 and ι0
floating point (IEEE double precision)	\`float	3.26 1e5 ¯3.1e-2 and 0 0ρ2.
box (enclosed object) and nested array with first item of type box	\`box	(3 4;5) and (\`ns;1;+) and <7.
empty array other than character, integer, and floating point, including the Null	\`null	() and 0 0 ρ(2;3 4;5) and 0ρ\`a\`b and 0↑(+;-;×;÷;\`s)
symbol, and nested array with first item simple symbol	\`sym	\`pp and \`rl \`stop and \`ns,(1;+)
function expression, function scalar, and nested array with first item function scalar	\`func	+ and <{+} and (+;-;×;÷) and (<{+}),\`sand (×;2 3;'a')

Examples of Primitive and Defined Functions

Examples of Primitive Functions

Sym	Description	Monadic Example	Dyadic Example
#	Tally and Choose	#⍳8 and #⍳2 3	1#⍳8 and 1#⍳2 3
⊂	Partition Count and Partition	⊂1 0 1 0 0	2 3⊂⍳5
⊃	Raze and Pick	⊃(2 3; 4 5 6)	1⊃(2 3; 4 5 6)
!	Item Ravel and Restructure	!⍳2 3	2!⍳8 and ¯2!⍳8
~	Not and Laminate	~1 0 1	4~⍳9
∨	Type and Cast	∨`ibm	`char∨⍳256
⊥	Pack and Decode	⊥'ibm'	24 60 60⊥2 5 59
⊤	Unpack and Encode	⊤`ibm	24 60 60⊤7559
I̲	Map In and Map	I̲`f	`f I̲ ⍳9 and 1I̲`f (open-write)
⍋	Grade Up and Bins	⍋?1000⍴100	¯1 0 1 ⍋ .3 1 5

Example of a Defined Function: Signal (↑) Display (↓) Stop&Trace (∧) Return (←) and Line Numbers

Function Definition	Function Line Indicator $si	Meaning
n foo t:{		
if(⍴⍴n) ↑ `rank;	foo[1]	signal rank error (for if and while, only zero is false)
if (~n) ↑ `stop;	foo[2]	signal stop
if (n<0) ← ↓n;	foo[3]	display and return n
(m←n) do{	foo[4]	repeat n times (m runs from 0 through n-1)

`^m;`
 	
`t←t,t[m]};`
 	
`t}`
	`foo[4 do 1]`
	
	`foo[4 do 2]`
	
	`foo[5]`
		stop, or trace *m*
		
		use *m* as index
		
		return *t* (last expression)

Errors and Stops

When there is an error the function and error type is shown. For example,

```
+: length
```

or

```
.foo: rank
```

This is preceded by

```
⍝[error]
```

When there is a stop the function and message are shown; names are shown fully qualified. At this point you can take one of the actions shown in the following table.

Error and Stop Actions

User Action	A+ Expression
reset (abort)	`$` or `→`
show state	`$si`
signal up an error	``↑`length``
return *x*	`← x`
resume execution, with an enlarged workspace when the error was `wsfull`	`←`
inspect the stack	`&0` (for entry 0)
change entries on the stack	`&1 ← 2 3 4` (entry 1)

A leading * is displayed for every level of resetting required to get clear. The state indicator shows a line for every function call.

The `&` symbol by itself represents self-reference; for example:

```
    fact n: if (n>0) n× & n-1 else 1
```

is the standard recursive factorial definition.

Files

File Names

Form of Name	Use	Example
`*.` or `*.+` or `*.a`	function and data definitions	`util.+`
`*.m`	mappable simple arrays	`price.m`

Printing Files

Command	Meaning	File Type
`$| ` *var* `lpr`	Pipe the file named *var* to `lpr` command.	reports or variables

Getting Around

Command	Meaning	Example
`$cd`	change Unix directory	`$cd /u/a`
`$cx`	change A+ context	`$cx stat`

Name Use

All files (`.a .+ .m .c` etc.) are in the Unix hierarchical file system, and can always be referred to by their full pathnames.
All A+ global variables are in either the root context or other contexts, and can be referred to by *context*`.`*name* .

System and Unix Command Syntax and Meaning

Examples of System Command Syntax and Meaning

Command	Meaning
`$cd` [*d*]	current [change to *d*] directory
`$load` *fns*	load *fns*`.` or *fns*`.+` or *fns*`.a`
`$cx` [*x*]	current [change to *x*] context
`$vars` [*x*]	list variables (in current [or *x*] context)
`$fns` [*x*]	list functions (in current [or *x*] context)
`$ops` [*x*]	list operators (in current [or *x*] context)
`$xfs` [*x*]	list external functions (in current [or *x*] context)
`$ex` *name*[*s*]	expunge *name*[*s*] in current context
`$cxs`	list all contexts
`$sfs`	list system functions
`$wa` [*m*]	workspace available [add *m* megabytes]. (When a wsfull error is encountered, ← alone on a line causes execution to be resumed with an enlarged workspace.)
`$pp` [*n*]	current [change to *n*] printing precision
`$rl` *n*	set random link
`$si`	state indicator stack
`$stop` [`0`|`1`|`2`]	stops [off or on or trace]
`$cmds`	list the commands
`$off`	sign off

Examples of Unix Command Syntax and Meaning

Command	Meaning
`$ls` **.a*	list all *.a* files
`$rm` *t.m*	remove *t.m*
`$mv` *f g*	move *f* to *g*
`$cp` *f g*	copy *f* to *g*
`$df /s/atmp`	disk free for workspace
`$mkdir` *d*	make a new directory *d*
`$rmdir` *d*	remove directory *d*
`$date`	display the date
Any non-A+ system command	remove `$` and pass to Unix

Atomic Vector and Graphic Characters

Graphic Characters for Atomic Vector
(`` `char⍴⍳16 16 ``) (HEX row and column labels)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00																	0F
10																	1F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	2F
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	3F
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	4F
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	5F
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	6F
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	?	7F
80																	8F
90																	9F
A0		¨	¯		≤		≥		≠	∨		×	⍪		⍨	⌿	AF
B0	⍲	¡		£	¥		□			⍱		»	?	≡	⌷	¿	BF
C0		α	⊥	∩	⌊	∊		∇	∆	⍳	∘		⎕	|	⊤	○	CF
D0			∩	⌈		↓	∪	⍵	⊃	↑	⊂	⊢	⍀	⊣		÷	DF

*E*0	*I̲*	⊖	⍎	⍝		⍷	⍫	⍒	⍋	í	e		⍞		⍕	i	*EF*
*F*0	⊗				⍉			⌽			¢	←	⍙		→	◊		*FF*
	0	1	2	3	4	5	6	7	8	9	*A*	*B*	*C*	*D*	*E*	*F*	

System Limits

Parse-Time System Limits

Entity	Limit
locals	999
items in a list notation	999
items in a vector notation	9999
nested parentheses depth	999
input strings	9999
number of function arguments	9
number of `dyld`'d function arguments	8

Run-Time System Limits

Entity	Limit
addressability	4 billion bytes
array rank	9
array dimensions	2 billion
array depth	approximately 87,000[1]
symbol length	2 billion
simultaneously mapped files (else a `maplim` error occurs)	2,000 default, but can be increased or decreased by `$maplim` and `` `maplim``.
open file descriptors	Depends on system; for Solaris, the default soft limit (as shown by `ulimit -a`) is 256 and the hard limit (`ulimit -Ha`) is 1024. The limit includes adap connections, but not mapped files, which are closed after they are mapped.

A+ recursive stack	2,000
⍒, ⍋ recursive stack	10,000
integer	roughly -2 billion to 2 billion (32 bits)
float	16 decimal digits of precision (IEEE 64 bit); results beyond the 16th digit are not valid

1. Depends on the depth of the C stack, and so on the machine architecture. [Back to table cell]