Chapter 1: About the A+ Reference Manual

This manual is intended mainly for reference. It covers the latest releases of A+ Versions 2 and 4, differences between the versions being highlighted in the text. For differences between the latest release and earlier releases of a version, see the release notes, at `http://saseo1/aplus/releasenotes.html` or as sent by email to the aplus group. For a summary of the differences between any two releases, see "Differences between selected releases", on the same page.

When it is obvious that something must be in Version 4 only, that fact is sometimes not stated: for example, for attributes that apply only to the notebook class, which is new (and stated to be new) in Version 4.

The printed *A+ Reference Manual* consists of three volumes, *A+ Language Reference*, *A+ Screen Interface*, and *A+ Extensions and Tools*. The indexes that appear in each of these volumes are identical except for page entry prefixes identifying references to the other volumes. This introduction applies to the entire manual.

The web version, for Netscape Navigator 4.0 (or equivalent browser), is at `http://saseo1/aplus`. Be sure to follow the instructions there for displaying the special APL characters.

Other Sources of Information

The web version has links to the A+ home page, the release notes, and a page from which you can obtain a list of the differences between any two releases. The home page tells about the aplus news group, and it and pages linked to it list various A+ documentation.

Organization

Because this manual is intended mostly for reference, the primitive functions, operators, system functions, system variables, and commands are listed alphabetically. In the body of the manual, they are alphabetized by their English names, i.e., names such as Add, Rank, Expunge, Print Precision, and Load. Many chapters have tables of contents at the beginning, showing English and A+ names, and (in the printed version) page numbers where the entries are defined; where the A+ names are alphanumeric except perhaps for a leading symbol, these tables are usually alphabetized by A+ name, giving you two alphabetic ways to look something up within a chapter. "Display Attributes", page S-37, has a table listing all attributes alphabetically, with brief descriptions and pointers to other chapters where appropriate.

When you are starting with the A+ name—e.g., `+`, `@`, `_ex`, `'pp`, `$load`—and don't know what chapter it is in, you can consult the index (see next section) or "Quick Reference", page 225. The first table there lists the symbols for the primitive functions and operators, ordered by category (arithmetical, logical, structural, and so on), gives the corresponding names, and indicates the chapters in which they are described. A symbol's dyadic definition has been preferred to its monadic definition where the two would dictate different categories. The next three tables are alphabetized by A+ name and give the corresponding English names.

Many other tables in this manual, however, do not include references to extended discussions of the items listed in those tables, because of space, (automatic) footnote placement, or other considerations. To find additional information about an item in such a table, look up the item in the index.

The Index

In the online version, an index or cross reference link points to the beginning of a paragraph, which is placed at the top of the frame (if enough text follows). In the printed version, a cross reference page number always refers to the beginning of a paragraph and a page number in the index usually does; when one does not, it refers to a word or phrase that is in italics, bold face, or other font that stands out from the surrounding text.

All system commands, system functions, and system variables are listed in the index under these three headings, in their A+ forms. They are listed separately under their English names, as well. Primitive functions

and operators are each listed under one or more English names. The A+ special characters are listed in the Symbols section of the index, and can be found (with references) in "Quick Reference", as already mentioned, and in tables at the beginnings of the chapters in which the functions they represent are defined. In the printed version, these tables give for each symbol the number of the page on which its definition appears.

The index includes entries for commonly used alternate names of primitive functions and operators; such an entry includes the name used here, in parenthesis, and the page number for the function or operator definition. The entry constitutes a *see* reference, while saving you the nuisance of looking elsewhere in the index when all you want is the principal reference for the function. Entries for certain other terms are of this form.

Terms

Data type and general type: There are seven data types in A+: character, integer, floating point, null, box, symbol, and function. The term *integer* is always used in this manual in this strict sense, to indicate not only a domain of values but also a particular internal representation, the one identified by the A+ type integer (`'int`, as the Type function calls it). *Integral* and *integral value* also refer to both domain and representation.

The term *restricted whole number* is used to refer to any representation, integer or floating point, of a member of the integral domain of values—the values for which integer internal representation is possible. The floating point representations in the set of restricted whole numbers need only be equal to integers within the comparison tolerance or be less than `1e-13` in absolute value. See "Comparison Tolerance", page 105. (Put another way, a scalar `x` is a restricted whole number if either (1) `'x` is `'int` or (2) `'x` is `'float` and either ´ `x` is less than `1e-13` or there is a `y` such that `x=y` is 1 and `'y` is `'int`.) The chief significance of this concept, of course, is that any floating point representation that represents a restricted whole number can always be faithfully coerced to an integer representation. All empty arrays are included in this set, as discussed below.

Many A+ functions and operators take arguments of several types, sometimes with some limitation, and it is convenient to divide A+ data objects into three classes, as they do. These classes are called *general types*:

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

Because the set of restricted whole numbers and the general types are used mostly to indicate inclusion in the domains of functions, and because most functions accept empty arrays of any type, all empty arrays are included in the definition of restricted whole numbers and in each general type. 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.

These words are used in the obvious ways, such as in the terms symbol array or symbolic array or array of symbols, meaning an array every element of which has data type symbol, and numeric array, meaning an array that has no elements or has only elements whose data type is integer or has only elements whose data type is floating point. The more elaborate term array of type symbol, on the other hand, means only that the first item of the array is of type symbol—i.e., the first item of the first item of ... the first item is a symbol; the other items of the array can be of types symbol, function, or box.

Origin: A+ enumerates lists and whatnot using the integers 0, 1, 2, ...; that is to say, A+ employs 0-origin indexing. This manual uses i-th, for any letter i, in the same sense, to agree with A+. The words first, second, third, and so on, however, are intended to convey their ordinary English meanings. Hence if an element of some list is spoken of as the third element and also as the n-th element, then n has the value 2. Digits are never used in such a construction: 0-th, 2-th, and so forth never appear again in this manual.

Comparison: A+ makes some comparisons using a tolerance. When two objects are equal within the tolerance, they are called *tolerably equal*. This term is also used in a more general way, to mean equal within the tolerance for those objects to which the tolerance applies and strictly equal for all others. Tolerably equal is said only with regard to comparisons that employ the tolerance under some circumstances.

Names and values: If the name `x` has the value 2 associated with it, one normally says simply that `x` is 2. This manual follows that or a similar usage, usually even in more complicated cases. For example, suppose `g` is the name of a function and the value associated with `x` is `'g`, the symbol form of the name `g`; the manual may simply say that `x` is a function. If `x` has the value `"'g"`, which is a character string that gives the display form of the symbol form of the name `g`, the manual may simply say that `x` names `g` when, say, `x` is an argument and its role is to supply the name of a function.

Notation

The conventions adopted for the use of APL font, capitalization, backquote, and quotation marks in this book are intended to promote simplicity, readability, and clarity.

APL font

(1) The A+ names of system variables and functions, A+ commands, A+ defined functions and variables, and the s functions (the screen management functions `is`, `show`, etc.) and, of course, the A+ primitive function and operator symbols always appear in APL font.

(2) All multicomponent A+ expressions appear in APL font, except that `'name` and `'name'` sometimes appear as just name, in ordinary text font, as discussed below.

(3) The A+ keywords (if, do, etc.) appear in APL font in multicomponent A+ expressions, as required by (2), but are shown in ordinary text font elsewhere—here, for instance.

(4) Numbers normally appear in ordinary text font. APL font is used for them, however, in these cases:
- (a) the number is part of a larger A+ expression, so rule (2) applies;
- (b) the number is explicitly an entry or display in an A+ session;
- (c) the same number or an associated number occurs nearby in a setting that requires APL font, so consistency dictates the use of that font;
- (d) the number is expressed in a form peculiar to A+ (as distinct from mathematics or ordinary English), as in `2.78e-4`.

(5) Names of attributes (for screen management) and names of the values that attributes can have are normally set in ordinary text font when they occur alone, but where a particular value is being explicitly given for an attribute, as in the “Default” column of a table, it may appear in APL font, together with a backquote or quotation marks, e.g., `'center`, `'kaplgallant'`.

Backquote

The backquote (`) is used only for system variables and where the context explicitly requires a symbol. Thus, in the “Attribute” column of a table only names appear (e.g., titlefg), and of course the response to “`show 'b`” is described as “`b` is displayed.”

Capitalization

The English names of the A+ primitive functions and operators, as contrasted to their symbols, and of the system functions, variables, and commands always appear in ordinary text font, with an upper-case initial letter and sometimes another upper-case initial: Plus, Grade up, Pi times, Natural log, Value in Context, Less than or Equal to, Inner Product, Set Attribute, Random Link, Global Objects, and so on.

Chapter 2: 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 that variable's next use. 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
- XEmacs as the 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", page 16). 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

Unix, AIX, and Linux Environments

A+ operates under various forms of Unix, AIX, and Linux. A+ processes can be started from a shell or an Emacs or XEmacs session. Hereafter, "Emacs" means Emacs or Xemacs here. Emacs provides the applica-

tion development environment for A+. You can 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 any time. Desk calculator mode is also the default for an A+ session started in a shell, but these A+ sessions are more commonly used for running applications with desk calculator mode turned off. If an application fails, appropriate entries to a log file can be written, or the A+ process can be returned 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", page E-7).

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+", page 202), 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 environments derived from APL 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

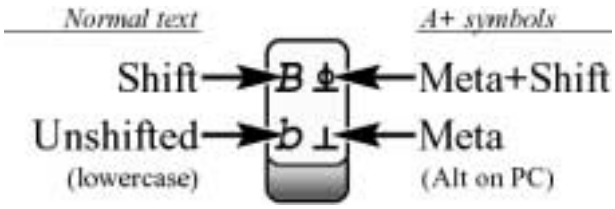
A+ uses the APL "union keyboard," which means that characters which are present on a standard keyboard retain their same positions for APL (A+) usage.

The special APL characters are entered by pressing a key while pressing either the **Meta** key, or both the **Meta** and **Shift** keys. The figures show the keyboard in two ways. The **Meta** keys (on Sun keyboards) are on either side of the **space** bar and are marked with diamonds. IBM PC keyboards have no **Meta** keys; use the **Alt** key, similarly situated, instead of **Meta**.

Note that the **Meta-Shift-m** (˝) looks like **Meta-m** (˝) and **Shift-\ (|)** but does not represent an A+ function.

Some APL symbols on the A+ keyboard are not used by A+. Following are two diagrams, one showing all of the symbols on the keyboard, and one showing only those symbols which have assigned meanings in A+.

On thse diagrams, each symbol is placed in one of four positions on each keycap, which are entered in the following manner:



Layout of the A+ Keyboard, showing the locations of all of the characters in the A+ font

Layout of the A+ Keyboard, showing only those characters which have defined meanings in A+

Chapter 3: 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, `«/ a`—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 (page 116), 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., `+.«`, 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 multi-element 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 the number on the left times 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 ⍴ (rho). For example, the result of the expression

```
      2 3 '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 ⍳ (iota) is somewhat like Reshape. It creates arrays of any specified shape whose elements are the integers 0, 1, ... For example,

```
      17      ⍝ 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, and 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.
```

Warning! In Version 2, sometimes `<` is displayed to indicate that what follows is a symbol; then no back-quote (`) is shown for the symbol.

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:

```
    ( 2 3 5)[1]    ⍝ The second item: any element of it is
15 16 17 18 19     ⍝ ( 2 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:

```
    (  4 5 6 2 3)[0;0;0]  ⍝ Any element of the first cell is
0 1 2                     ⍝ (  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  5 10   ⍝  3 (5( @1)  5 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 (rho) is called Shape. It produces the shape vector of its array argument. For example, `2 3` is the vector 2 3, and `'axrTVw'` is the one-element vector whose only element is 6.

The result of a, 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 `36` or `'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.

```
    ' 2 5
‘int
```

```
    '2.71828 3.14159
‘float
    ''axrTVw'
‘char
    ''‘pp ‘rl
‘sym
    '{+}    ⍝ Parser needs braces as hint that + is an arg.
‘func
    '<{+}   ⍝ A function scalar is also of type ‘func.
‘func
    '()
‘null
```

Next, the three types of nested arrays. The type of a nested array is the type of the first item.

```
    '<2 3 4
‘box
    ''‘rl,(;2.7 3.1) ⍝ Comma concatenates two args.
‘sym
    '(+;)            ⍝ A function scalar.
‘func
```

Last, the four types of empty arrays.

```
    ''’
‘char
    ' 0
‘int
    '0 12 10.1
‘float
    '0 4 (+;-;«; )
‘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.

```
    ⍎ 2 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

A file that shows two dimensional representations of data is distributed with the Version 2 A+ system and resides (after loading) in the `disp` context. Here is a sample of its use:

```
$load disp
disp.disp 2 3 (’ab’;‘abc‘def; 2 4; 1.1 2.2;;«)
```

```
<---------------------------------.
¨.--.       .----------. .--------.¨
¨¨ab¨       ¨'abc 'def ¨ ¨ 0 1 2 3¨¨
¨'--'       '----------.¨ 4 5 6 7¨¨
¨                       '--------'¨
¨.--------. .-.         .-.       ¨
¨¨1.1 2.2 ¨ ¨ ¨         ¨«¨       ¨
¨'--------' '-'         '-'       ¨
'---------------------------------'
```

This file is not distributed with Version 4. A more up-to-date version is available in `/common/a/disp.+`. It is used as follows:

```
     $load /common/a/disp
     disp.disp 2 3 ('ab';'abc'def; 2 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-------+           +¨+      ¨
+ -----------------------------+
```

Consult the ASCII text file `/common/a/disp.doc` for further information.

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, page 148, 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 (page 82) 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”, page 105) 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.)

Chapter 4: 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 Table B-1, page 225.

Two of the symbols can be used alone, viz., ß 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 ß 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”, page 27, 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 `a` 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      a 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* (i.e., 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):

Table 4-1: 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)	`\x`*hh*	`x`*hh*	
(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"ß "\121"  "\6a"ß "\006a"ß "\06",'a'  "\123456"ß "\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", page 177), 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 (ß). For example, the expression

```
abcß1 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 "Assignment, or Specification", page 31.

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", page 29). 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{;}`, `f{;a;b}`, `f{;;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.

Table 4-4, page 34, 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 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 Each, the Rank operator has both a function argument and a data argument. For a 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 operators. As with primitive operators, only infix notation is allowed for operator and operands. Like Each, the operand of a monadic defined operator is to the left of the operator name. For example, if the operator is `opm` then `+opm` is the derived function for `+`. For 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 Table 4-5, page 35. See also “Operator Syntax”, page 178.

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 five other symbols (`.˚/\fi`) that can appear with certain primitive function symbols, 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 five symbols in question as representing operators, namely, Inner Product, Outer Product (`˚.`), Reduce, Scan, and Bitwise.

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 Table 4-2.

Table 4-2: Special Character Sequences (Quasi-Operators)

"Operator" Name	"Derived" Functions
Bitwise	'fi (Cast and Or) ^fi ~fi <fi /fi =fi ⍷fi >fi ¤fi
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 Table 4-3 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.

Table 4-3: 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", page 28).

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 "Scope Rules for Function Expressions", page 32.

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”, page 32, and “Bracket Indexing”, page 57.

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 .

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”, page 35.

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 the “Result” section, page 89.

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”, page 32.

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 (page 66), denoted by <. A one-element object 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(<{ })¡b is a b.

Assignment, or Specification

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

```
aß1
fß+
```

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)ß(1 2 3;3 4 7;'txt')
```

Ordinary Assignment can also be expressed as (a)ßb. Any appearance of aßb inside a function or operator definition means that a will be a local variable, if a is an unqualified name. The form (a)ßb can be used to assign a value to the global variable a, provided that aß... doesn't appear elsewhere in the definition. If both aß... and (a)ß... 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 Table 7-2, page 92, 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      ª Read as: "b plus the result of c plus d."
x- y       ª 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 con-

sequently 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

```
bß(aß5;a«a)
```

is executed, the value in the second element of `b` will be 4, proving that the assignment `aß5` 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 (ß). 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 “Control Statements”, page 120. 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”, page 27), 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 Table 4-4 for a summary of function header formats.

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.

Table 4-4: Function Call Expressions and Function Header Formats

Valence	Forms
niladic	`f{}`
monadic	`f a` or `f{a}`
dyadic	`a f b` or `f{a;b}` (A position next to a semicolon can be empty for calls)
general	`f{a;b;...;c}` (A position next to a semicolon can be empty for calls)

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 Table 4-5 for a summary of operator header formats.

Note the parentheses in the forms in “Operator Header Formats”. While parentheses are not necessary in operator call expressions, they are necessary in operator definition headers to specify the function expression part. Compare with “Operator Call Expressions”, page 29.

Operator Result

The result of a defined operator, which is strictly speaking the result of the derived function, is the result of

Table 4-5: 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`

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.

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.

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 “Dependencies”, page 186. 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 (ß) 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 “Right-to-Left Order of Execution”, page 31, 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.

Chapter 5: Monadic Scalar Functions

Name	Symbol	Page	Name	Symbol	Page	Name	Symbol	Page
Absolute Value	´	37	Identity	+	38	Pi times	˘	39
Ceiling		37	Natural log		38	Reciprocal		39
Exponential	*	37	Negate	-	39	Roll	?	40
Floor	˜	38	Not	~	39	Sign	«	40

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 (ß), 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.

Absolute value ´ x

Argument and Result

The argument and result are simple numeric arrays. In Version 2, the result is always floating point. In Version 4, 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”, page 105).

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”, page 105).

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
```