REFERENCE MANUAL

KX SYSTEMS
VERSION 2.0

This manual describes the capability of K, the complete application development environment and analytical platform from Kx Systems. The manual is being provided with a demo copy of K-Lite, which is a subset of the K product.

K-Lite is a time-limited, reduced version of K which enables interested developers to learn the language and develop small applications. K-Lite consists of the K language and interpreter, GUI software, and ASCII file read/write capability. It does not include connections, file mapping, interprocess communications or runtime capabilities. K-Lite is for educational purposes, and is not intended for commercial use. Accordingly, Kx Systems does not provide training, technical support or upgrades. K-Lite is not meant as an alternative to K, but an introduction to it.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

What is K?

K is a high-level, interactive application development environment that is oriented towards performance and system integration as well as functionality. It is designed for the rapid deployment of dynamic applications that scale, that is, applications with potentially large amounts of data that require very short development and maintenance cycles. K has all the necessary components of application development: database management, graphical user interface, connectivity to other products and languages, interprocess communication, component management and a vector expression programming language, also called K. All K components provide their functions in effective, highly abstract ways that together greatly reduce application code mass and development time without subsequent loss in machine efficiency. And in cases where special functionality is required, C programs can be written and seamlessly integrated into K.

Dependencies and Data Bases

Two of the most successful programming models in application software are spreadsheets and relational databases. Spreadsheets provide a convenient, easily understood user interface for data layout and interdependence of data values based on formulas. Relational databases provide, in principle, a mathematically consistent way to create application-specific views of data stored in base tables. Despite the popularity and widespread use of these models, the popular commercially-available implementations have severe limitations in the application area where K stakes its claim: spreadsheets, while rich in features, are only effective for relatively small amounts of data, and access to relational data is much more inefficient, awkward and tedious than the relational model suggests.

K brings both the spreadsheet and relational database models to bear on problems with large amounts of data. The data in these problems tend to be organized in homogeneous fields with many items, that is, fields that each consist entirely of one type of data, such as floating-point numbers or character strings. It is crucial in these applications to manage long fields of homogeneous data effectively, and K does that for both the spreadsheet and relational database models. In the case of spreadsheet-like interactions, K dependencies describe interrelations among entire fields — not just on individual cells within those fields — while relational tables can be organized in files so as to appear like ordinary K data objects to applications, thereby bringing the complete language and graphical user interface to bear directly on stored data.

On the other hand, K is not restricted to problems in which the data are organized in homogeneous fields. The general data structure of the language is lists of lists, which accept heterogeneous mixtures of the underlying data types, but have homogenous fields as special cases. Consequently the more common logical organizations of data as individual cells in spreadsheets and records in relational databases are also available.

The Language

The language is a compact and complete formula vector language that can implement any algorithm, usually with much less code than conventional scalar-oriented, control-structure-based languages. For example, the following expression defines present value as a spreadsheet-like dependency on cash flows, discounts, and payment dates:

```
  PresentValue..d :"+/ CashFlows * Discounts[Dates]"
```

The execution of this expression, “sum the product of the cash flows and discounts at specified dates”, is as efficient as a native C program. And whenever CashFlows, Discounts, or Dates changes, a subsequent reference of PresentValue will cause its new value to be computed.

Or, consider a common relational table manipulation to link two tables A and B on a field F, which means realign the records of A and B so that they match up on field F. If F is a key field in A, that is, has unique items, then B can be left unchanged while A is realigned relative to B. The indented lines in those below give a K expression, and the non-indented line or lines below give the value of the expression. First, an index field I and key field F are defined for table A. Then the field F is defined for B. Next we look for the index in A.F of each item of B.F. Finally, we show the complete expression for realigning A relative to B, and the result of executing that expression:

```
  A.I: 1 2 3
  A.F: 2 5 7
  B.F: 5 2 5 2 2 7 2 5 7 5

  A.F ?/: B.F
1 0 1 0 0 2 0 1 2 1

  A[; A.F ?/: B.F]
(2 1 2 1 1 3 1 2 3 2
 5 2 5 2 2 7 2 5 7 5)
```

There is nothing in this expression suggesting that the table A and the fields A.F and B.F are anything other than ordinary variables. No matter how files are organized in a K database, whether as fields or more general lists, once opened they are treated like ordinary data objects. This is part of the foundation for building applications that scale: applications can be developed using local workspace test data, and then applied to data in files without any code modification.

Most languages assign the basic arithmetic functions to symbols and permit expressions to be formed in the common mathematical way, for example with the symbols between pairs of arguments, as in `x + y * z` for “x plus y times z.” K has a much richer set of primitives that can be used in this manner, including ones for common functions like sorting and searching, summarizing and updating. Many other functions can be expressed in terms of these primitives, which has two principal effects. First, the K implementation has made these few primitives as efficient as possible. And secondly, it is worthwhile for K programmers to use this expressiveness as much as possible, thereby taking advantage of the underlying efficient implementation, and indirectly greatly reducing the code volume that would otherwise result from reliance on control structures for everything beyond a few arithmetic expressions.

In fact, the primitives run at or close to maximum machine performance. On a Sun workstation (40 mhz Sparc 10) it is possible to sort 1,000,000 records per second in tables with tens of millions of records. Searching is constant time. Data access, update and append are also constant time. Sorting is linear time. The simplest and fastest algorithms are used in the implementation of the primitive functions.

Graphical User Interface

The value of a variable can be printed during an interactive K session simply by entering its name alone on a line. The value can also be shown on a display screen in any one of the usual ways: as a chart, table, button, layout, and so forth, and almost as simply as entering its name. For example, if n is a list of numbers, you may chart these as x-values by assigning “chart” as n’s display class attribute:

```
n..c: `chart
```

followed by

```
`show $ `n
```

The screen display of any value and the value itself are tightly coupled: if the screen view is edited, the value of the variable is automatically changed accordingly, and if the variable changes, so does the view. Spreadsheet-like dependencies displayed on the screen are automatically updated so that displays of all formula relationships are consistent.

The implementation of display screens is also very efficient, relying on low level graphics functions. For example, relational tables with tens of millions of records can be viewed and scrolling is instantaneous.

Connectivity

The K environment connects easily and efficiently with other products, either as subroutines callable within K applications or by way of interprocess communication. K is a very good environment for working with C and Fortran. A subroutine written in one of these languages can be dynamically loaded into the K environment, where test data can be interactively generated and results displayed in a variety of ways. Such tests may suggest changes to the source code, which is easily reloaded after changes have been made.

In addition to being able to call C subroutines from a K application, it is also possible to connect to other products with process-to-process message passing. Values can be set and retrieved from processes and expressions can be remotely executed, all based on ordinary language expressions. Any transaction can be phrased as a single message, and therefore any transaction can be made atomic.

Component Management

Application code is organized in a hierarchical name space called the K-tree. Every utility library can be assigned its own place in the hierarchy to avoid name conflicts. All screen objects — indeed all objects tied to events — are global variables in the name space, as are all their attributes. Attributes occupy special positions in the name space relative to the variables they modify. Every component in an application is a data object in the K-tree, including open data sets, programs, screen objects and their attributes, and messages to and from other processes.

About this manual

This manual provides a complete definition of the K language. See the K User Manual for examples of its use and an introduction for new users. If you are new to K and the User Manual is not available, familiarize yourself with the first two chapters before using the rest of this manual.

CHAPTER 2

SYNTAX

This chapter is concerned with two things: how to arrange symbols and names into expressions, and the way these expressions are executed. The purpose is to not only explain which lines of characters are meaningful and which are not, but also to help programmers read and write applications. Thus, the content of the chapter is not just syntax; there is often reference to the meanings of the symbols in order to place things in context.

All printable ASCII symbols have syntactic significance. Some denote verbs, that is, actions to be taken; some denote nouns, which are acted on by verbs; some denote adverbs, which modify nouns and verbs to produce new verbs; some are grouped to form names and constants; and others are punctuation that bound and separate expressions and expression groups.

The term *token* is used to mean one or more characters that form a syntactic unit. For instance, the tokens in the expression `10.86 +/ LIST` are the constant `10.86`, the name `LIST`, and the symbols `+` and `/` . The only tokens that can have more than one character are constants and names.

At various points in this chapter it is necessary to refer to the token to the left or right of another unit. Terms like “immediately to the left” and “followed immediately by” mean that there are no spaces allowed between the two tokens.

Nouns

All data are syntactically *nouns*. Data include atomic values, collections of atomic values in lists, lists of lists, and so on. The atomic values include the usual character, integer, and floating-point values, as well as symbols, functions, dictionaries, and a special atom `_n`, called *nil*. All functions are atomic data. List constants include several forms for the empty list denoting the empty integer list, empty symbol list, and so on. One-item lists use the comma to distinguish them from atoms, as in `,2` (the one-item list consisting of the single integer item 2).

Numerical (integer and floating-point) constants are denoted in the usual ways, with both decimal and exponential notation for floating-point numbers. The special numeric atoms `0I` and `0N` refer to integer infinity and "not-a-number" (or "null" in database parlance) concepts, and similarly `0i` and `0n` for floating-point. A negative numerical constant is denoted by a minus sign immediately to the left of a positive numerical constant.

An atomic character constant is denoted by a single character between double quote marks, as in `"a"`; more than one such character, or none, between double quotes denotes a list of characters. A symbol constant is denoted by a back-quote to the left of a string of characters that form a valid name, as in `` `a..b_2 ``. The string of characters can be empty; that is, back-quote alone is a valid symbol constant. A symbol constant can also be formed for a string of characters that does not form a valid name by including the string in double-quotes with a back-quote immediately to the left, as in `` `"a-b!" ``.

Dictionaries are created from lists of a special form. Functions can be denoted in several ways, all of which are presented below. In effect, any notation for a function without its arguments denotes a constant function atom, such as `+` for the Plus function.

Verbs

Each of the symbols `+ - * % | & ^ < > = ! # _ ~ $ ? @ .` and `,` represents a *verb*. They are called verbs in general, but are also called *primitive verbs* when it is necessary to distinguish them from the *derived verbs* formed by adverbs, described in the next section. Any verb can appear between nouns, as in

```
  2 - 3
```

or to the left of a noun with either nothing or something other than a noun to its left, as in

```
  - 3
  …( - 3 …
  12.5 + - 3
```

Expressions like `2 - 3` are called *infix* expressions, and those like the latter three are called *prefix* expressions.

Whenever a verb appears in one of these two ways its functional meaning is determined. For example, `-` denotes subtraction in the expression `2 - 3`, and negation in the three example prefix expressions. Every verb denotes two functions: a function of two arguments when there are nouns to the left and right, and a function of one argument when there is a noun to the right but not to the left. These functions are called the *dyadic* and *monadic* functions of the verb, respectively, or, more simply, the *dyad* and the *monad*.

Adverbs

There are three adverb symbols and three adverb symbol pairs; slash and slash-colon (`/` and `/:`), back-slash and back-slash-colon (`\` and `\:`), and quote and quote-colon (`'` and `':`). Any one of these, in combination with the noun or verb immediately to its left, denotes a new verb. For instance, `+/` denotes a verb that can appear between nouns, as in `a +/ b`, or to the left of a noun with no noun to its left, as in `+/ b`. The resulting verb is a variant of the object modified by the adverb. For example, `+` is Plus and `+/` is Sum:

```
  +/ 1 2 3 4                sum the list 1 2 3 4
10
  16 +/ 1 2 3 4             sum the list with starting value 16
26
```

Verbs created by adverbs are called *derived verbs*. The functions associated with primitive verbs are called *primitive functions*, while those associated with derived verbs are called *derived functions*.

List Notation

A sequence of expressions separated by semicolons and surrounded by left and right parentheses denotes a noun called a *list*. The expression for the list is called a *list expression*, and this manner of denoting a list is called *list notation*. For example:

```
(3 + 4; a _ b; -20.45)
```

denotes a list. The empty list is denoted by `()`, but otherwise at least one semicolon is required. When parentheses enclose only one expression they have the common mathematical meaning of bounding a sub-expression within another expression. For example, in

```
(a * b) + c
```

the product `a * b` is formed first and its result is added to c; the expression `(a * b)` is not list notation. One-item lists use the Enlist verb (comma), as in `,"a"` and `,3.1416`.

Index and Argument Notation

A sequence of expressions separated by semicolons and surrounded by left and right brackets (`[` and `]`) denotes either the indices of a list or the arguments of a function. The expression for the set of indices or arguments is called an *index expression* or *argument expression*, and this manner of denoting a set of indices or arguments is called *index* or *argument notation*. For example, `m[0;0]` selects the element in the upper left corner of a matrix m, and `f[a;b;c]` evaluates the triadic function f with the three arguments a, b, and c. Unlike list notation, index and argument notation do not require at least one semicolon; one expression between brackets will do.

Verbs can also be evaluated with argument notation. For example, `+[a;b]` means the same as `a + b`. All dyadic verbs can be used in either prefix or infix notation.

Bracket pairs with nothing between them also have meaning; `m[]` selects all items of a list m and `f[]` evaluates the nilad f. Finally, there is the special form `d[.]` for a dictionary d, which produces a list of all the attribute dictionaries for the entries of d.

Conditional Evaluation and Control Statements

A sequence of expressions separated by semicolons and surrounded by left and right brackets (`[` and `]`), where the left bracket is preceded immediately by a colon, denotes conditional evaluation. If the word `do`, `if`, or `while` appears instead of the colon then that word together with the sequence of expressions denotes a control statement. The first line below shows conditional evaluation; the next three show control statements:

```
     :[a;b;c]
   do[a;b;c]
   if[a;b;c]
while[a;b;c]
```

Function Notation

A sequence of expressions separated by semicolons and surrounded by left and right braces (`{` and `}`) denotes a function. The expression for the function definition is called a *function expression*, and this manner of defining a function is called *function notation*. The first expression in a function expression can optionally be an argument expression of the form `[name1;name2;…;nameN]` specifying the arguments of the function. Like index and argument notation, function notation does not require at least one semicolon; one expression (or none) between braces will do.

Juxtaposition and Vector Notation

There is another similarity between index and argument notation. Prefix expressions evaluate monadic functions, or monads, of verbs, as in `-a`. This form of evaluation is permitted for any monad. For example:

```
  {x - 2} 5
3
```

This form can also be used for item selection, as in:

```
  (1; "a"; 3.5; `xyz) 2
3.5
```

Juxtaposition is also used to form constant numeric lists, as in:

```
3.4 57 1.2e20
```

which is a list of three items, the first 3.4, the second 57, and the third 1.2e20. This method of forming a constant numeric list is called *vector notation*.

The items in vector notation bind more tightly than the objects in function call and item selection. For example, `{x - 2} 5 6` is the function `{x - 2}` applied to the vector `5 6`, not the function `{x - 2}` applied to 5, followed by 6.

Compound Expressions

As a matter of convenience, function expressions, index expressions, argument expressions and list expressions are collectively referred to as *compound expressions*.

Empty Expressions

An empty expression occurs in a compound expression wherever the place of an individual expression is either empty or all blanks. For example, the second and fourth expressions in the list expression `(a+b;;c-d;)` are empty expressions. Empty expressions in both list expressions and function expressions actually represent a special atomic value called *nil*.

Colon

The colon has several uses, including conditional evaluation (`:[a;b;c]`) noted previously. Its principal use is denoting assignment. It can appear with a name to its left and a noun or verb to its right, or a name followed by an index expression to its left and a noun to its right, as in `x:y` and `x[i]:y`. It can also have a primitive verb immediately to its left, with a name or a name and index expression to the left of that, as in `x+:y` and `x[i],:y`. A pair of colons within a function expression denotes global assignment, that is, assignment to a global name (`{… ; x::3 ; …}`).

The verbs associated with I/O and interprocess communication are denoted by a colon following a digit, as in `0:` and `1:` .

A colon used monadically in a function expression, as in `:r`, means return with the result r.

A verb with a noun to its right is a dyad if there is also a noun to its left, and is otherwise a monad. These are the *immediate uses* of a verb, because evaluation takes place immediately. Other uses are not immediate. For example, a verb that is modified by an adverb is not evaluated immediately, although the derived function may be. Other than in immediate use, the verb always denotes its dyad. For example, the first item in the list `(-;3)` is the minus function `x - y`. If it is needed, the monad can be specified by appending a colon to the right of the symbol, as in `(-:;3)`, where the first item is now the negate function `- x`. The monad cannot be used for derived verbs, and cannot be used when there is a noun to the right of the verb.

Names

Names consist of the upper and lower case alphabetic characters, the numeric characters, dot (`.`) and underscore (_). The first character in a name cannot be numeric. Names whose first character is the underscore are system names, and cannot be created by users. There are several system names that are syntactically like the dyads of verbs. For example, `_bin` is one, and can be evaluated by `a _bin b` or `_bin[a;b]`. Names with dots are *compound* names, and the segments between dots are simple names. All simple names in a compound name have meaning relative to the K-tree, and the dots denote the K-tree relationships among them.

At most two dots in a row can occur in names. Any simple name following two dots denotes an entry in an attribute dictionary. Compound names beginning with a dot are called *absolute* names, and all others are *relative* names.

Function Composition

Any sequence of primitive verb symbols that is not immediately followed by a noun or adverb denotes a dyadic function. Each symbol denotes its monad except the rightmost one, which denotes its dyad. For example, `(%-)[a;b]` is `%(a - b)` and the function `ne:~=` is “not equals”. Composed dyads cannot be used in infix expressions because of ambiguity problems. For example, `a~=b` means `a~(=b)`, which is quite different from `~a=b` or `ne[a;b]`.

If the right-most symbol in such a sequence is followed by a colon immediately to its right, the colon modifies that symbol and denotes the monad of that verb, and the sequence with the colon denotes a monad. For example, `(%-:)3` is `%(- 3)`.

Adverb Composition

A verb is created by any string of adverb symbols with a noun or verb to the left of the string and no spaces between any of the adverb symbols or between the noun or verb and the leftmost adverb symbol. For example, `+\/:\:` is a well-formed adverb composition. The meaning of such a sequence of symbols is understood from left to right. The leftmost adverb modifies the verb or noun to create a new verb, the next adverb to the right of that one modifies the new verb to create another new verb, and so on, all the way to the adverb at the right end.

Fixing the Left Argument of the Dyad of a Verb

If the left argument of a dyad is present but the right argument is not, the argument and verb symbol together denote a monad. For example, `3 +` denotes the monad “3 plus”, which in the expression `(3 +) 4` is applied to 4 to give 7.

Precedence and Order of Evaluation

All verbs in expressions have the same precedence, and with the exception of certain compound expressions the order of evaluation is strictly right to left. For example,

```
  a * b + c
```

is `a*(b+c)`, not `(a*b)+c`.

This rule applies to each expression within a compound expression and, other than the exceptions noted below, to the set of expressions as well. That is, the rightmost expression is evaluated first, then the one to its left, and so on to the leftmost one. For example, in the following pair of expressions, the first one assigns the value 10 to x. In the second one, the rightmost expression uses the value of x assigned above; the center expression assigns the value 20 to x, and that value is used in the leftmost expression:

```
  x: 10
  (x + 5; x: 20; x - 5)
25 20 5
```

The sets of expressions in index expressions and argument expressions are also evaluated from right to left. However, in function expressions, conditional evaluations, and control statements the sets of expressions are evaluated left to right. For example:

```
  f:{a : 10; : x + a; a : 20}
  f[5]
15
```

The reason for this order of evaluation is that the function f written on one line above is identical to:

```
  f:{ a : 10
      : x + a
      a : 20 }
```

It would be neither intuitive nor suitable behavior to have functions executed from the bottom up. (Note that in the context of function expressions, monadic colon is Return.)

Incomplete Expressions

Individual expressions can occupy more than one line in a source file or can be entered on more than one line in an interactive session. Expressions can be broken at the semicolons that separate the individual expressions within compound expressions, and when this is done the semicolon should be omitted; in effect, continuing an expression on a new line inserts a new-line character as the statement separator. For example:

```
  (a + b
   c - d)
```

is the list `(a+b;c-d)`. The effect of a semicolon at the end of a line within an incomplete expression is to introduce an empty expression. For example:

```
  (a + b;
   c - d)
```

is the three item list `(a+b;;c-d)`.

Note that whenever a set of expressions is evaluated left to right, such as those in a function expression, if those expressions occupy more than one line then the lines are evaluated from top to bottom.

Spaces

Any number of spaces are usually permitted between tokens in expressions, and usually the spaces are not required. The exceptions are:

- No spaces are permitted between the symbols
  - `'` and `:` when denoting the adverb `':`
  - `\` and `:` when denoting the adverb `\:`
  - `/` and `:` when denoting the adverb `/:`
  - a digit and `:` when denoting a verb such as `0:`
  - `:` and `:` for assignment of the form name `::` value;
- No spaces are permitted between an adverb symbol and the verb, noun or adverb symbol to its left;
- No spaces are permitted between a primitive verb symbol and a colon to its right whose purpose is to denote either assignment or the monadic case of the verb;
- No spaces are permitted between a left bracket and the character to its left. That is, index and argument notation as well as the left bracket in a conditional evaluation or control statement must immediately follow the token to its left;
- If a `/` is meant to denote the left end of a comment then it must be preceded by a blank, for otherwise it will be taken to be part of an adverb;

- Both the underscore character (`_`) and dot character (`.`) denote verbs and can also be part of a name. The default choice is part of a name. A space is therefore required between an underscore or dot and a name to its left or right when denoting a verb;
- At least one space is required between neighboring numeric constants in vector notation;
- A minus sign (`-`) denotes both a verb and part of the format of negative constants. A minus sign is part of a negative constant if it is next to a positive constant and there are no spaces between, except that a minus sign is always considered to be the verb if the token to the left is a name, a constant, a right parenthesis or a right bracket, and there is no space between that token and the minus sign. The following examples illustrate the various cases:

`x-1`	x minus 1
`x -1`	x applied to -1
`3.5-1`	3.5 minus 1
`3.5 -1`	numeric list with two elements
`x[1]-1`	`x[1]` minus 1
`(a+b)- 1`	`(a+b)` minus 1

Special Constructs

Slash, back-slash, colon and single-quote (`/ \ : '`) all have special meanings outside ordinary expressions, denoting comments, commands and debugging controls.

CHAPTER 3

TERMINOLOGY

Atoms

All data are atoms and lists composed ultimately of atoms. See Nouns in the chapter Syntax.

Atom Functions

There are several recursively-defined primitive functions, which for at least one argument apply to lists by working their way down to items of some depth, or all the way down to atoms. The ones where the recursion goes all the way down to atoms are called *atom functions*, or *atomic functions* .

A monad is said to be atomic if it applies to both atoms and lists, and in the case of a list, applies independently to every atom in the list. For example, the monad Negate, which is monadic `-` , is atomic. A result of Negate is just like its argument, except that each atom in an argument is replaced by its negation. For example:

```
  - 3 4 5                          -(5 2; 3; -8 0 2)
-3 -4 -5                         (-5 -2
                                  -3
                                  8 0 -2)
```

Negate applies to a list by applying independently to every item. Accessing the ith item of a list x is denoted by `x[i]` , and therefore the rule for how Negate applies to a list x is that the ith item of Negate x , which is `(-x)[i]` , is Negate applied to the ith item, that is `-x[i]` .

Negate can be defined recursively for lists in terms of its definition for atoms. To do so we need two language constructs. First, any function f can be applied independently to the items of a list by modifying the function with the Each adverb, as in `f'` . Secondly, the monadic primitive function denoted by `@x` is called Atom and has the value 1 when x is an atom, and 0 when x is a list. Using these constructs, Negate can be defined as follows:

```
  Negate:{:[ @ x; - x; Negate' x]}
```

That is, if x is an atom then `Negate x` is `-x`, and otherwise `Negate` is applied independently to every item of the list x. One can see from this definition that `Negate` and `Negate'` are identical. In general, this is the definition of atomic: a function `f` of any number of arguments is atomic if `f` is identical to `f'` .

A dyad f is atomic if the following rules apply (these follow from the general definition that was given just above, or can be taken on their own merit):

- `f[x;y]` is defined for atoms x and y ;
- for an atom x and a list y, the result `f[x;y]` is a list whose ith item is `f[x;y[i]]` ;
- for a list x and an atom y, the result `f[x;y]` is a list whose ith item is `f[x[i];y]` ;
- for lists x and y , the result `f[x;y]` is a list whose ith item is `f[x[i];y[i]]` .

For example, the dyad Plus is atomic.

```
  2 + 3                                2 6 + 3
5                                    5 9
  2 + 3 -8                             2 6 + 3 -8
5 -6                                 5 -2

  (2; 3 4) + ((5 6; 7 8 9); (10; 11 12))
((7 8
  9 10 11)
 (13
  15 16))
```

In the last example both arguments have count 2. The first item of the left argument, 2, is added to the first item of the right argument, `(5 6; 7 8 9)`, while the second argument of the left argument, `3 4`, is added to the second argument of the right argument, `(10; 11 12)`. When adding the first items of the two lists, the atom 2 is added to every atom in `(5 6; 7 8 9)` to give `(7 8; 9 10 11)`, and when adding the second items, 3 is added to 10 to give 13, and 4 is added to both atoms of `11 12` to give `15 16`.

Plus can be defined recursively in terms of Plus for atoms as follows:

```
  Plus:{:[(@ x) & @ y; x + y; Plus'[x;y]]}
```

The arguments of an atom function must be conformable, or else a Length Error is reported. The evaluation will also fail if the function is applied to atoms that are not in its domain. For example, `1 2 3 + (4;"a";5)` will fail because `2 + "a"` fails with a Type Error.

Atom functions are not restricted to monads and dyads. For example, the triadic function `{x+y^z}` is an atom function ("x plus y to the power z").

A function can be atomic relative to some of its arguments but not all. For example, the Index primitive `@[x;y]` is an atom function of its right argument but not its left, and is said to be *right-atomic*, or *atomic in its second argument*). That is, for every left argument x the projected monadic function `x@` is an atom function. This primitive function, like `x[y]`, selects items from x according to the atoms in y, and the result is structurally like y, except that every atom in y is replaced by the item of x that it selects. A simple example is:

```
  2 4 -23 8 7 @ (0 4 ; 2)
(2 7
 -23)
```

Index 0 selects 2 , index 4 selects 7 , and index 2 selects -23 . Note that the items of x do not have to be atoms.

It is common in descriptions of atom functions elsewhere in this manual to restrict attention to atom arguments and assume that the reader understands how the descriptions extend to list arguments.

Character Constant

A character constant is defined by entering the characters between double-quotes, as in `"abcdefg"`. If only one character is entered the constant is an atom, otherwise the constant is a list. For example, `"a"` is an atom. The notation `,"a"` is required to indicate a one character list. See Escape Sequences for entering non-graphic characters in character constants.

Character String

Character string is another name for character vector.

Character Vector

A character vector is a simple list whose items are all character atoms. When displayed in an interactive session, it appears as a string of characters surrounded by double-quotes, as in:

```
"abcdefg"
```

not as individual characters separated by semicolons and surrounded by parentheses (that is, not in list notation). When a character vector contains only one character, this is distinguished from the atomic character by prepending the Enlist monad (comma), as in `,"x"`.

Comparison Tolerance

Because floating-point values resulting from computations are usually only approximations to the true mathematical values, the Equal primitive is defined so that `x = y` is 1 (true) for two floating-point values that are either near one another or identical. To see how this works, first set the print precision so that all digits of floating-point numbers are displayed.

`\p 18`   see Print Precision in the chapter Commands

The result of the following computation is mathematically 1.0, but the computed value is different because the addend 0.001 cannot be represented exactly as a floating-point number.

```
  x: 0                    initialize x to 0
  do[1000;x+:.001]        increment x one thousand times by 0.001
  x                       the resulting x is not quite 1.000
0.9999999999999062
```

However, the expression `x = 1` has the value 1, and x is said to be *tolerantly equal* to 1:

```
  x = 1                   is x equal 1?
1                         yes
```

Moreover, two distinct floating-point values x and y for which `x = y` is 1 are said to be *tolerantly equal*. No nonzero value is tolerantly equal to 0. Formally, there is a system constant *E* called the *comparison tolerance* such that two non-zero values *a* and *b* are tolerantly equal if:

$$| a - b | \leq E \times max(| a | , | b |)$$

but in practice the implementation is an efficient approximation to this test. Note that according to this inequality, no nonzero value is tolerantly equal to 0. That is, if *a*`=0` is 1 then *a* must be 0. To see this, substitute 0 for *b* in the above inequality and it becomes:

$$| a | \leq E \times | a |$$

which, since *E* is less than 1, can hold only if *a* is 0.

In addition to Equal, comparison tolerance is used in the verbs Find, Floor, More, Less, Match, the adverbs Over and Scan for monads, and the system function `_in`.

Conformable Data Objects

The idea of conformable objects is tied to atom functions like Plus, functions like Form with behavior very much like atom functions, and functions derived from Each. For example, the primitive function Plus can be applied to vectors of the same count, as in

```
  1 2 3 + 4 5 6
5 7 9
```

but fails with a Length Error when applied to vectors that do not have the same count, such as:

```
  1 2 3 + 4 5 6 7
length error
1 2 3 + 4 5 6 7
      ^
```

The vectors `1 2 3` and `4 5 6` are said to be *conformable*, while `1 2 3` and `4 5 6 7` are not conformable.

Plus applies to conformable vectors in an item-by-item fashion. For example, `1 2 3+4 5 6` equals `(1+4),(2+5),(3+6)`, or `5 7 9`. Similarly, Plus of an atom and a list is obtained by adding the atom to each item of the list. For example, `1 2 3+5` equals `(1+5),(2+5),(3+5)`, or `6 7 8`.

If the argument lists of Plus have additional structure below the first level then Plus is applied item-by-item recursively, and for these lists to be conformable they must be conformable at every level; otherwise, a Length Error is reported. For example, the arguments in the following expression are conformable at the top level – they are both lists of count 2 – but are not conformable at every level.

```
  (1 2 3;(4;5 6 7 8)) + (10;(11 12;13 14 15))
```

Plus is applied to these arguments item-by-item, and therefore both `1 2 3+10` and `(4;5 6 7 8)+(11 12;13 14 15)` are evaluated, also item-by-item. When the latter is evaluated, `5 6 7 8+13 14 15` is evaluated in the process, and since `5 6 7 8` and `13 14 15` are not conformable, the evaluation fails.

All atoms in the arguments to Plus must be numeric, or else Plus will fail with a Type Error. However, the types of the atoms in two lists have nothing to do with conformability, which is only concerned with the lengths of various pairs of sub-lists from the two arguments.

The following function tests for conformability; its result is 1 if its arguments conform at every level, and 0 otherwise.

```
  conform:{ :[ (@x) | @y ; 1
               (#x) = #y ; &/ x conform' y; 0]]}
```

That is, atoms conform to everything, and two lists conform if they have equal counts and are item-by-item conformable (see Over Dyad in the chapter Adverbs for the meaning of the derived function `&/`).

Two objects x and y are said to *conform at the top level* if they are atoms or lists, and have the same count when both are lists. For example, if f is a dyad then the arguments of `f'` (that is, f-Each) must conform at the top level. More generally, x and y are said to *conform at the top two levels* if they conform at the top level and when both are lists, the items `x[i]` and `y[i]` also conform at the top level for every index i; and so on.

These conformability concepts are not restricted to pairs of objects. For example, three objects x, y, and z conform if all pairs `x,y` and `y,z` and `x,z` are conformable.

Console

Console refers to the source of messages to K and their responses that are typed in a K session.

Dependencies

Dependencies provide spreadsheet-like formulas within applications. A dependency is a global variable with an associated expression describing its relationship with other global variables. The expression is automatically evaluated whenever the variable is referenced and any of the global variables in the expression have changed value since the last time the variable was referenced. If evaluated, the result of the expression is the value of the variable. If not referenced, the value of this variable is the last value it received, either by ordinary specification or a previous evaluation of the dependency expression.

The dependency expression is an attribute of a global variable whose value is a character string holding the dependency expression, for example:

```
  v..d: "b + c"
```

for “v is b+c”. For example:

```
  b: 10 20 30
  c: 100
  v..d: "b + c"
  v                           v has the value b + c
110 120 130
```

```
  v[2]: 1000              v can be amended
  v
110 120 1000
  b[1]: 25                amend any part of  b or c
  v                       once again, v has the value  b + c
110 120 130
```

And of course, b and c can also be dependencies. Note that relative referents like b and c are not resolved in the attribute dictionary of v, but are entries in the same directory as v. Moreover, the dependency expression on v cannot contain an explicit reference to v itself.

Dependent Variables

If a dependency expression is defined for a variable v then v is said to be *directly dependent* on all those variables that appear in that expression and *dependent* on all those variables than can cause it to be re-evaluated when it is referenced. Not only is v dependent on all variables in its dependency expression, but on all variables in the dependency expressions of those variables, and so on.

Depth

The *depth* of a list is the number of levels of nesting. For example, an atom has depth 0, a list of atoms has depth 1, a list of lists of atoms has depth 2, and so on. The following function computes the depth of any data object:

```
  depth:{:[ @ x; 0; 1 + |/ depth' x]}
```

That is, an atom has depth 0 and a list has depth equal to 1 plus the maximum depth of its items. The symbols `|/` denote Max-Over. When applied to a list of numeric values, as in `|/  w` , the result is the largest value in w (see Over Dyad). For example:

```
  depth 10                          depth {x + y}
0                                 0
  depth 10 20                       depth (10 20;30)
1                                 2
```

Depth is a useful notion that appears in several examples elsewhere in this manual.

Dictionary

A *dictionary* is an atom that is created from a list of a special form, using the Make Dictionary verb, denoted by the dot (`.`). Each item in the list is a list of three items, the *entry*, the *value* and the *attributes*. The entry is a symbol, holding a simple name, that is, a name with no dots. The value may be any atom or list. The attributes are themselves a dictionary, giving the attributes of the item. An entry may have no attributes, or equivalently an empty dictionary (`.()`) or nil. A dictionary can be indexed by any one of its symbols, and the result is the value of the symbol. When a dictionary is a global variable it is also a directory on the K-tree, and its entries are the global variables in that directory. See Make/Unmake Dictionary and K-tree.

Dyad

A dyad (or dyadic function) is a function of two arguments. Dyadic verbs may be used in either infix or prefix notation. However, defined dyadic functions must be used in prefix notation only.

Empty List

The generic empty list has no items, has count 0, and is denoted by `()`. The empty character vector is denoted `""`, the empty integer vector `!0`, the empty floating-point vector `0#0.0`, and the empty symbol vector ``0#` ``. The distinction between `()` and the typed empty lists is relevant to certain verbs (e.g. Match) and also to formatting data on the screen.

Entry

The entries of a dictionary d are the symbols given by its enumeration, `!d`. A global dictionary is a directory on the K-tree, and its entries are the global variables in that directory.

Escape Sequence

An escape sequence is a special sequence of characters representing a character atom. An escape sequence usually has some non-graphic meaning, for example the tab character. An escape sequence can be entered in a character constant and displayed in character data. The escape sequences in K are the same as those in the C-language, but often have different meanings. As in C, the sequence `\b` denotes the backspace character, `\n` denotes the new-line character, `\t` denotes the horizontal tab character, `\"` denotes the double-quote character, and `\\` denotes the back-slash character.

In addition, `\o` and `\oo` and `\ooo` where each `o` is one of the digits from 0 through 7, denotes an octal number. If the character with that ASCII value has graphic meaning, that graphic is displayed, or if that character is one that can be specified by one of the escape sequences in the first paragraph, that sequence is displayed. For example:

```
  "\b\a\11"                  enter a character constant
 "\ba\t"                     \b displays as \b, \a as a, \11 as \t
```

Floating-Point Vector

A floating-point vector is a simple list whose items are all floating-point numbers. When displayed in a K session, it appears as a string of numbers separated by blanks, as in:

```
  10.56 3.41e10 -20.5
```

not as individual numbers separated by semicolons and surrounded by parentheses (that is, not in list notation). The empty floating-point vector is denoted `0#0.0` .

Function Atom

A function can appear in an expression as data, and not be subject to immediate evaluation when the expression is executed, in which case it is an atom. For example:

```
  f: +                     f is assigned Plus
  @ f                      f is an atom
1
  (f;102)                  f can be used like any other atom
(+;102)
```

Handle

A handle is a symbol holding the name of a global variable, which is a node in the K-tree. For example, the handle of the name `a_c..b` is `` `a_c..b ``. The term "handle" is used to point out that a global variable is directly accessed. Both of the following expressions are used to amend x:

```
  x: .[x; i; f; y]
     .[`x; i; f; y]
```

In the first, referencing x as the first argument causes its entire value to be constructed, even though only a small part may be needed. In the second, the symbol `` `x `` is used as the first argument. In this case, only the parts of x referred to by the index i will be referenced and reassigned. The second case is usually more efficient than the first, sometimes significantly so. In the case where the value of x is a directory, referencing the global variable x causes the entire dictionary value to be constructed, even though only a small part of it may be needed. Consequently, in the description of Amend, the symbol atoms holding global variable names are referred to as handles.

Homogeneous List

A homogeneous list is one whose atoms are all of the same type. For example, a character vector is a homogeneous list of depth 1. A list of integers is one whose atoms are all integers. Similarly for a list of characters, or floating-point numbers, or symbols.

Integer Vector

An integer vector is a simple list whose items are all integers. When displayed in a K session, it appears as a string of numbers separated by blanks, as in:

```
  10 20 -30 40
```

not as individual integers separated by semicolons and surrounded by parentheses (that is, not in list notation). The empty integer vector is denoted `!0` .

Item

An item is a component of a list, and may be either an atom or a list. The item of x at index position i is called the ith item and is denoted by `x[i]`.

If an item is a list then it also has items, and any of these items that are lists may have items, and so on. Items of a list are sometimes called *top-level items* to distinguish them from items of items, items of items of items, etc., which are generally referred to as *items-at-depth* . When it is necessary to be more specific, top-level items are called items at level 1 or items at depth 1, items of items are called items at level 2 or items at depth 2, and so on. Generally, an item is at depth n if it requires n indices to reach it.

There is also the related concept of *items of specified depth*, meaning items-at-depth that are a specified level above the bottom. For example, items of depth 1 would be lists of atoms within another list, as in:

```
  (1 2 3;(4 5; ("a";`bc)))
```

where the items of depth 1 are `1 2 3` and `4 5` and `("a";`bc)`. (The items at depth 1 are `1 2 3` and `(4 5;("a";`bc))` .) Generally, an item is of depth n if there is atom within it that is at depth n, but no atom at depth `n+1`.

A list may contain one or more empty items (i.e. the nil value `_n`), which are typically indicated by omission:

```
  (1 ; _n ; 2)
(1;;2)
```

K-Tree

The K-tree is the hierarchical name space containing all global variables created in a K session. The initial state of the K-tree when K is started is a working directory whose absolute path name is `.k` together with a set of other top-level directories containing various utilities. The working directory is for interactive use and is the default active, or current, directory. Each application should define its own top-level directory that serves as its logical root, using a name which will not conflict with any other top-level application or utility directories present. Every subdirectory in the K-tree is a dictionary that can be accessed like any other variable, simply by its name. The root directory has no name, but can be accessed by the expression `` .` `` ("dot back-quote").

Left-Atomic Function

A left-atomic function f is a dyad f that is atomic in its left, or first, argument. That is, for every valid right argument y, the monad `f[;y]` is atomic.

List

A list is one of the two fundamental data types, the other being the atom. The components of a list are called items (see Item). See Nouns in the chapter Syntax.

Matrix

A matrix is a rectangular list of depth 2. An integer matrix is one whose atoms are all integer atoms. Similarly for character matrix, floating-point matrix, and symbol matrix.

Monad

A monad, or monadic function, has one argument.

Nil

Nil is the value of an unspecified item in a list formed with parentheses and semi-colons. For example, nil is the item at index position 2 of ``(1 2;"abc";;`xyz)``. Nil is an atom; its value is `_n`, or `*()`. Nils have special meaning in the right argument of the primitive function Index and in the bracket form of function application.

Nilad

A nilad, or niladic function, has no arguments.

Numeric List

A numeric list is one whose atoms are either integers or floating-point numbers. For example, the arguments to Plus and Times are numeric lists.

Numeric Vector

A numeric vector is a list that is either an integer vector or a floating-point vector.

Primitive Function

A primitive function is either the dyad or monad of a simple verb, where a simple verb is one of the symbols `+` , `-` , `*` , `%` , `|` , `&` , `^` , `<` , `>` , `=` , `!` , `#` , `_`,`~` , `$` , `?` , `@` , `.` and `,` .

Rank

The rank of x is the number of items in its shape, namely `#^x` . The rank of an atom is always 0, and that of a list is always 1 or more. If the rank of a list is n, then the list must be rectangular to depth n. The rank of a matrix is 2. The rank of a dictionary d is defined to be `*^d[]`.

Rectangular List

A list of depth 2 is said to be *rectangular* if all its items are lists of the same count. For example:

```
  (1 2 3; "abc"; `x `y `z; 5.4 1.2 -3.56)
```

is a rectangular list. The shape of a rectangular list of depth 2 has two items, the first being the count of the list and the second the count of any item.

```
  ^ (1 2 3; "abc"; `x `y `z; 5.4 1.2 -3.56)
 4 3
```

Analogously, a list of depth 3 is rectangular if all items have depth 2 and all items of items are lists of the same count. The shape of a rectangular list of depth 3 has three items, the first being the count of the list, the second the count of any item, and the third the count of any item of any item. For example:

```
  ((1 2; `a `b; "AB"); ("CD"; 3 4; `c `d))
```

is a rectangular list of depth 3 and its shape is:

```
  ^ ((1 2; `a `b; "AB"); ("CD"; 3 4; `c `d))
 2 3 2
```

Rectangular lists of any depth can be defined.

It is possible for a list of depth d to be *rectangular to depth n*, where n is less than d. For example, the following list is of depth 3 and is rectangular to depth 2:

```
  ((0 1 2; `a; "AB"); ("CD"; 3 4; `c `d))
```

This list has two items, each of which has three items, but the next level of items vary in count. The shape of this list has only two items, the first being the count of the list and the second the count of any item:

```
  ^ ((0 1 2; `a; "AB"); ("CD"; 3 4; `c `d))
 2 3
```

The list x is rectangular to depth n if its shape has n items, that is if n equals `#^x` .

Right-Atomic Function

A right-atomic function f is a dyad that is atomic in its right, or second, argument. That is, for every valid left argument x, the monadic function `f[x;]` is an atom function (see Fixing Function Arguments in the chapter Functions).

Script

A script file, or *script* for short, is a source file for an application or utility. It is a text file of function definitions and statements for execution, possibly including commands to load other scripts or operating system commands (see Load and OS Command in the chapter Commands). The typical way to start an application is to give the name of its start-up script in the command that starts the K process.

Simple List

A simple list is a list whose items are all atoms, i.e. a list of depth 1 (see Depth). The atoms need not be of the same type.

Simple Vector

A simple vector is a list which is either a character vector, floating-point vector, integer vector, or symbol vector. See also Vector Notation.

String

See Character String.

String-Atomic Function

A string-atomic function f is like an atom function, except that the recursion stops at strings rather than their individual atomic characters.

String Vector

A string vector is a list whose items are all character strings.

Symbol

A symbol is an atom which holds a string of characters, much as an integer holds a string of digits. For example, `` `abc `` denotes a symbol atom. This method of forming symbols can only be used when the characters are those that can appear in names. To form symbols containing other characters, put the contents between double quotes, as in `` `"abc-345" ``.

A symbol is an atom, and as such has count 1; its count is not related to the number of characters that appear in its display. The individual characters in a symbol are not directly accessible, but symbols can be sorted and compared with other symbols. Symbols are analogous to integers and floating-point numbers, in that they are atoms but their displays may require more than one character. (If they are needed, the characters in a symbol can be accessed by converting it to a character string.)

Symbol Vector

A symbol vector is a simple list whose items are all symbols. When displayed in a K session, it appears as a string of symbols separated by blanks, as in:

```
  `a `b `x_y.z `"123"
```

not as individual symbols separated by semicolons and surrounded by parentheses (that is, not in list notation). The empty symbol vector is denoted `` 0#` `` .

Trigger

A trigger is an expression associated with a global variable that is executed immediately whenever the value of the variable is set or modified. The purpose of a trigger is to have side effects, such as setting the value of another global variable. For example, suppose that whenever the value of the global variable x changes, the new value is to be sent to another K process where it is to become the new value of the 0th item of the variable b. This trigger is set simply by placing the expression on the appropriate node of the K-tree:

```
  x..t: "pid 3: (`b; 0; :; x)"
```

where `pid` is the identifier of the other process. Note that relative referents like b are not resolved in the attribute dictionary of x, but are entries in the same directory as x.

Valence

The valence of a function is the number of its arguments. For example, the valence of a tetrad is 4, of a triad 3, of a dyad 2, of a monad 1, and of a nilad 0. A function called with the wrong number of arguments will cause a Valence Error to be reported.

Vector

A list whose items are all of the same type is called a vector of that type. Thus we have integer vectors, floating-point vectors, character vectors, symbol vectors, and string vectors.

Vector Notation

An integer or floating-point vector constant can be defined by putting the atoms next to one another with at least one space between each atom. For example, for the integer vector `1 -2 3`:

```
  # 1 -2 3                      a vector with 3 items
3
  1 -2 3[1]                     item 1 of the vector
-2
  # 3 4 5.721 1.023e10          a vector with 4 items
4
```

Note that only one item of a floating-point vector defined by vector notation has to be given in decimal or exponential notation. The other items, if whole numbers, can be given in integer format, such as the items 3 and 4 in the above floating-point vector. For example, `1 2 3.0 4` is a floating-point vector, while `1 2 3 4` is an integer vector.

Characters appear between double-quote marks for string vectors. Items in symbol vectors need not be delimited by spaces, since the back-quote character serves to distinguish them.

```
  `one`two`three            #"Kx Systems"
`one `two `three          10
```

One-item vectors employ the comma in their notation, as in:

```
  ,"a"                  ,`abc                 ,3.14159265
```

Empty vectors are denoted as `!0`, `0#0.0`, `""` and ``0#` `` for integer, floating-point, string and symbol vectors, respectively.

CHAPTER 4

VERBS

Amend Item

```
@[d; i; f; y]
@[d; i; :; y]
@[d; i; f]
```

Description

Modify the items of the list d at indices i with f and, if present, the atom or list y, and similarly for the dictionary d at entries i.

Arguments

The first argument d is either a symbol atom, dictionary, or any list, and the second argument i is either nonnegative integer or symbolic. The third argument f is any monadic or dyadic function; the first of the above expressions corresponds to dyadic f and the third to monadic f. The argument y, if present, is any atom or list for which i and y are conformable, and where items-at-depth in y corresponding to atoms in i must be valid right arguments of f.

Definition

If the first argument is a symbol atom then it must be a handle, and the definition proceeds as if the value of the global variable named in the symbol is used as the first argument (but see Handle in the chapter Terminology). In addition, the modified value becomes the new value of the global variable, and the symbol is the result. The first argument is assumed to be a dictionary or list for the remainder of this definition.

The description that follows starts with the case of dyadic f. The second of the above expressions can be viewed as a special case of the first expression, where the dyadic function represented by the colon simply returns its right argument, i.e. `{[x;y] y}`. The purpose of the first expression is to modify items of d selected by i with values of f applied to those items as left argument and items-at-depth in y as right argument. The second expression simply replaces those items with items-at-depth in y. The third expression, where there is no y and f is monadic, replaces each of those items with the values of f applied to it.

In the case of a left argument list d, Amend Item permits modification of one or more items of that list by a function f and, when f is dyadic, items-at-depth in y. The result is a copy of d with those modifications. For example:

```
  d:9 8 7 6 5 4 3 2 1 0
  i:2 7 1 8 2 8
  y:5 3 6 2 7 4
  @[d; i; +; y]
9 14 19 6 5 4 3 5 7 0
```

This result is developed as follows. Starting at index 0 of i, item `d[i[0]]` is replaced with `d[i[0]]+y[0]`, i.e. `d[2]` becomes `7+5`, or 12. Then `d[i[1]]` is replaced with `d[i[1]]+y[1]`, i.e. `d[7]` becomes `2+3`, or 5. Continuing in this manner, `d[1]` becomes `8+6`, or 14, `d[8]` becomes `1+2`, or 3, `d[2]` becomes `12+7`, or 19 (modifying the previously modified value 12), and `d[8]` becomes `3+4`, or 7 (modifying the previously modified value 3).

In general, i can be any atom or list whose atoms are valid indices of d, i.e. from the list `!#d`, and i and y must be conformable. However, the function is not an atom function. Instead, the function proceeds recursively through i and y as if they were the arguments of a dyadic atom function, but whenever an atom of i is encountered, say k, the current value of `d[k]` is replaced by `f[d[k];z]`, where z is the item-at-depth in y that was arrived at the same time as k. The result is the modified list. For example:

```
  d: 9 8 7
  i: (0; (1;2 2))
  y: ("abc"
      ((1.5; `xyz)
       (100; (3.76; `efgh))))
```

Before evaluating Amend Item for this data, compare the structures of i and y to see that the 0 in i goes with `"abc"` in y, the 1 in i goes with `(1.5;`xyz)` in y, the first 2 of `2 2` in i goes with 100 in y, and the second 2 of `2 2` in i goes with `(3.76;`efgh)` in y. Now:

```
  @[d; i; ,; y]          f is Join
((9;"a";"b";"c")         Join 9 and "abc"
 (8;1.5;`xyz)            Join 8 and (1.5;`xyz)
 (7;100;3.76;`efgh))     Join 7 and 100, then join with (3.76;`efgh)
```

The general case of `@[d; i; f; y]`

The general case of Amend Item for dyadic f can be defined recursively as follows, based on the definition for an atom second argument:

```
  AmendItem:{[d;i;f;y] :[ @ i; @[d; i; f; y]
                        AmendItem/[d; i; f; y]]]}
```

Note the application of Over to AmendItem, which requires that whenever i is not an atom, either y is an atom or `#i` equals `#y`. Over is used to accumulate all changes in the first argument d.

The case of `@[d; i; :; y]`

The second case simply replaces the items of d with items-at-depth in y. In effect, the dyadic case for the function that simply returns its right argument as its result, i.e. `{[x;y] y}`. For example:

```
  d:9 8 7 6 5 4 3 2 1 0
  i:2 7 1 8 2 8
  y:50 30 60 20 70 40
  @[d;i;:;y]
9 60 70 6 5 4 3 30 40 0
```

This result is developed as follows. Starting at index 0 of i, item `d[i[0]]` is replaced with `y[0]`, i.e. `d[2]` becomes 50. Then `d[i[1]]` is replaced with `y[1]`, i.e. `d[7]` becomes 30. Continuing in this manner, `d[1]` becomes 60, `d[8]` becomes 20, `d[2]` becomes 70 (overwriting the previous change to 50), and `d[8]` becomes 40 (overwriting the previous change to 20).

The case of `@[d; i; f]`

The third case of Amend Item, for a monad f, is similar to the case for dyadic f, but simpler. As the function proceeds recursively through i, whenever an atom of i is encountered, say k, the current value of `d[k]` is replaced by `f[d[k]]`. The result is the modified list. As in the dyadic case, if an index k of d appears more than once in i, then `d[k]` will be modified more than once.

Facts About Amend Item

If an index of d appears more than once in i, then that item of d will be modified more than once. The above definition in terms of Over gives the correct order in which the replacements are made.

The function f is applied separately for each atom in i.

The cases of Amend Item with a function f are sometimes called Accumulate Item because the new items-at-depth are computed in terms of the old, as in `@[x;2 6;+;1]`, where items 2 and 6 are incremented by 1.

Error Reports

Domain Error if the symbol d is not a handle, i.e. does not hold the name of an existing global variable.

Index Error if any atom of the right argument is not a valid index of the left argument.

Length Error if the second argument i and the last argument y are not conformable.

Rank Error if the object being modified is a non-dictionary atom.

Type Error if any atom of i is not an integer, symbol or nil.

Amend

```
.[d; i; f; y]
.[d; i; :; y]
.[d; i; f]
```

Description

Modify the items-at-depth in list d at indices i with f and, if present, the atom or list y, and similarly for items-at-depth at indices `1_ i` in the dictionary entries `*i` .

Arguments

The first argument d is either a dictionary, symbol atom, or any list. Each atom of the second argument i is either nonnegative integer or symbolic. The special case of an atom d other than a dictionary or symbol together with the empty list i is permitted. The relationships between d and i are the same as for Index.

The third argument f is any monad or dyad; the first of the above expressions corresponds to dyadic f and the third to monadic f. The argument y, if present, is any atom or list; i and y must be conformable in a sense described below, and items-at-depth in y corresponding to paths in i must be valid right arguments of f.

Definition

In the case of a left argument list d, Amend modifies one or more items-at-depth in that list by a function f. The result is a copy of d with those modifications.

If the left argument is a symbol atom then it must be a handle, and the definition proceeds as if the value of the global variable named in the symbol is used as the left argument (but see Handle in the chapter Terminology). In addition, the modified value becomes the new value of the global variable named in the handle, and the symbol is the Amend result.

If i is a nonnegative integer atom then the ith item of d is amended. If i is a symbol atom then d must be a dictionary or handle of a directory and the ith entry is amended. If d is an atom other than a dictionary or symbol then i must be the empty list, and d is amended. If d is a list and i is nil then all of d is amended, but one item at a time, as if i was `!#d`.

The remainder of this section assumes that both d and i are non-empty lists. It is also assumed that the reader is familiar with the definition of Index.

The description that follows starts with the case of dyadic f. The expression above with the colon in third position can be viewed as a special case of the first expression, where the dyadic function represented by the colon simply returns its right argument, as in `{[x;y] y}`. The items-at-depth in d that are to be replaced are selected by i just like in Index, i.e. `d.i` . For monadic f, each selected item-at-depth is replaced by the result of f applied to that item. If y is present the selection proceeds as before, but with `d.i` and y together, as if they were the arguments of a dyadic atomic function. When the selection of an item-at-depth in d is about to be made, we have also arrived at an item-of-depth in y. In the expression above with the colon, the item-at-depth in d is simply replaced by that item-at-depth in y. In the case of dyadic f, the item-at-depth in d is replaced by the result of applying f with the item-at-depth in d as the left argument and the item-at-depth in y as the right argument.

The case where i is a non-negative integer vector (a single path)

If the second argument i is a nonnegative integer vector then a single item at depth `#i` in d is replaced. For example:

```
  (5 2.14; "abc") . 1 2                 Index to select "c"
"c"                                     selected item-at-depth
  .[(5 2.14; "abc"); 1 2; ,; "xyz"]     append "xyz" to "c"
(5 2.14                                 the result is the amended list
 ("a"
  "b"
  "cxyz"))                              amended item-at-depth
```

The case where the items of i are non-negative integer vectors

The following is an example of cross-sectional amending, and can be reduced to a series of single path amends like the first case above:

```
  d:((1 2 3; 4 5 6 7)
     (8 9; 10; 11 12)
     (13 14; 15 16 17 18; 19 20))
  i:(2 0; 0 1 0)
  y:(100 200 300; 400 500 600)
  r:.[d; i; ,; y]
```

The following display of d adjacent to r provides easy comparison:

```
  d                          r
((1 2 3                    ((1 2 3 400 600
  4 5 6 7)                   4 5 6 7 500)
 (8 9                       (8 9
  10                         10
  11 12)                     11 12)
 (13 14                     (13 14 100 300
  15 16 17 18                15 16 17 18 200
  19 20))                    19 20))
```

The shape of y is `2 3`, the same shape as the cross-section selected by `d . i`. The `(j;k)`th item of y corresponds to the path `(i[0;j];i[1;k])`. The first single path Amend is equivalent to:

```
  d: .[d; (i . 0 0; i . 1 0); ,; y . 0 0]
```

(since the amends are being done individually, and the assignment serves to capture the individual results as we go), or:

```
  d: .[d; 2 0; ,; 100]
```

and item `d . 2 0` becomes `13 14,100`, or `13 14 100`. The next single path Amend is:

```
  d: .[d; (i . 0 0; i . 1 1); ,; y . 0 1]
```

or

```
  d: .[d; 2 1; ,; 200]
```

and item `d . 2 1` becomes `15 16 17 18 200`. Continuing in this manner, item `d . 2 0` becomes `13 14 100 300` (modifying the previously modified value `13 14 100`); item `d . 0 0` becomes `1 2 3 400`; item `d . 0 1` becomes `4 5 6 7 500`; and, finally, item `d . 0 0` becomes `1 2 3 400 600` (modifying the previously modified value `1 2 3 400`).

The case of **`.[d; i; :; y]`**

The second case of Amend, where the colon appears in place of the dyadic function f, simply replaces the items-at-depth in d with items-at-depth in y. Repeating the earlier example with colon in place of Join:

```
d:((1 2 3; 4 5 6 7)
   (8 9; 10; 11 12)
   (13 14; 15 16 17 18; 19 20))
i:(2 0; 0 1 0)
y:(100 200 300; 400 500 600)
r:.[d; i; :; y]
```

The following display of d next to r provides easy comparison:

```
  d                  r
((1 2 3            ((600               replaced twice
  4 5 6 7)           500)              replaced once
 (8 9               (8 9
  10                 10
  11 12)             11 12)
 (13 14             (300               replaced twice
  15 16 17 18        200               replaced once
  19 20))            19 20))
```

Note that there are multiple replacements of some items-at-depth in d, corresponding to the multiple updates in the earlier example.

The case of `.[d; i; f]`

The third case of Amend, for a monadic function f, replaces the items-at-depth in d with the results of applying f to them. Repeating the earlier example with Negate in place of Join:

```
d:((1 2 3; 4 5 6 7)
   (8 9; 10; 11 12)
   (13 14; 15 16 17 18; 19 20))
i:(2 0; 0 1 0)
y:(100 200 300; 400 500 600)
r:.[d; i; -:]
```

The following display of d next to r provides easy comparison:

```
  d                    r
((1 2 3              ((1 2 3               negated twice
  4 5 6 7)             -4 -5 -6 -7)        negated once
 (8 9                 (8 9
  10                   10
  11 12)               11 12)
 (13 14               (13 14               negated twice
  15 16 17 18          -15 -16 -17 -18     negated once
  19 20))              19 20))
```

Note that there are multiple applications of f to some items-at-depth in d, corresponding to the multiple updates in the first example.

The general case

In general, the argument i can be any atom that is a valid index of d, i.e. one of `!#d`, or a list representing paths to items at depth `#i` in d. The function proceeds recursively through `i[0]` and y as if they were the arguments of a dyadic atom function, except that when arriving at an atom in `i[0]`, that value is retained as the first item in a path and the recursion continues on with `i[1]` and the item-at-depth in y that had been arrived at the same time as the atom in `i[0]`. And so on until arriving at an atom in the last item of i. At that point a path p into d has been created and the item at depth `#i` selected by p, namely `d . p`, is replaced by `f[d . p;z]` for dyadic f or `f[d . p]` for monadic f, where z is the item-at-depth in y that had been arrived at the same time as the atom in the last item of i.

The general case for dyadic f can be defined recursively by partitioning the index list into its first item and the rest:

```
  Amend:{[d;F;R;f;y] :[ _n ~ F; Amend[d; !#d; R; f; y]
                       0 = # R; @[d; F; f; y]
                           @ F; Amend[d @ F; *R; 1_R; f; y]
                                Amend[;; R;;]/[d; F; f; y]}
```

Note the application of Over to Amend, which requires that whenever F is not an atom, either y is an atom or `#F` equals `#y`. Over is used to accumulate all changes in the first argument d.

Facts About Amend

In the general case of a one-item list i, `.[d;i;f;y]` is identical to `@[d;*i;f;y]` and `.[d;i;f]` is identical to `@[d;*i;f]`.

In the case of a non-empty index list i, the function f is applied once for every path generated from i, just as the above definition indicates. However, if the index i is the empty list, i.e. `()`, then Amend is “Amend Entire”. That is, the entire value in d is replaced, in the first case `.[d;();f;y]` with `f[d;y]`, in the second case `.[d;();:;y]` with y, as in `d:y`, and in the third case `.[d;();f]` with `f[d]`. For example:

```
  .[2 3; (); ,; 4 5 6]
2 3 4 5 6
```

On the other hand, if i is the enlist of nil, then according to the above definition Amend is “Amend Each”. That is, every item of d is modified by separate applications of f in the first and third cases of Amend, and by the corresponding items of y in the second case. For example:

```
  .[2 3; ,_n; ,; 4 5]
(2 4
 3 5)
```

The cases of Amend with a function f are sometimes called Accumulate because the new items-at-depth are computed in terms of the old, as in `.[x; 2 6; +; 1]`, where item 6 of item 2 is incremented by 1.

Error Reports

Domain Error if the symbol d is not a handle, i.e. does not hold the name of an existing global variable.

Index Error if any path in the index list i is not a valid path of the first argument.

Length Error in the cases of dyadic f and : if i and y are not conformable as described above.

Type Error if any atom of i is not an integer or symbol or nil.

Apply (Monadic)

```
f @ x
```

Arguments

The left argument f is any monadic function or atom symbol holding the global name of a monadic function, and the right argument is any argument of f.

Definition

Apply (Monadic) applies the function f to the argument x. For example:

```
  { x ^ 2 } @ 3
9.0
```

If f is a symbol atom then it must hold the name of a global variable whose value is a monadic function, and that function is applied.

Error Trap (Monadic)

Error Trap (Monadic) is denoted `@[f; x; :]` and defined in the same way as Error Trap under Apply.

Facts about Apply (Monadic)

`f@x` is identical to `f . ,x`.

Error Reports

Type Error if the symbol d is not a handle, i.e. does not hold the name of an existing global variable whose value is a monadic function.

Valence Error if f is niladic.

Apply

`f . x`

Arguments

The left argument f is any function or atom symbol holding the name of a function, and the right argument x is an argument list of f.

Definition

Apply applies the left argument function f to the argument list x. If f has 1 argument then x has 1 item and f is applied to the argument `x[0]`. If f has 2 arguments then x has 2 items and f is applied to the first argument `x[0]` and the second argument `x[1]`, and so on. If the left argument is a symbol atom then it must hold the name of a function, such as `` `g``, and that function – g in this example – is applied to the argument list.

Assume f is a function, not a symbol. In terms of the bracket-semicolon notation for function application (see Apply in the chapter Amend, Index, Apply and Assign), `f . x` is identical to:

```
  f[x[0]; x[1]; …; x[-1+#x]]
```

for functions with valence at least 2, and

```
  f[x[0]]
```

for monadic functions. See below for niladic functions.

If f is a symbol atom then it must hold the name of a global variable whose value is a function, and that function is applied.

Niladic Functions

Niladic functions are handled differently. The pattern above suggests that the empty list `()` is argument list to niladic f, but `f . ()` is a case of Index, and the result is simply f. Niladic Apply is denoted by `f . ,_n`, i.e. the right argument is the enlisted nil. For example:

```
  a: 2 3
  b: 10 20
  {a + b} . ,_n
12 23
```

Error Trap

When debugging an application it is helpful for execution to be suspended when an error in a function occurs so that the problem can be analyzed (see the chapter Controls and Debugging for discussions of suspended execution). However, in a production application, rather than suspending execution, it is often preferable to log the error and either continue on or exit the application, whichever is appropriate. This behavior can be obtained with a variant of Apply known as Error Trap.

For a function of arbitrary valence, Error Trap is denoted by `.[f; x; :]` and always produces a two-item list. If `f . x` evaluates successfully then `.[f; x; :]` is identical to `(0;f . x)`, i.e. 0 indicating success followed by the result of applying f to x. However, if the evaluation of `f . x` fails then `.[f; x; :]` is `(1;"text")`, i.e. 1 indicating failure followed by a character vector holding the text of the error message that would ordinarily be displayed in the session log. For example:

```
  .[%; (3;4); :]
(0;0.75)
  .[%; (3;0); :]
(0;0i)
  .[=; 0; :]
(1;"valence")
```

Facts About Apply

If f is a monadic function, then `f . ,y` is identical to `f@y` , i.e. Apply (Monadic).

Error Reports

Type Error if the symbol d is not a handle, i.e. does not hold the name of an existing global variable whose value is a function.

Valence Error if the count of x is greater than the valence of f.

Atom

`@ x`

Argument

Any atom or list.

Definition

Atom applies to any atom or list, returning 1 if the object is an atom, and 0 if the object is a list.

```
  @ 1
1
  @ 2 3                  this is a 2-item vector
0
  @ "Z"
1
  @ ""                   this is the empty character vector
0
  @ {x+y}                functions are atomic
1
  @ (+;-)                this is a 2-item function list
0
  @ `symbol
1
  @ .((`a;2);(`b;3))     this is a dictionary with 2 entries, `a and `b
1
  @ _n                   nil is an atom
1
```

Count

```
# x
```

Argument

Any atom or list.

Definition

Count yields the number of items in a list argument. The result is 1 for an atom.

```
  # 3 1 4 2
4
  ms:(8 1 6;3 7;4 9 2)
  # ms
3
  # "A"                      "A" is a character atom
1
  # ,"A"                     a list with one item
1
  # "count"                  another character vector
5
```

Divide

`x % y`

Arguments

The arguments x and y are conformable numeric atoms or lists.

Definition

Divide is an atom function that produces x divided by y for atoms x and y. Integer atoms are treated like floating-point atoms, and the result is always floating-point.

There is one difference from the common mathematical definition: `0%0` is defined to be 0.0.

The result is 0.0 if the mathematical result would be too small in magnitude to be represented as a floating-point number. The result could also be `0i` or `-0i`, meaning plus or minus infinity respectively, in the appropriate cases.

Error Reports

Length Error if the arguments are not conformable.

Type Error if either argument is not numeric.

Drop / Cut

`x _ y`

Arguments

The left argument is an integer atom in the case of Drop or an integer vector in the case of Cut. The right argument is an atom or list for Drop, or a list for Cut.

Definition

Drop is the case of `x _ y` where x is an integer atom. The result of `x _ y` is to drop the first x items of y if x is positive, or the last `-x` items of y if x is negative.

```
  1 _ "stares"                 -2 _ "stares"
"tares"                      "star"
```

If y is an empty list or an atom, Drop is an identity function. If x is not less than the count of y, the result is the appropriate empty list.

```
  0 _ 7               88 _ ""               9 _ !6
7                   ""                    !0
```

Cut is the case of `x _ y` where x is an integer vector. The items of x must be indices to items of list y, in non-descending order. The effect of `x _ y` is to partition y into sub-lists beginning at indices x.

```
  0 3 _ 0 1 2 3 4 5            a:"try to cut into words"
(0 1 2                         m: & a = " "
 3 4 5)                        m
  0 4 _ 0 1 2 3 4 5          3 6 10 15
(0 1 2 3                       (0,m) _ a
 4 5)                        ("try"
  0 4 _ "seashells"           " to"
("seas"                       " cut"
 "hells")                     " into")
                              " words")
```

Duplicates in x result in empty lists among the items of the result:

```
  1 1 3 _ !6
(!0
 1 2
 3 4 5)
```

The result always has the same number of items as the left argument.

Error Reports

Domain Error if the left argument is an integer vector but not in non-descending order, or it contains any negative integers.

Int Error if the left argument is not an integer atom or vector, or if the left argument is a vector and the right is an atom.

Length Error if the left argument is a vector containing invalid indices.

Enlist

```
, x
```

Argument

Any atom or list x.

Definition

Enlist creates a one-item list containing x. The count of `,x` is always 1, the first item of `,x` is identical to x, and the shape of `,x` is 1 followed by the shape of x.

```
  x: 1 2 3
  ^ x                     the shape of x
,3
  y: , x
  ^ y                     the shape of  ,x
1 3                       1 followed by the shape of x
  x ~ * y                 Does x match the First of  ,x  ?
1                         Yes.
```

Enumerate

```
! x
```

Argument

Nonnegative integer atom, symbol atom, character atom or string, or dictionary x.

Definition

For a nonnegative integer atom x, Enumerate produces a list of the x integers from 0 through `x-1` in increasing order.

```
  ! 10                           ! # (1 2; "abc"; `xwz)
0 1 2 3 4 5 6 7 8 9            0 1 2
```

These integers are indices into lists of length x. For example:

```
  list: "abcdefghij"
  # list
10
  list[!10]
"abcdefghij"
```

If a symbol x is the handle of a directory on the K-tree, then `!x` is a symbol vector whose items are the entries in that directory. The result is nil for any other symbol argument. Similarly for a dictionary x, the items of the symbol vector `!x` are the entries in x.

The case of a character atom or string x is analogous to the symbol atom case. If x holds the name of a directory in the host operating system, then `!x` is a string vector whose entries hold the names of the entries in that directory. The result is nil for any other character atom or string argument.

Error Reports

Domain Error if x is not a nonnegative integer atom, symbol atom, dictionary or character string.

Equal

```
x = y
```

Arguments

The arguments x and y are conformable atoms or lists. When both are atoms they must both be numeric, or both characters, or both symbols.

Definition

Equal is an atom function, and for atoms x and y, the result of `x=y` is:

- 1 if x and y are numeric, and x is equal to y in the usual mathematical sense, and 0 otherwise;
- 1 if x and y are the same character, and 0 otherwise;
- 1 if x and y are the same symbol, and 0 otherwise.

```
  3 = 3                     "cat" =  "rat"
1                         0 1 1
  3 = -3                    `abc = `abcdefg
0                         0
```

Comparison tolerance is used when both x and y are numeric and at least one is floating-point. That is, `x=y` is 1 if x and y are close in value, even if they are actually distinct.

```
  3.0 = 3.1
0
  3.0 = 3.000000000001
0
  3.0 = 3.0000000000001
1
```

Error Reports

Length Error if the arguments are not conformable.

Type Error if applied to atoms that are not both numeric, or both characters, or both symbols.

Find

```
d ? y
```

Arguments

The left argument d can be any list or nil, and the right argument y can be any atom or list.

Definition

If y occurs among the items of d then `d?y` is the smallest index of all occurrences. Otherwise, `d?y` is `#d` (the smallest nonnegative integer that is not a valid index of d). When d is nil, the result is y.

```
  9 8 7 6 5 4 3 ? 7
2                               7 is found
  9 8 7 6 5 4 3 ? 1
7                               1 is not found
  ms: (8 1 6 ; "abcdef" ; 4 9 2 ; `x `y `z `w ; 4 9 2)
  ms ? 4 9 2
2                               the first  4 9 2  is found
  words: ("canoe"
          "tug"
          "raft"
          "rowboat"
          "ark"
          "liner")
  words ? "raft"
2                               "raft"  is found
  words ? "submarine"
6                               "submarine"  is not found
```

Find uses Match for comparing items of d and y, and therefore comparisons of numeric objects when at least one is floating-point are based on comparison tolerance.

Error Reports

Domain Error if d is atomic.

First

```
* x
```

Argument

Any list or atom x.

Definition

The result of First is the first item of list x, or x itself if x is an atom..

```
  * "abc"
"a"
  * ("abc"; "defg"; "hijkl")
"abc"
  * ,,`pqr
,`pqr
  * `pqr
`pqr
```

First is also defined for empty vectors and the empty list. For each of the empty vectors, the result is a suitable prototype for the left argument of Form:

```
  *!0           *0#0.0        *0#""         *0#`
0             0.0           " "           `

  *()
```

(the result of *() is nil)

The last item of a list is obtained by the K idiom `*|x` called “First Reverse”, as in:

```
  * | ("abc"; "defg"; "hijkl")
"hijkl"
```

Flip

```
+ x
```

Arguments

Any list for which all its list items have the same count, or any atom.

Definition

Flip applies to a list having a shape of length two or more, and interchanges the top two levels of the list. Mathematically speaking, flip is matrix transpose. If the top two levels of x have shape p and q, the top two levels of `+ x` will have shape q and p.

```
  m: 3 4 # ! 12
  m                           + m
(0 1 2 3                    (0 4 8
 4 5 6 7                     1 5 9
 8 9 10 11)                  2 6 10
                             3 7 11)
  ^ m                         ^ + m
3 4                         4 3

  k: ((0 1 2;3);(4 5;6 7 8 9);(10 11;12))
  k                           + k
((0 1 2      item (0;0)     ((0 1 2      item (0;0)
  3)         item (0;1)       4 5        item (0;1)
 (4 5        item (1;0)       10 11)     item (0;2)
  6 7 8 9)   item (1;1)      (3          item (1;0)
  (10 11     item (2;0)       6 7 8 9    item (1;1)
  12))       item (2;1)       12))       item (1;2)
  ^ k                         ^ + k
3 2                         2 3
```

Atoms are permitted among the items of x, so long as all list items have the same count. The result is as if each atom was first replicated n times, where n is the count of list items.

```
  a: (1 2 3                   b: (1 2 3
      "C"                         "CCC"
      `x `y `z)                   `x `y `z)
  + a                        + b
((1;"C";`x)                ((1;"C";`x)
 (2;"C";`y)                 (2;"C";`y)
 (3;"C";`z))                (3;"C";`z))
```

Flip is an identity function for atoms:

```
  + `abc             + 67               +{x*y}
`abc               67                 {x*y}
```

If all items of x are atoms then `+ x` is identical to x.

```
  + `a`b`c`d
`a `b `c `d
```

Error Reports

Length Error if the counts of the list items are not all equal.

Floor

```
_ x
```

Argument The argument is any numeric atom or list.

Definition

Floor is an atom function. Its definition depends on the greatest and least representable integers for the computer on which K is running. For those computers with 32-bit integers, these values are:

```
G: 2147483647         i.e., _ -1+2^31
S:-2147483648         i.e., _ - 2^31
```

Floor is defined for all numeric values which are greater than or equal to S, and less than G. The value of `_ x` for such a numeric atom x is the largest representable integer that is not greater than x. For example:

```
  _ 4.6                     _ -4.6
4                         -5
```

Comparison tolerance is used when floating-point arguments are near integer values. Namely, if a floating-point argument that is tolerantly equal to but actually less than an integer, then its floor is that integer, not that integer Minus 1. In the following, the decimal number in the first row is not tolerantly equal to 2, but the one in the second row is:

```
  2 = 1.999999999999          _ 1.999999999999
0                         1
  2 = 1.9999999999999         _ 1.9999999999999
1                         2
```

Compare this primitive function with the system function `_floor`, whose result is also the integer part of its argument, but as a floating-point number and without using comparison tolerance.

Error Reports

Domain Error if a numeric argument is either less than the least representable integer or greater than or equal to the greatest representable integer.

Type Error if the argument is not numeric.

Format

`$ x`

Arguments

Any atom or list.

Definition

Format is an atom function. The result `$ x` is like x, except that every atom in x is replaced with its character vector representation. For example:

```
  $ 0 9 7 6
(,"0"
 ,"9"
 ,"7"
 ,"6")
  $ "1234"
"1234"
  $ {[x;y] x+y}
"{[x;y] x+y}"
  $ 1.2e-34
"1.2e-34"
```

The results for floating-point numbers depend on the print precision setting (see Print Precision in the chapter Commands).

```
  \p 4                  set print precision
  $ 1.2345678
"1.235"                 only 4 digits in the result
  \p 6
  $ 1.2345678
"1.23457"               now there are 6 digits in the result
  \p 10
  $ 1.2345678
"1.2345678"             now there are 8 digits (no padding to 10)
```

Format (Dyadic)

```
x $ y
```

Arguments

The arguments are conformable atoms and lists; the left argument is numeric and the right argument consists of integers, floating-point numbers, and symbols.

Definition

Format is an atom function, whose result for an atom left argument and an atom right argument is always a character vector.

Integer left argument

The right argument may be integer, floating-point, symbol, or string. The effect is as follows: Apply `$` to the atom y (see Format) to produce an intermediate result t. Then if the integer x is positive, start at the right-most character in t and, moving to the left, select x characters from t. If x is greater than `#t`, select all of t and append to the left with `x-#t` blanks. For example:

```
  2 $ 2.345                      7 $ `abcd
"45"                           "   abcd"
```

If x is negative, start at the left-most character of t and move to the right, appending with blanks on the right if necessary.

Floating-point left argument

The right argument must be numeric, i.e. either integer or floating-point. Let n be the first decimal digit in x, e.g. n is 2 if x is 5.27. Apply `$y` to the atom y (see Format) to produce an intermediate result t. If t has more than n decimal digits, it is rounded to n decimal digits. If it has fewer, say m, then `n-m` zeros are appended to the right. If y is an integer then a decimal point and n zeros are appended to the right. Finally, use the integer portion of x to select from or extend this intermediate result in the same manner as the integer left argument case. For example:

```
  7.2 $ 2.345                    7.2 $ 714
"   2.35"                      " 714.00"
```

Negative x, in addition to causing selection from left to right and padding on the right, specifies exponential format. For example:

```
  -9.2 $ 2.345                  -9.2 $ 714
"2.35e+00 "                   "7.14e+02 "
```

Error Reports

Domain Error when an atom left argument and atom right argument are not one of the combinations listed in the definition, except for the type error case below.

Length Error if the arguments are not conformable.

Type Error in Format for a floating-point left argument and symbol right argument.

Form

`x $ y`

Arguments

The arguments are conformable atoms and lists. Any atom in the left argument must be one of the five prototypes `0`, `0.0`, `` ` ``, `" "`, and `{}`. The right argument is a character atom or list.

Definition

Form converts from character type to K data types. The left argument specifies the K type of the result, while the value of the result comes from the right argument.

When the left argument is an atom it must be one of the five K prototypes given above. The definition of Form for these five cases are as follows:

Case	Definition of x $ y
`0`	The right argument consists of a sequence of characters that form a valid integer representation. The result is that integer. For example, `0$"27"` is 27.
`0.0`	The right argument contains a sequence of characters that form a valid integer or floating-point number representation. The result is that floating-point number. For example, `0.0$"3.4"` is `3.4` and `0.0$"27"` is `27.0`.
`" "`	`" "$y` equals y for all character vectors y.
`` ` ``	`` `$y `` is the character string with the same contents as the character vector y. For example, `` `$"abc" `` is `` `abc ``.
`{}`	The right argument is any character vector whose contents are a valid K expression, and the result of `{}$y` is the result of evaluating that expression.

Form is like an atom function. However, unlike an atom function, as Form recursively descends through its arguments, it evaluates a result a `$v` whenever it arrives at an atom t from the left argument and character vector or atom v from the right argument; it does not continue the descent into a character vector v to evaluate the results a `$v[i]` for the character atoms in v. For example:

```
  ` $ "abc"
`abc                    not `a `b `c
  (0; (0.0; `)) $ ("23"; ("23.4"; "abc"))
(23
 (23.4;`abc))
```

That is, Form can be defined recursively in terms of an atom left argument and vector or atom right argument, as follows:

```
  Form:{ if[ (@x) & ~ 1 < depth y; x $ y
                   ~ 1 < depth y; x Form\: y
                                  x Form' y]}
```

Error Reports

Domain Error when an atom left argument and a vector right argument are not one of the combinations listed in the definition.

Length Error if the arguments are not conformable.

Function Inverse

```
f ? y
?[f; y; x]
```

Description

Evaluation of the inverse function of the monadic function f. That is, if x equals `f ? y`, then y equals `f @ x`.

Arguments

The argument f is a monadic numeric function. The argument y is a valid result of the function f, i.e. there is an argument a for which y equals `f @ a`.

Definition

The system functions `_exp` and `_log` are inverse functions of one another, in that if y equals `_exp x` then x equals `_log y`. That is, `_log y` equals `(_exp)?y`. For example:

```
  (_exp) ? 2
0.6931472
  _log 2
0.6931472
```

Actually, `f?y` uses an approximation method and produces a result that is accurate only to within a specified tolerance, which means that `(_exp)?y` and x `_log y` differ by at most that tolerance.

In the next example, the function `f@x` does not have an inverse function for all x, but instead has one inverse function for x less than 0.5, and another for x greater than 0.5.

```
  f:{(x^2)+(-x)+(-1)}
  f ? 0
1.618034
  f 1.618034                    the answer, which should be close to 0
2.5156e-08
```

The approximation method uses default initial approximations to the result which, in this example and others like it, has the effect of choosing one of the two or more inverse functions to be computed. The triadic form `?[f; y; x]` must be used to compute the other inverse functions. At the least, the initial approximations must in the range of the particular inverse function of interest, which can be accomplished with the triadic form by properly specifying the third argument x. In this simple example it is enough to choose any value less than 0.5 as the third argument. For example:

```
  ?[f; 0; 0.25]
-0.618034                    another solution y of  y = f 0
  f -0.618034                a check of the answer, which should be close to 0
2.5156e-08
```

The triadic form must also be used when the default initial approximation is not close enough to the result, which can happen for functions with more complicated graphs than those in the above examples.

The approximation method used in Function Inverse is the secant method, with up to 20 iterations. The default initial approximations are 0.9999 and 0.9998, which represent annual spreads of 10 and 20 basis points (0.01%) in financial models where the result of the function f is a discount rate. The tolerance applied is 1e-6 times the magnitude of the argument y, that is 1/100 of a basis point. For the triadic case `?[f; y; x]`, the initial approximations are x and `0.9999*x`.

In general, the function f should be a monadic atom function with numeric arguments and results, and then both `?[f;]` and `?[f;;]` are atom functions.

Since `?` is a primitive verb, in situations where it cannot be determined from context whether it represents a monadic or dyadic function, the dyadic case is assumed. This assumption is not strict, however. For example, if `f:?` then f is dyadic, but it can also be given three arguments to evaluate the other case of Function Inverse. The reason that ambivalence is allowed here is that the functionality of the two cases of Function Inverse is essentially the same.

Error Reports

Limit Error if the result is not produced in 20 iterations.

Grade Down

`> x`

Description

A permutation of `!#x` that sorts the items of x in non-ascending order.

Argument and Result

The argument x is any list. The result is a nonnegative integer vector with the same count as x.

Definition

The result of `>x` is a permutation of `!#x` for which the items of `x[>x]` are in non-ascending order. For example:

```
  > 3 1 4 2
2 0 3 1                  permutation vector
  3 1 4 2[2 0 3 1]
4 3 2 1                  reordering of 3 1 4 2 by the permutation vector
```

```
  ms : (8 1 6 ; 3 5 7 ; 3 9 2)
  >ms
0 2 1                    permutation vector
  ms[> ms]
(8 1 6                   8 comes before 3
 3 9 2                   the 3's are a tie, then 9 comes before 5
 3 5 7)
```

```
  "dozen"[> "dozen"]
"zoned"
```

When the items of x are distinct they can be rearranged in descending order and there is only one permutation that will produce this arrangement. However, there is always more than one when x has duplicate items, because while the indices of duplicates must be grouped together in the permutation, they can arranged in any way within the group. For example, `"xuyus"` can be arranged in non-ascending order by both `2 0 1 3 4` and `2 0 3 1 4`. The duplicate item `"u"` is at indices 1 and 3, which are grouped together in both permutations as `1 3` in one and `3 1` in the other. The result of `>"xuyus"` is the first of these two, because the indices of duplicates in x are always in increasing order in `>x`.

Note that the permutation is the more generally useful result than sorting the argument directly because it often happens that several lists are to be reordered in the same way, based on the sort order of one of them.

Any list can be sorted, and for Grade Down the sort order is determined as follows:

1) First sort the items in the order of all atoms first, followed by lists in decreasing count, as in:

```
  > (1 4 3 2; 1.3 1.2 1.1; 3405; `a `b; "acb")
2 0 1 4 3
```

2) Within the group of atomic items, sort in the order: function atom first, then symbol atom, character atom, floating-point atom, integer atom.

3) Within each group of list items with equal count, sort in the order: general lists, including lists whose items are all functions, integer vectors, floating-point vectors, character vectors, symbol vectors.

4) Within each atom group of like type defined in 2) other than functions, sort according to the definitions of Equal and More. The sort order of function atoms is not described here.

5) Within each vector group of like type defined in 3), sort lexicographically. That is, first sort a group of vectors of the same type and count by their first items, as described in 4; then within each group, sort vectors with the same first item by their second items; and so on.

6) Sort items that are general lists of the same count lexicographically.

Facts About Grade Down

The indices of duplicate items in x are always in increasing order in `>x`.

Note that a list of characters vectors will not be sorted lexicographically, but first by the counts of items and then lexicographically within each group of equal counts. A symbol vector, which can be viewed as a string vector, is sorted lexicographically. For example:

```
  ("aaa";"bb";,"c";"d")[> ("aaa";"bb";,"c";"d")]
("d"
 "aaa"
 "bb"
 ,"c")
  `aaa `bb `c[> `aaa `bb `c]
`c `bb `aaa
```

Error Reports

Rank Error if the argument is an atom.

Grade Up

`< x`

Description

The permutation of `!#x` that sorts items of the list x in non-descending order.

Argument and Result

The argument x is any list. The result is a nonnegative integer vector with the same count as x.

Definition

The result of `<x` is a permutation of `!#x` for which the items of `x[<x]` are in non-descending order. For example:

```
  < 3 1 4 2
1 3 0 2
  3 1 4 2[1 3 0 2]
1 2 3 4
  "taped"[<"taped"]
"adept"
```

```
  ms:(8 1 6;3 5 7;4 9 2)
  <ms
1 2 0
  ms[<ms]
(3 5 7
 4 9 2
 8 1 6)
```

See Grade Down for a discussion of duplicate items. Like Grade Down, indices of duplicate items in the argument x are in increasing order in the result. Also like Grade Down, any list can be sorted with Grade Up. See Grade Down for a discussion of the general sort order, but note that except for duplicates, the order given there is the opposite of what it would be here.

Error Reports

Rank Error if the argument is an atom.

Group

`= x`

Arguments

Any list x.

Definition

Group produces a list of nonnegative integer vectors whose count is the number of distinct items in the argument, and:

- in which each item of `!#x` appears once and only once in the result, and;
- i and j are in the same item of the result if `x[i]` matches `x[j]` (see Match);

For example:

```
  = 2 1 2 2 1 1
(0 2 3                     the indices of 2
 1 4 5)                    the indices of 1
  = "weekend"
(,0                        the indices of "w" in "weekend"
 1 2 4                     the indices of "e"
 ,3                        the indices of "k"
 ,5                        the indices of "n"
 ,6)                       the indices of "d"
```

Each item of the result corresponds to a distinct item in the argument, and within each item the indices are in increasing order. Those distinct items are `?x`, i.e. the ith item of =`x` corresponds to the ith item of `?x` (see Range). For instance, in the previous example, the second item of the result, `1 2 4`, holds the indices of `"e"` in `"weekend"`, and `"e"` is the second distinct item in `"weekend"`.

The argument to Group can be any list, not just vectors. For example:

```
  =(9 2 3
    4 5
    9 2 3
    6 7 8
    4 5
    9 2 3)
```

```
(0 2 5                     items 0, 2, and 5 equal  9 2 3
 1 4                       items 1 and 4 equal  4 5
 ,3)                       item 3 is  6 7 8
```

Facts about Range

`?x` is identical to `x@*:'=x` (see Range).

Error Reports

Rank Error if the argument is an atom.

Index Item, or At

```
d @ i
```

Arguments

The left argument d is either a symbol atom, a dictionary, or any list, and the right argument is either nonnegative integer or symbolic.

Definition

If the left argument is a symbol atom then it must be a handle, and the definition proceeds as if the value of the global variable named in the symbol is used as the left argument (but see Handle in the chapter Terminology).

Index Item is a right-atomic function. Every atom in the right argument must be a valid index of d (one of `!#d`), or an entry in d for a dictionary d (one of `!d`). The result is equivalent to replacing each atom of the right argument with the item of the left argument whose index or entry is that atom. For example:

```
  "abcdefg" @ 4
"e"

  "abcdefg" @ 5 0 3 4 3
"faded"

  "abcdefg" @ (5 0;(3;,4 3))
("fa"                      items 5 and 0
 ("d"                      item 3
  ,"ed"))                  items 4 and 3

  (8 1 6; 3 5; 7 4 9 2) @ (2 1; 1 2 0)
((7 4 9 2                  item 2
  3 5)                     item 1
 (3 5                      item 1
  7 4 9 2                  item 2
  8 1 6))                  item 0
```

If d is a dictionary then the right argument is composed of the entries of d:

```
  d:.((`a;2 3 4);(`b;"abcdefg"))
  d @ `a
2 3 4
```

```
  d @ `b `a
("abcdefg"
 2 3 4)
```

The general case of Index Item can be defined recursively as follows, based on the definition for an atom right argument:

```
  IndexItem:{[d;i] :[ @ i; d @ i; d IndexItem/: i]}
```

That is, if i is not an atom then apply Index Item to d and every item of i.

Since `@` is a primitive verb, in situations where it cannot be determined from context whether it represents a monadic or dyadic function, the dyadic case is assumed. This assumption is not strict, however. For example, if `f:@` then f is dyadic, but it can also be given three or four arguments to evaluate Amend Item. The reason that multi-valence is allowed here is that the functionality of Index Item and Amend Item is closely related.

Facts about Index Item

`d @ i` is identical to `d . ,i`.

Error Reports

Domain Error if the symbol d is not a handle, i.e. does not hold the name of an existing global variable.

Index Error if any atom of the right argument is not a valid index of the left argument.

Type Error if any atom of the right argument is not an integer, symbol or nil.

Index, or Of

`d . i`

Arguments

The left argument d is either a dictionary, symbol atom, or any list. Each item of the right argument is either nonnegative integer or symbolic. The special case of an atom d other than a dictionary or symbol together with the empty list i is permitted.

Definition

If the left argument is a symbol atom then it must be a handle, and the definition proceeds as if the value of the global variable named in the symbol is used as the left argument (but see Handle in the chapter Terminology). If the right argument is a nonnegative integer atom then `d . i` is the ith item of d, i.e. equals `d @ i`. If d is an atom other than a dictionary or symbol then i must be the empty list, and the result equals d. It is also true that `d . ()` equals d for every d. The case of a dictionary d and a symbol atom i is discussed below in the section Dictionaries and Symbolic Indexing. Other than that case, the remainder of this section assumes that both d and i are non-empty lists.

The case where i is a non-negative integer vector

If the right argument is a nonnegative integer vector then d . i is a single item at depth `#i` in d, and i is called the path to that item-at-depth. The first item `i[0]` selects an item of d, then `i[1]` selects an item of that item, then `i[2]` selects an item of that one, and so on. For example:

```
  d:((1 2 3
     4 5 6 7)
    (8 9
     10
     11 12)
    (13 14
     15 16 17 18
     19 20))
  d . 1                        select item 1
(8 9
 10
 11 12)
```

```
  d . 1 2                    select item 2 of item 1
11 12
  d . 1 2 0                  select item 0 of item 2 of item 1
11
```

The selections at each level are individual applications of Index Item: first, item `d@i[0]` is selected, then `(d@i[0])@i[1]`, then `((d@i[0])@ i[1])@ i[2]`, and so on. These expressions can be rephrased using Over Dyad applied to Index Item; the first is `d@/i[0]`, the second is `d@/i[0 1]`, and the third is `d@/i[0 1 2]`. In general, for a vector i of any count, `d . i` is identical to `d@/i`. Continuing the above example:

```
  ((d @ 1) @ 2) @ 0          selection in terms of a series of @'s
11
  d @/ 1 2 0                 selection in terms of @-Over
11
```

The case where the items of i are non-negative integer vectors (cross-sectional index)

Index is cross-sectional when the items of i are lists. That is, items-at-depth in y are indexed for paths made up of all combinations atoms of `i[0]` and atoms of `i[1]` and atoms of `i[2]`, and so on to the last item of i. The simplest case of cross-sectional index occurs when the items of i are vectors. For example, `d .(2 0;0 1)` selects items 0 and 1 from both items 2 and 0:

```
  d . (2 0;0 1)
((13 14                      item 0 of item 2
 15 16 17 18)                item 1 of item 2
 (1 2 3                      item 0 of item 0
 4 5 6 7))                   item 1 of item 0
```

Note that items appear in the result in the same order as the indices appear in i.

The first item of i selects two items of d, as in `d@i[0]`. The second item of i selects two items from each of the two items just selected, as in `(d@i[0])@'i[1]`. If there had been a third vector item in i, say of count 5, then that item would select five items from each of the four items-at-depth 1 just selected, as in `((d@i[0])@'i[1])@''i[2]`, and so on. When the items of i are vectors the result is rectangular to at least depth `#i`, depending on the regularity of d, and the kth item of its shape vector is `#i[k]` for every k less than `#i`. That is, the first `#i` items of the shape of the result are `#:'i` .

The case where the items of i are rectangular non-negative integer lists

More general cross-sectional indexing occurs when the items of i are rectangular lists, not just vectors, but the situation is much like the simpler case of vector items. In particular, the shape of the result is `,/^:'i`.

The case where some of the items of i are nil

Nils in the right argument mean “select all”: if `i[0]` is nil, then continue on with d and the rest of i, i.e. `1 _ i`; if `i[1]` is nil, then for every selection made through `i[0]`, continue on with that selection and the rest of i, i.e. `2 _ i`; and so on. For example, `d .(;0)` means that the 0th item of every item of v is selected:

```
  d . (;0)
(1 2 3                    item 0 of item 0
 8 9                      item 0 of item 1
 13 14)                   item 0 of item 2
```

Another example, this time with `i[1]` equal to nil:

```
  d . (0 2;;1 0)
((2 1                     items 1 and 0 of item 0 of item 0
  5 4)                    items 1 and 0 of item 1 of item 0
 (14 13                   items 1 and 0 of item 0 of item 2
  16 15                   items 1 and 0 of item 1 of item 2
  20 19))                 items 1 and 0 of item 2 of item 2
```

Note that `d .(;0)` is the same as `d .(0 1 2;0)`, but in the last example, there is no value that can be substituted for nil in `(0 2;;1 0)` to get the same result, because when item 0 of d is selected, nil acts like `0 1`, but when item 2 of d is selected, it acts like `0 1 2`.

The general case of a nonnegative integer list i

In the general case, when the items of i are nonnegative integer atoms or lists, or nil, the structure of the result can be thought of as cascading structures of the items of i. That is, with nils aside, the result is structurally like `i[0]`, except that wherever there is an atom in `i[0]`, the result is structurally like `i[1]`, except that wherever there is an atom in `i[1]`, the result is structurally like `i[2]`, and so on. The general case of Index can be defined recursively in terms of Index Item by partitioning the list i into its first item and the rest:

```
Index:{[d;F;R] :[ _n ~ F; Index[d; *R; 1 _ R]
                  0 = #R; d @ F
                     @ F; Index[d @ F; *R; 1 _ R]
                          Index[d;; R]'F ]}
```

That is, `d . i` is `Index[d;*i;1 _ i]`. To work through the definition, start with F as the first item of i and R as the remainder. At each step in the recursion, if F is nil then select all of d and continue on, with the first item of the remainder R as the new F and the remainder of R as the new remainder; otherwise, if the remainder is the empty vector apply Index Item (the right argument F is now the last item of i), and we are done; otherwise, if F is an atom, apply Index Item to select that item of d and continue on in the same way as when F is nil; otherwise, apply Index with fixed arguments d and R, but independently to the items of the list F.

Dictionaries and Symbolic Indexing

If i is a symbol atom then d must be a dictionary or handle of a directory on the K-tree, and `d . i` selects the value of the entry named in i. For example, if:

```
dir: .((`a;2 3 4);(`b; "abcdefg"))
```

then `` `dir .`b `` is `"abcdefg"` and `` `dir .(`b;1 3 5) `` is `"bdf"`.

If i is a list whose items are nonnegative integer atoms and symbols atoms, then just like the nonnegative integer vector case, `d . i` is a single item at depth `#i` in d. The difference is that wherever a symbol appears in i, say as the kth item, the selection up to the kth item must produce a dictionary or a handle of a directory. Selection by the kth item is the value of an entry in that dictionary or directory, and further selections go on from there. For example:

```
  (1;.((`a; 2 3 4);(`b; 10 20 30 40))) . (1; `b; 2)
30
```

As we have seen above for the general case, every atom in the kth item of i must be a valid index of all items at depth k selected by `d . k # i`. Moreover, symbols can only select from dictionaries and directories, and integers cannot. Consequently, if the kth item of i contains a symbol atom, then all items selected by `d . k # i` must be dictionaries or handles of directories, and therefore all atoms in the kth item of i must be symbols.

It follows that each item of i must be made up entirely of nonnegative integer atoms, or entirely of symbol atoms, and if the kth item of i is made up of symbols, then all items at depth k in d selected by the first k items of i must by dictionaries.

Note that if d is either a dictionary or handle to a directory then `d . ,!d` is a list of values of all the entries.

Facts About Index

In the general case of a one-item list i, `d . i` is identical to `d @ *i`.

When the index i is the empty list, i.e. `()`, the meaning is “Index All”. That is, when i is the empty list then `d . i` is d. For example:

```
  1 2 3 . ()                  10 . ()
1 2 3                       10
```

The last paragraph in the definition of Index Item applies equally to Index and Amend.

Error Reports

Domain Error if the symbol d is not a handle, i.e. does not hold the name of an existing global variable.

Index Error if any atom in i is not a valid index to the item-at-depth in d.

Type Error if any atom of i is not an integer, symbol or nil.

Join

```
x , y
```

Arguments

Any atoms or lists.

Definition

Join connects the items of x with the items of y. The count of `x,y` is `(#x)+(#y)`. If x is an atom then it is identical to the first item of `x,y`, and if y is an atom it is identical to the last item of `x,y`. If x is a list then `x[i]` is identical to `(x,y)[i]`, and if y is a list then `y[i]` is identical to `(x,y)[(# x)+i]`. For example:

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

  1 2 3 , 4 5 6 7
1 2 3 4 5 6 7

  1 2 3 , (8 1 6;3 5 7;4 9 2)
(1                     item 0 of the join
 2                     item 1
 3                     item 2
 8 1 6                 item 3
 3 5 7                 item 4
 4 9 2)                item 5

  ("canoe";`dinghy),("kayak";66545;{x + y})
("canoe"               item 0 of the join
 `dinghy               item 1
 "kayak"               item 2
 66545                 item 3
 {x + y})              item 4
```

Less

`x < y`

Arguments

The arguments x and y are conformable atoms or lists. When both are atoms they must both be numeric, or both characters, or both symbols. The result is the integer atom 0 or 1, or an integer list consisting of 0's and 1's.

Definition

Less is an atom function. For atoms x and y, the value of `x < y` is:

- 1 if x and y are numeric, and x is less than y in the usual mathematical sense, and 0 otherwise;
- 1 if x and y are characters, and the ASCII value of x is less than the ASCII value of y, and 0 otherwise (see the system function `_ic` for ASCII values);
- 1 if x and y are symbols, and x comes before y in lexicographic order, and 0 otherwise.

For example:

```
  1 < -1 0 1 2
0 0 0 1
  "a" < "z"
1
  "aA" < "Z"
0 1
  `inch `mile < `foot `yard
0 1
```

Comparison tolerance is used when both x and y are numeric and at least one is floating-point. That is, `x < y` is 0 if x and y are close in value, even if x is actually less than y.

```
  1 < 1.000000000001          0.999999999999 < 1
1                           1
  1 < 1.0000000000001         0.9999999999999 < 1
0                           0
```

Note that `0I` (integer infinity) is greater than all other integers, `0N` is less than all other integers, and `-0I` is less than all integers except `0N`. Similar relations hold for the floating-point values `0i`, `-0i`, and `0n`.

Error Reports

Length Error if the arguments are not conformable.

Type Error if atoms x and y are not both numeric, or not both character, or not both symbols.

Make / Unmake Dictionary

`. x`

Description

Create a dictionary from a list x of a special form, or create a list of that form from a dictionary x.

Argument

The argument x is either a dictionary or list satisfying the following conditions:

(i) each item is a list with two or three items;

(ii) the first item of each item is a symbol that is a valid dictionary entry; and

(iii) if an item has three items then the third one is a dictionary.

Definition

If x is a list as described above then `. x` is a dictionary whose entries are the first items of the items of `x`, i.e. `*:'x`. That is, the dictionary entry created from the ith item `x[i]` is `x[i;0]`. Also, the value of the dictionary entry created from `x[i]` is `x[i;1]`, and if `x[i]` has three items then `x[i;2]` is the attribute dictionary of that entry. For example, the following expression creates a dictionary with 2 entries `` `a `` and `` `b ``, and the display class of `` `b `` is `` `button ``:

```
  c: .((`a;1 2 3);(`b;"2+3";.,(`c;`button)))
  c.a
1 2 3
  c.b
"2+3"
  c.b..c
`button
```

If x is a dictionary then `. x` is a list of this special form, and x is identical to `.(. x)`.

Error Reports

Rank Error if the argument x does not have the appropriate shape as specified in (i) above.

Type Error if any item of the items of x is not as specified in (ii) and (iii) above.

Match

```
x ~ y
```

Arguments

Any atoms or lists x and y.

Definition

The result is the integer atom 1 if x is identical to y, and otherwise it is the integer atom 0. Comparison tolerance is used when comparing numeric values when at least one is floating-point; see Equal for an example of its effect. Empty lists do not necessarily match other empty lists; they must be of the same type.

```
  2 3 ~ 2 3           () ~ !0            "a" ~ ,"a"
1                   0                  0
```

Max / Or

```
x | y
```

Arguments

The arguments are conformable numeric atoms or lists.

Definition

Max / Or is an atom function. For atoms x and y the result is the mathematically greater of the two. For example:

```
  3 | 8                          123.45 | 987.65
8                              987.7
  3 | -8                         123.45 | -987.65
3                              123.4
```

If both arguments are integers, the result is integer, and if at least one is floating-point, the result is floating-point.

When the arguments consist of the integer atoms 0 or 1, the result is the logical-Or function:

```
  0 0 1 1 | 0 1 0 1
0 1 1 1
```

Error Reports

Length Error if the arguments are not conformable.

Type Error if either argument is not numeric.

Min / And

```
x & y
```

Arguments

The arguments are conformable numeric atoms or lists.

Definition

Min / And is an atom function. For atoms x and y the result is the mathematically lesser of the two. For example:

```
  3 & 8                    123.45 & 987.65
3                        123.45
  3 & -8                   123.45 & -987.65
-8                       -987.65
```

If both arguments are integers, the result is integer, and if at least one is floating-point, the result is floating-point.

When the arguments consist of the integer atoms 0 or 1, the result is the logical-And function:

```
  0 0 1 1 | 0 1 0 1
0 0 0 1
```

Error Reports

Length Error if the arguments are not conformable.

Type Error if either argument is not numeric.

Minus

```
x - y
```

Arguments

The arguments are conformable numeric atoms or lists.

Definition

Minus is an atom function. For atoms x and y the result is the mathematical difference of the two.

If both arguments are integers, the difference is integer, and is computed using integer arithmetic.

If one of the arguments is floating-point, the other is made floating-point (if it is not already), and the difference, which is also floating-point, is computed using floating-point arithmetic.

The result is 0.0 if the mathematical result would be too small in magnitude to be represented as a floating-point number.

Error Reports

Length Error if arguments are not conformable.

Type Error if either argument is not numeric.

More

```
x > y
```

Arguments

The arguments x and y are conformable atoms or lists. When both are atoms they must both be numeric, or both characters, or both symbols.

Definition

More is an atom function. For atoms x and y, the value of `x > y` is:

- 1 if x and y are numeric, and x is greater than y in the usual mathematical sense, and 0 otherwise;
- 1 if x and y are characters, and the ASCII value of x is greater than the ASCII value of y, and 0 otherwise (see the system function `_ic` for ASCII values);
- 1 if x and y are symbols, and x comes after y in lexicographic order, and 0 otherwise.

For example:

```
  1 > -1 0 1 2
1 1 0 0
  "a" > "z"
0
  "aA" > "Z"
1 0
  `inch `mile > `foot `yard
1 0
```

Comparison tolerance is used when both x and y are numeric and at least one is floating-point. That is, `x > y` is 0 if x and y are close in value, even if x is actually greater than y.

```
  1.000000000001 > 1          1 > 0.999999999999
1                           1
  1.0000000000001 > 1         1 > 0.9999999999999
0                           0
```

Note that `0I` (integer infinity) is more than all other integers, `0N` is less than all other integers, and `-0I` is less than all integers except `0N`. Similar relations hold for the floating-point values `0i`, `-0i`, and `0n`.

Error Reports

Length Error if the arguments are not conformable.

Type Error if atoms x and y are not both numeric, or not both character, or not both symbols.

Negate

`- x`

Argument

x is any numeric atom or list.

Definition

Negate is the atom function defined by `0 - x`. The type of the result (integer or floating-point) for an atom argument x is the same type as x.

Error Reports

Type Error if the argument x is not numeric.

Not / Attribute

```
~ x
```

Argument

The argument x is a numeric atom, symbolic atom, or a list whose atoms are all numeric or all symbols.

Definition

Not / Attribute is an atom function. If x is numeric, then `~x` is `0=x`. For example:

```
  ~ 1 0                       logical negation
0 1
  ~ 4.6 0 -4.6                general case
0 1 0
```

If x is a symbol atom, say `` `a``, then `~x` is `` `a..`` Consequently, if the symbol x is a handle holding the name of a global variable, then `~x` is the handle of the attribute directory of that variable. For example, if x is a handle then the dependency definition on this handle is ``(~x).`d``, and the dependency definitions for all entries in x are ``x[~!d;`d]``.

Also, the compound handle `` `a.b.c`` can be produced from the simple handles `` `a``, `` `b``, and `` `c`` as follows:

```
  {`$($ ~ x) , $ y}/`a `b `c
```

(see Format, Format (Dyadic) and Over).

Error Reports

Type Error if an atom in x is not a symbol or numeric.

Plus

```
x + y
```

Arguments

The arguments are conformable numeric atoms or lists. The result is numeric; if both arguments are integer atoms or list, the result is integer.

Definition

Plus is an atom function. For atoms x and y the result is the mathematical sum of the two.

If both arguments are integers, the sum is integer, and is computed using integer arithmetic.

If one of the arguments is floating-point, the other is made floating-point (if it is not already), and the sum, which is also floating-point, is computed using floating-point arithmetic.

The result is 0.0 if the mathematical result would be too small in magnitude to be represented as a floating-point number.

Error Reports

Length Error if arguments are not conformable.

Type Error if either argument is not numeric.

Power

```
x ^ y
```

Arguments

The arguments are conformable numeric atoms or lists.

Definition

Power is an atom function. Integer arguments are converted to floating-point before the function is applied. The definition for atoms x and y is as follows:

- If x is positive then `x^y` is identical to `exp[y*log[x]]`. (A special case is a positive whole number y, where `x^y` is the product of x with itself y times);
- If x equals 0 then `x^y` is `0.0` for all nonzero y;
- If y is 0 then `x^y` is `1.0` for all x;
- If x is negative and y is a whole number then `x^y` is `-(-x)^y` if y is odd and `(-x)^y` if even.

For example:

```
  2^3
8.0
  -2^3
-8.0
  0^0
1.0
```

```
  -2.0^2
4.0
  2.0^0.5
1.414
  10^1000
0i
```

The result is 0.0 if the mathematical result would be too small in magnitude to be represented as a floating-point number. Also, the result is `0i` (infinity) if the mathematical result would be too large to be represented, and similarly for `-0i` .

Error Reports

Domain Error if the left argument is negative and the right argument is not a whole number.

Length Error if arguments are not conformable.

Type Error if either argument is not numeric.

Range

```
? x
```

Argument

Any list.

Definition

The result is a list of the unique items of x, in the order of their first occurrence (i.e., the occurrence with the smallest index). For example:

```
  ? 9 6 8 6 9 7 8 9 6        ? "strange"
9 6 8 7                    "strange"
                             ? "raccoon"
                           "racon"
```

```
  ? (9 2 3;4 5;9 2 3;6 7 8;4 5;9 2 3)
(9 2 3
 4 5
 6 7 8)
```

See the primitive function Group for the relationship between it and this primitive.

Range is an identity function for empty lists.

Error Reports

Rank Error if x is an atom.

Reciprocal

```
% x
```

Arguments

The argument x is a numeric atom or list.

Definition

Reciprocal is an atom function defined by `1%x`. The result is always floating-point, even when x is 1. When x is 0 or `-0`, the result is `0i`, i.e. floating-point infinity. Similarly, the reciprocal of infinity (and minus infinity) is `0.0`.

```
  %2                  %1                  %0
0.5                 1.0                 0i
```

Error Reports

Type Error if x is not numeric.

Reverse

```
| x
```

Arguments

Any atom or list.

Definition

This function reverses the order of the items in its argument. For example:

```
  | 3 1 4 2
2 4 1 3
  m : (8 1 6 ; 3 5 7 ; 4 9 2)
  m                          | m
(8 1 6                     (4 9 2
 3 5 7                      3 5 7
 4 9 2)                     8 1 6)
```

Also, since the primitive function First produces the first item in a list x, the K idiom First-Reverse, denoted `*|x`, produces the last item.

```
  *| `one `two `three
`three
```

Reverse is an identity function for all atoms, empty lists, and one-item lists.

```
  | "a"                 | !0                  | ,3 1 4 2
"a"                   !0                    ,3 1 4 2
```

Rotate / Mod

`x ! y`

Description

Rotate the list y by x positions, or compute the remainder of the atom y divided by x.

Arguments

Rotate: The left argument is an integer atom and the right argument is any list;

Mod: the left argument is a numeric atom or list and the right argument is a numeric atom.

Definition

If the right argument is a numeric atom then `x!y` is Mod and the result is the remainder of x divided by y, i.e. `x - y * _ x % y` . f both arguments are integer, so is the result. Otherwise, the result is floating-point. For example:

```
  5 ! 3                         5 ! -3
2                             -1

  1.8 -2.7 ! 0.2                -3 4 -17 ! -4
0 0.1                         -3 0 -1
```

If the right argument is a list then `x!y` is Rotate, and the result is a list with the same items as y, but rotated `x!#y` positions (and hence the connection between Mod and Rotate). The rotation is towards the front if x is positive and the back if x is negative (that is, towards index position 0 if x is positive and index position `-1+#y` if x is negative).

Specifically, if x is positive and less than `#y` then item `y[x]` becomes item 0 of the result, item `y[x+1]` becomes item 1, and so on; item `y[x-1]` becomes the last item of the result, item `y[x-2]` the next-to-last, and so on. If x is greater than or equal to `#y` the remainder `x!#y` is used for x. For example:

```
  5 ! "abcdefgh"                21 ! "abcdefgh"
"fghabcde"                    "fghabcde"
```

If x is negative the rotation is in the other direction, i.e. if the positive value `-x` is less than `#y` then item `y[0]` becomes item `-x` of the result, and so on. As above, if `-x` is greater than or equal to `#y` the remainder `x!#y` is used for x. For example:

```
  -5 ! "abcdefgh"              -21 ! "abcdefgh"
"defghabc"                   "defghabc"
```

Both Mod and Rotate are atom functions of the left argument, i.e. `![;y]` is an atom function for every y. In either case the result is structurally like the left argument x, except that in the case of Mod, every atom in x is replaced by the remainder of that atom divided by y, while for Rotate, every atom in x is replaced by the list y, rotated according to that atom.

Error Reports

Int Error if the left argument is not integer.

Type Error if an atom right argument is not numeric.

Shape

```
^ x
```

Arguments

Any atom or list.

Definition

Associated with every data object is an integer vector called the shape of the object. An atom has an empty shape:

```
  ^ 3.14
!0
```

A list x has a non-empty shape whose first item is the count of the list, while subsequent items are counts of the items-at-depth for successive depth levels in x, and are present only if the list is sufficiently regular to those depths. For example, consider the following list r:

```
  r:(("ab";"cd";"ef")
     ("gh";"ij";"kl")
     ("mn";"op";"qr")
     ("st";"uv";"wx"))
```

This is a list of four items, each of which is a list of three items, and each of these has items that are character vectors of count 2. This list is said to be rectangular because it is rectangular at every level, which in this case means that all items are lists of the same count, and all items of items are lists of the same count. The shape of a rectangular list gives the counts of the items at consecutive levels, in depth order:

```
  ^ r
4 3 2
```

Each level in this example is rectangular. A list that isn't rectangular at every level may be rectangular for the first n levels, but irregular below that, in which case it is said to rectangular to level n (see the topic Rectangular List in the chapter Terminology). For example:

```
  s:(("aby";"cd";"ef")
     ("gh";"i";"kl1")
     ("mn";"opz";"qr")
     ("st";"uv";"w"))
```

is rectangular to level 2, since every item is a list of count 3, but the items of items vary. The shape of this list has only two items, even though it has three levels:

```
  ^ s
4 3
```

In the next example every item at level 2 has count 2, but the items of the list, which are the lists at level 1, do not all have the same count. The shape of this list has only one item, the count of the list, because it is not rectangular at level 2. Even though the items at level 2 are all lists of the same count, the list cannot be rectangular at that level because of gaps introduced at level 1.

```
  t:(("ab";"cd";"ef")
     ("gh";"ij")
     ("kl";"mn";"op";"qr")
     ("st";"uv";"wx"))
  ^ t
,4
```

The shape of a list always has at least one item, the count of the list. The only item of the result of shape that can be zero is the last.

Take / Reshape

```
x # y
```

Arguments

The left argument for Take is either an integer atom or an integer vector. The right argument is any atom or list.

Definition

If the left argument x is a nonnegative integer atom then the result of `x#y` is a list whose count equals x and which consists of the first x items of y. If x is a negative integer atom the result is a list whose count equals -x and which consists of the last -x items of y.

```
  3 # 4 5 6 7 8 9                -3 # 4 5 6 7 8 9
4 5 6                           7 8 9
```

If x is greater than `#y` or less than `-#y` then the items of y are used repeatedly to fill out the result; if x is positive the items are of y are selected cyclically from first to last and are placed in the result first to last, while if x is negative the items of y are selected cyclically from last to first and are placed in the result from last to first. For example:

```
  8 # 4 5 6 7 8 9                -13 # 4 5 6 7 8 9
4 5 6 7 8 9 4 5                 9 4 5 6 7 8 9 4 5 6 7 8 9
```

Note that for positive x, the shape of the result is `x,1_^y`.

The second case of Take is an integer vector x, and extends the first case. With one exception x must be nonnegative, and that case is discussed below. For now assume that x is positive, in which case `x#y` is just like the case for positive integer x, except the selected items are arranged in a rectangle of dimensions x. For example:

```
  r:(("ab";"cd";"ef")
     ("gh";"ij";"kl")
     ("mn";"op";"qr")
     ("st";"uv";"wx"))
  r ~ 4 3 2 # "abcdefghijklmnopqrstuvwxyz"
1
```

Once again, the shape of the result is `x,1_^y`.

If the vector x contains 0’s then the shape of the result the vector consisting of the leading items of x, up to and including the first 0, that is, `x[!1+x?0]`. For example:

```
  ^ 2 3 4 0 7 8 # "abc"
2 3 4 0
```

If the vector x contains negative integers, there must be only one and its value must be -1. For example, `2 -1 3` is a valid left argument but `2 -2 3` and `-1 2 3 -1` are not. The following examples illustrate the general case:

```
  2 -1 # "abcd"            2 -1 # "abcdefgh"
("ab"                    ("abcd"
 "cd")                    "efgh")
```

The shapes of the results:

```
  ^ 2 -1 # "abcd"          ^ 2 -1 # "abcdefgh"
2 2                      2 4
```

These examples illustrate the general case for the left argument `2 -1`; y can be any list with an even number of items, and the result is a list of shape equal to `2,(#y)%2` . The -1 in the left argument stands for `(#y)%2` . A second example:

```
  2 -1 3 # "abcdef"        2 -1 3 # "abcdefghijkl"
(,"abc"                  (("abc"
 ,"def")                   "def")
                          ("ghi"
                           "jkl"))
```

The shapes of the results:

```
  ^ 2 -1 3 # "abcdef"      ^ 2 -1 3 # "abcdefghijkl"
2 1 3                    2 2 3
```

Once again these examples illustrate the general case for the left argument `2 -1 3`; y can be any list for which `#y` is a multiple of 6, and the result has shape equal to `2,((# y)%6),3`. The -1 in the left argument stands for `(#y)%6`.

The general idea is that the item of the result shape corresponding to -1 is not specified by x, but is computed to be `#y` divided by the product of all other items of x except -1. If -1 is included in the product it simply negates what it would be otherwise, and therefore the result shape is `(#y)%-*/x`.

Facts About Take

If the right argument is the empty list then the left argument must be the atom 0 or a vector containing one or more zeros.

Error Reports

Domain Error if the left argument is an integer vector and contains more than one negative integer, or exactly one that is not -1.

Int Error if the left argument is not integer.

Length Error if the vector left argument contains one occurrence of -1 but `#y` is not equal to an integer multiple of `-*/x`.

Times

```
x * y
```

Arguments

The arguments are conformable numeric atoms or lists. The result is numeric, and if both arguments are integer atoms or list, the result is integer.

Definition

Times is an atom function. For atoms x and y the result is x multiplied by y.

If both arguments are integers, the product is integer, and is computed using integer arithmetic.

If one of the arguments is floating-point, the other is made floating-point (if it is not already), and the product, which is also floating-point, is computed using floating-point arithmetic.

The result is 0.0 if the mathematical result would be too small in magnitude to be represented as a floating-point number. Similarly, the result is `0i` or `-0i` when the result is too large in magnitude.

Error Reports

Length Error if arguments are not conformable.

Type Error if either argument is not numeric.

Value / Execute

```
. x
d @ s
```

Description

Evaluate the contents of x, and if d is present, in the directory named in d.

Arguments

The argument x is either a character vector, enlisted character vector or a symbol atom. The argument d is a symbol atom.

Definition

If in the monadic case x is a character vector then its contents must be a valid expression or command, and the effect of Value is to evaluate that expression or execute that command. The result of `. x` is the result of the expression, except when the last thing executed in the expression is Assign, Amend or Item Amend (see the chapter Amend, Index, Apply and Assign), in which case the result is nil (these exceptions are exactly the same as when the result of an expression entered in an interactive session is not displayed). The result is also nil when the contents of x are a valid command because commands do not have explicit results. For example:

```
  . "2+3*4"
14
  . "r: 2+3*4"
```

Nothing displayed; the result is nil.

Again in the monadic case, if x is a symbol atom that holds the name of a global variable, the esult of `. x` is the value of that variable. For example:

```
  f: {x ^ 2}
  . `f
{x ^ 2}
```

In the dyadic case `d@x` the left argument must be a symbol holding the name of a directory on the K-tree, in which case the execution or evaluation of x takes place in that directory. This means that any relative names in x are resolved with that directory as the reference point. Names with a single leading dot are absolute references and always refer to the same object.

If d is not present the execution or evaluation of x takes place in the current directory.

Execution of a character vector x proceeds just as if its contents were typed in an interactive session and Return was pressed, assuming that in the dyadic case that the Directory command was first executed to change to the current directory to the one named in d. In particular, if x is a character vector holding a function expression, as in `"{x + a}"`, the effect of Execute is to bind any relative references to global variables – a in this example – to the directory named in d, or in the absence of d, the current directory.

Note that when both d and x are symbol atoms, `d@x` and `d . x` coincide with cases of Index Item and Index.

The monadic case `. s` is identical to `_d @ s`.

Error Reports

Domain Error if the argument x is not a character vector, enlisted character vector, or symbol atom, or if the symbol d is not a handle, i.e. does not hold a valid directory name.

Where

```
& x
```

Arguments

The argument x is any nonnegative integer atom or vector.

Definition

The result is a nonnegative integer vector containing indices of x, i.e. integers from `! # x`, where each index i appears `x[i]` times. In particular, for a boolean list x, the result is a vector of the indices of x where 1's appear. For example:

```
   & 0 0 1 0 1 0 0 1
2 4 7                              the indices of 1's in the argument
   & 3 0 4
0 0 0 2 2 2 2                      three 0's, no 1's, and four 2's
```

An atom is treated like a one-item list, and always returns that many 0's, as in:

```
   &3
0 0 0
```

Error Reports

Domain Error if the argument is an integer atom or vector, but contains negative integers.

Int Error if the argument is not an integer atom or vector.

CHAPTER 5

ADVERBS

Adverbs (sometimes called operators) modify nouns and verbs to create new verbs. For example, Plus is a dyad that produces the sum of its two arguments, while Plus-Over, denoted by `+/` , is a monad that produces the sum of all items of its argument. There are six adverbs:

- Each, denoted by `'`, which applies the function it modifies to the items of its arguments, rather than the arguments themselves;
- Each Pair (`':`),
- Each Right (`/:`), and
- Each Left (`\:`), which are variants of Each;
- Over (`/`) and its counterpart,
- Scan (`\`).

The object modified by an adverb is called its operand. Syntactically, an adverb symbol must be immediately adjacent to its operand. For example, `+/` is Plus-Over, but `+ /` is not.

Just as verbs are resolved to functions for execution, the verbs created by adverbs are resolved to derived functions.

There are essentially two sources of errors when using adverbs: the adverbs themselves and the functions to which they are applied. For example, in applying the Each adverb to a dyad f, as in the expression `x f' y`, the expression will fail if x and y are not conformable, which is an Each error, or if the function `f` fails for some pair `x[i]` and `y[i]` . The errors listed in any Error Reports subsection in this chapter are those for which the adverbs are the direct cause.

Each

```
f' x
x f' y
f'[x;y;z...]
```

Description

Apply the function f to the items of the argument(s).

Arguments

In the first case, the operand f is a monad and x is an atom or list. In the second case, f is a dyad and x and y can be either atoms or lists, but if both are lists they must have the same count. The last case extends the second case: f can have any valence of at least one, the number of objects between brackets equals the valence of f, and each argument can be an atom or list, but if two or more are lists then all lists must conform.

Definition

In the first case, Each applies a monad f to each item of x:

```
  q: !:' 6 4 5          Enumerate-Each
  q
(0 1 2 3 4 5            Enumerate 6
 0 1 2 3                Enumerate 4
 0 1 2 3 4)             Enumerate 5
  #:' q                 Count-Each
6 4 5
  |:' q                 Reverse-Each
(5 4 3 2 1 0
 3 2 1 0
 4 3 2 1 0)
  +/' q                 Sum-Each
15 6 10
```

Note that whenever Each is applied to the monad of a primitive verb, as in `!:'` for Enumerate-Each, the monadic case must be made explicit by modifying the verb with colon. The dyadic case is assumed if no modifier is present. For example, Take-Each, an example of the dyadic case `x f'y`:

```
  4 -7 9 #' !:' 6 4 5
(0 1 2 3
 1 2 3 0 1 2 3
 0 1 2 3 4 0 1 2 3)
```

In the dyadic case `x f'y` , the arguments x and y must conform. For example, Join-Each of two atoms is the same as Join:

```
  "a" ,' "b"                    ("a" ,' "b") ~ "a" , "b"
"ab"                          1
```

Join-Each of an atom and a list joins the atom to each item of the list:

```
   "a" ,' "bcd"                  "abc" ,' "d"
 ("ab"                         ("ad"
  "ac"                          "bd"
  "ad")                         "cd")
```

Join-Each of two lists joins items of one to items of the other:

```
   "abc" ,' "def"
 ("ad"
  "be"
  "cf")
```

The general case is a straightforward extension of the dyadic case. If there are only two arguments x and y then `f'[x;y]` is identical to `x f'y` . Otherwise, if all arguments are atoms, `f'[x;y;z;…]` is identical to `f[x;y;z;…]` . In general, when there are lists among the arguments they must all conform. The ith item of the result is

```
  f[ xi; yi; zi; …]
```

where `xi` denotes `x[i]` if x is a list and x itself if x is an atom, and similarly for `yi`, `zi`, etc. As in the other cases, if the list arguments are the empty list then so is the result.

If x is an atom then `f'x` is `f x` .

The valence of `f'` equals the valence of f .

If at least one argument of `f'` is the empty list then so is the result, and the function f is never applied.

Error Reports

Length Error if, in the case of dyadic f, the arguments x and y do not conform, or in the general case, two or more arguments are nonconforming lists.

Each Left

`x f\: y`

Description

Apply dyad f to each item of x with all of y.

Arguments

The operxand f is a dyadic function, and the arguments x and y can be any atom or list.

Definition

If x is an atom then `x f\:y` is `f[x;y]`.

If x is a list then `x f\:y` is a list with the same count as x and the ith item of the result is `f[x[i];y]` for every i.

For example, Join-Each-Left joins every item of the left argument to the right argument:

```
  2 3 4 ,\: 5 6 7
(2 5 6 7
 3 5 6 7
 4 5 6 7)
```

(Compare with the example `2 3 4,/:5 6 7` in the section on Each Right).

A commonly used K idiom is “In-Each-Left”, that is, `_in\:`. By itself, the system function `_in` searches for its left argument among the items of its right argument. However, often one wants to search for every item of the left argument among the items of the right, and that functionality is provided by `_in\:`. For example:

```
  4 _in 1 7 2 4 6 3
1                          4 is an item of the right argument
  4 3 _in 1 7 2 4 6 3
0                          vector 4 3 is not an item of the right
  4 3 _in\: 1 7 2 4 6 3
1 1                        both 4 and 3 are items of the right
```

See the system function `_lin` and the companion idiom `?/:` in the section on Each Right.

If x is the empty list then so is the result, and the function f is never applied.

`x f\:y` is identical to `x f'y` for atoms y .

`x f\:y` is identical to `f[;y]'x` for all x and y (see Projection; Fixing Function Arguments in the chapter Functions).

Each Pair

```
f': y
x f': y
```

Description

Apply dyad f to pairs of consecutive items.

Arguments

The operand f is a dyadic function, and the arguments x and y can be any atom or any non-empty list.

Definition

Each Pair applies its argument function f to successive pairs of consecutive items in the list argument y. If y is a list of count at least two, then `f':x` is a list with count `(#x)-1` and the ith item of the result is `f[x[i+1];x[i]]`. For example:

```
  -': 1 4 9 14 25 36
3 5 5 11 11
```

Find sentence endings in text, say all periods followed immediately by blanks:

```
  a: "This seeks sentence endings. There are two. "
  & {(x=" ")&y = "."}': a
27 42
  a[27 42]                    check that they are indeed periods
".."
```

The result is always the empty list when the argument x is a list of count 1, and the function f is never applied.

The dyadic case `x f':y` is defined in terms of the monadic as `(,x),f':y`, and in addition is defined for the case when y is the empty list.

`f':x` is `f[x;x]` for atoms x.

Error Reports

Length Error in the monadic case if the argument is the empty list.

Each Right

```
x f/: y
```

Description

Apply dyad f to all of x with each item of y.

Arguments

The operand f is a dyadic function, and the arguments x and y can be any atom or list.

Definition

If y is an atom then `x f/:y` is `f[x;y]`, and if y is a list then `x f/:y` is a list with the same count and the ith item of the result is `f[x;y[i]]` for every i. If y is the empty list then so is the result.

For example, Join-Each-Right joins every item of the right argument to the left argument:

```
  2 3 4 ,/: 5 6 7
(2 3 4 5
 2 3 4 6
 2 3 4 7)
```

(Compare with the example `2 3 4,\:5 6 7` in the section on Each Left).

A commonly used K idiom is "Find-Each-Right", that is, `?/:`. By itself, Find searches for its right argument among the items of its left argument. However, often one wants to search for every item of the right argument among the items of the left, and that functionality is provided by `?/:`. For example:

```
  1 7 2 4 6 10 3 ? 4
3                                   index of 4 in left argument
  1 7 2 4 6 10 3 ? 4 3
7                                   4 3  is not an item of left argument
  1 7 2 4 6 10 3 ?/: 4 3
3 6                                 indices of 4 and 3 in left argument
```

See the companion idiom `_in\:` in the section on Each Left.

If y is the empty list then so is the result, and the function f is never applied.

`x f/:y` is identical to `x f'y` for atoms x.

`x f/:y` is identical to `f[x;]'y` for all x and y (see Projection; Fixing Function Arguments in the chapter Functions).

Over Dyad

```
f/ y
x f/ y
```

Description

In the `f/ y` case: `(…((y[0] f y[1]) f y[2]) f …) f (*|y)`

In the `x f/ y` case: `(…(x f y[0]) f y[1]) f …) f (*|y)`

Arguments

The operand f is a dyad, and the arguments x and y are any atoms or lists.

Definition

Consider the second case, `x f/ y`. The left argument x must be a valid left argument of f and every item of the right argument must be a valid right argument, unless it is an atom, and then the atom must be a valid right argument.

If y is a non-empty list the evaluation proceeds as follows:

```
  x: f[x; y[0]]
  x: f[x; y[1]]
  .
  .
  .
  x: f[x; *|y]
```

`*|y` is the last item of y

That is, f is applied iteratively to the left argument with the items of y as successive right arguments. The result of `x f/ y` is the last value of x in the above sequence. For example, all items of a list can be added to an initial value as follows:

```
  10 +/ 1 2 3
16
```

The first case, `f/ y`, is similar. If y is a list with at least one item the evaluation proceeds as follows:

```
   x: y[0]
   x: f[x; y[1]]
   x: f[x; y[2]]
   .
   .
   .
   x: f[x; *|y]           *|y  is the last item of  y
```

For example:

```
  +/1 2 3                 +/1                    +/,1
6                       1                      1
```

Over Dyad is used in two important K-idioms, `|/` and `&/`, generally known as Maximum and Minimum, respectively. When applied to a numeric vector, `|/` produces the greatest value among the items of its argument, and `&/` produces the least value. For example:

```
  |/ 1 4 -6 9 1 3            &/ 1 4 -6 9 1 3
9                          -6
```

In the special case where the argument x is a vector with boolean values 0 and 1, `|/x` is 0 only if all items of x are 0, and `&/x` is 1 only if all items of x are 1. Consequently `|/` and `&/` applied to boolean vectors are used to check for the condition “if some condition is true” and “if all conditions are true”.

See Over Monad for finding the largest and smallest values in an arbitrary numeric list.

Note that when Over is applied to a primitive verb there is no immediate context to establish whether the verb denotes its monad or dyad. The general rule is applied, which says that it is the dyad. For example, `+/` is Plus-Over, not Flip-Over. As in other situations, the monad must be explicitly specified by modifying the symbol with a colon, as in `+:/` .

Like the primitive verbs, the derived verb `f/` for dyadic f has two cases, one monadic and one dyadic, but unlike the primitive verbs, in situations where the valence cannot be determined from context the monadic case is assumed. For example, if `f: +` then f is strictly dyadic; if `g: +:` then g is strictly monadic, but if `h: +/` then h is monadic. However, h is not strictly monadic; it can be evaluated as a dyad, as in `h[x;y]`, but when a choice must be made, it will be monadic. The reason that ambivalence is allowed here is that the functionality of the monadic and dyadic cases of Over is essentially the same.

If y is an atom then `f/y` is y and `x f/y` is `f[x;y]` .

If y is the empty list then so is `x f/y` and the function f is never applied.

If y is a one-item list then `f/y` is `*y` .

If y is empty and f is either `+` , `*` , `|` , or `&` , then `f/y` is 0, 1, 0, or 1, respectively, and the function f is never applied.

Note that `x f/y` is identical to `f/(,x),y` .

Error Reports

Length Error in `f/x` if the argument x is the empty list and function f is not one of `+` , `*` , `|` , or `&` .

Over

```
f/[x;y;z;…]
```

Description

Apply f iteratively to x and successive items of y, z, ...

Arguments

The operand f is a function with at least two arguments. The relationship of the Over-f arguments to the arguments of f is like that of Each-f to f except for the first argument x. That is, each argument other than the first can be an atom or list, but if two or more of those are lists then all those lists must conform. The first argument x is any valid first argument of f.

Definition

The general case of Over for functions with valence at least two is a direct extension of the dyadic case. The first argument x serves the same role as the left argument of the dyadic case, and all other arguments have the same role as the right argument. That is, if all of y, z,... are atoms then `f/[x;y;z;…]` is identical to `f[x;y;z;…]` , and otherwise `f/[x;y;z;…]` is evaluated as follows:

```
x: f[x;y0;z0;…]
x: f[x;y1;z1;…]
.
.
.
x: f[x;yn;zn;…]
```

where y0 is `y[0]` if y is a list or y itself if y is an atom, and similarly for y1...yn, z0...zn. The name yn stands for the last item of y if y is a list or y itself if y is an atom, and similarly for zn, ... The result is the last value of x.

See the sections on Amend and Amend Item in the chapter Verbs for examples where Over is used in the definitions of functions that describe the behavior of these primitives.

For dyadic f, `f/[a;b]` is identical to `a f/ b` .

For functions f with valence at least three, the valence of `f/` equals the valence of f.

If all list arguments other than the first equal the empty list then the result is the first argument, and f is never applied.

Error Reports

Length Error if any two list arguments among y, z,... do not conform.

Over Monad

```
f/ x
n f/ x
b f/ x
```

Description

Apply f iteratively to x until, in the first case, a result matches either the previous or the initial result, or in the second case, n times, or in the third case, the value of b applied to the iterative result is 0.

Arguments

The operand f is any monad, and x is a valid argument of f. The argument n is a positive integer while b is a monad.

Definition

The function f is applied iteratively to x, as in:

```
x: f[x]
.
.
.
x: f[x]
```

The result is the last value of x or the next-to-last, depending on the case.

In the other cases of Over, the iteration results accumulate in the first argument, while the other arguments determine by their count the number of iterations. Over for a monad must provide other means for terminating the iterative process, which are as follows:

- the evaluation of `f/ x` ends when two successive iterative results match, or the result of an iteration matches that of the first iteration. The primitive function Match is used in the test, and therefore comparison tolerance is used for floating-point values. The result is the next-to-last value of x;
- the evaluation of `n f/ x` ends after n iterations (this case is sometimes called Do). The result is the last value of x;

- the evaluation of `b f/ x` ends when the result of `b[x]` matches 0 (this case is sometimes called While). The result is the last value of x, that is, the first value of x where `b[x]` matches 0. The primitive function Match is used in the test, but comparisons to 0 are never approximate, even when comparison tolerance is used (see Comparison Tolerance in the chapter Terminology).

An interesting example of the first of these three cases is the common phrase `,// x`. This idiom applies to any list x and produces a list of depth 1 whose items are the atoms in x. For example:

```
  ,// ("a";(1 2; `bc;("xyz"; 2.35)))
("a";1;2;`bc;"x";"y";"z";2.35)
```

The derived function `,//` is `(,/)/`. The comma in `,/` denotes the dyadic primitive function Join, and therefore `,/` is an instance of Over Dyad. Consequently `,/` denotes both a monad or dyad; which it is in `,//` cannot be seen from the immediate context (which is `,//`) and therefore the monad case is assumed (see Over Dyad). Consequently `,// x` is Over applied to the monad `{,/x}`. This monad joins all items of x into one list of depth one less than the depth of x unless the depth of x is 0 or 1, in which case its result equals x. Over Monad has the effect of applying this function repeatedly until there are no changes, that is, until the depth of the result is 0 or 1. See Scan Monad for a trace of the intermediate results of the iteration.

The expressions `|/` and `&/` were introduced in Over Dyad as Maximum and Minimum for finding the greatest and least value in a numeric vector. To apply these expressions to lists of greater depth, first use `,//` to collect all items of a list in a list of depth 1, and then apply `|/` or `&/`. That is, the general form of Maximum is `|/,//` and the general form of Minimum is `&/,//`. For example:

```
  |/,//(1;(2.3 25;(6 7 -9;10)))
25.0
  &/,//(1;(2.3 25;(6 7 -9;10)))
-9.0
```

Also see the examples in Scan Monad, particularly for Do and While. The last paragraph in the definition of Over Dyad applies equally to Over Monad.

Error Reports

Int Error if the argument n for Do is not a nonnegative integer.

Scan Dyad

```
f\ y
x f\ y
```

Description

Trace the iteration in Over Dyad.

Arguments

The operand f is a dyad, and the arguments x and y are any atoms or lists.

Definition

Scan Dyad f evaluates like Over Dyad f in all cases; under the same conditions where the iterative definition of Over-f applies, the result of Scan-f is a list whose items are the intermediate results, in order, of that iterative process (the successive values of x in the definition of Over Dyad f). In particular, the Over-f result is the last item of the Scan-f result. For example:

```
  +\ 1 3 5 7
1 4 9 16
```

The last two paragraphs in the definition of Over Dyad apply equally to Scan Dyad.

If y is an atom then `f\ y` is y and `x f\ y` is `f[x;y]` .

If y is a 1-item list then `f\ y` is `,y` .

If y is the empty list then so is `f\ y`, while `x f\ y` is `,x` . This differs from Over Dyad f, when the monad form `f/ ()` is defined only for + , `*` , `|` , and `&` .

Scan

`f\[x;y;…]`

Description

Trace the iteration in Over.

Arguments

The operand f is a function with at least two arguments. The relationship of the Scan-f arguments to the arguments of f is like that of Each-f to f except for the first argument x. That is, the number of objects between brackets equals the valence of f, and the arguments other than the first must all conform. The first argument x is any valid first argument of f.

Definition

Scan is to Over as Scan Dyad is to Over Dyad.

For dyad f, `f\[a;b]` is identical to `a f\ b` .

For functions f with valence at least three, the valence of `f\` equals the valence of f.

If all list arguments other than the first match the empty list the result is `,x` .

Error Reports

Length Error if any two arguments among y, z,... do not conform.

Scan Monad

```
 f\ x
 n f\ x
 b f\ x
```

Description

Trace the iteration in Over Monad.

Arguments

The operand f is any monad, and x is a valid argument of f. The argument n is a positive integer while b is a monad.

Definition

Scan Monad is to Over Monad as Scan Dyad is to Over Dyad.

The following examples could have been given in the section on Over Monad. The advantage of giving them here is that Scan reveals all intermediate results, making it easier to understand the iterative process. An important use of Scan is helping to set up the right function for Over.

```
  f:{:[x ! 2; x; _ x % 2]}
  f\ 5
,5
  f\ 12
12 6 3
  f\ 640640
640640 320320 160160 80080 40040 20020 10010 5005
```

The second and third cases provide ways for the user to specify conditions under which the iteration process is terminated. A nonnegative integer left argument specifies the actual number of iterations. This form is sometimes called Do with Trace:

```
  f\ 640640
640640 320320 160160 80080 40040 20020 10010 5005
  4 f\640640
640640 320320 160160 80080 40040
```

The first item is the initial value of the argument x, and the first four iterates follow.

If the left argument is a monad the iteration proceeds until the result of that function applied to an intermediate result matches 0. This form is sometimes called While with Trace:

```
  b:{x > 100000}
  b f\ 640640
640640 320320 160160 80080
```

Recall the expression `,//x` discussed in the section on Over Monad, which applies to any list and produces a list of depth 1 whose items are all the atoms in x. It may help to see how this function produces its results by evaluating the corresponding Scan expression `,/\x` and examining the intermediate results. The display of the result below has been edited to make comparisons of the intermediate results easier.

```
  ,/\("a";(1 2; `bc;("xyz"; 2.35)))
(("a";(1 2;`bc;("xyz";2.35)))         Argument
 ("a";1 2;`bc;("xyz";2.35))           1st iteration
 ("a";1;2;`bc;"xyz";2.35)             2nd iteration
 ("a";1;2;`bc;"x";"y";"z";2.35))      3rd iteration
```

Finally, the last paragraph in the definition of Over Monad applies equally to Scan Monad.

Error Reports

Int Error if the argument n for Do with Trace is not a nonnegative integer.

CHAPTER 6

AMEND, INDEX, APPLY & ASSIGN

All the constructs in this chapter, which are in terms of brackets and semicolons, have equivalent forms in terms of `.` and sometimes `@` , and the latter are sometimes more widely applicable. For example, the semicolons and brackets in the present constructs are syntax, so that the number of objects appearing between brackets is fixed at each occurrence. As a consequence, the Index expression `x[a;b;c]` will always access items of x three levels down, no matter how often it is evaluated. On the other hand, `x . p` will access items at level `#p` , which may vary from one evaluation to the next. However, the use of semicolons and brackets is convenient, and often easier to read than the other forms. For example, most function applications involve an unchanging function expression, and therefore a fixed number of arguments, so that the bracket form of Apply (e.g. `f[a;b;c]`) is appropriate. Bracket-semicolon constructs have the additional advantage of familiarity for most readers, and because of that are used in the definitions of the other forms in the preceding chapters.

You may notice in examples like `m:"3CAK342"` that the result is not displayed in the session log. All constructs in this chapter using `:` are treated specially by K for display purposes: whenever any such construct is the last one executed for an input line in an interactive session, its result is not displayed.

Amend

```
v : y
v f: y
v f:
v :: y
```

Description

Modify the object whose name appears in place of v with the function f and the atom or list y, whichever is present.

Arguments

The v on the left is a place holder for the name of a variable. If both f and y are present then f is dyadic, the variable must currently have a value that is a valid left argument of f, and y is any atom or list that is a valid right argument of f. If f is present but y is not then f is monadic, and again the variable must currently have a value that is a valid left argument of f. If f is not present then y is any atom or list.

Definition

In every use of Amend, the name of a variable appears in place of v. Say the name is b. In all cases the effect is to change the value of b and the result is the new value of b.

The first case associates the value of y with the name b, whether or not that name previously had a value.

```
  m                       assume m has not been given a value
  m:"3CAK342"             now give m a value
  m
"3CAK342"
  m:{x+y*z}               give m another value
  m
{x+y*z}
```

Any name can be given any value, and subsequent values need not conform in any way to previous values. This case of Amend is sometimes called Assign, but Amend is used here for all three cases in reference to amending the K-tree.

The effect of the second case is to replace the value of b with `f[b;y]`, and in the third case with `f[b]`. The explicit result is also the new value. For example:

```
  a: 1 2 3
  a +: 5
  a
6 7 8
  b: 3 2 # ! 6
  b
(0 1
 2 3
 4 5)
  r: b +:                b becomes flip b, as does the result r of b +:
  b
(0 1 2
 3 4 5)
  r ~ b
1
```

The last case is syntactically a special form of the second case, but its meaning is somewhat different. If b is a global variable and the expression `b :: y` appears in a function expression, the effect to assign the contents of y to the global variable b in the same directory as where the function is defined. (However, if the definition also contains an ordinary assignment `b: z` then b will be local).

Error Reports

Various errors can occur when an improperly formed named appears in place of v, as K attempts to parse the faulty expression.

Amend

```
v[j;k;…]: y
v[j;k;…] f: y
v[j;k;…] f:
v[]: y
v[] f: y
v[] f:
```

Description

In the first case replace the items-at-depth in the list whose name appears in place of v at paths specified by the index list `(j;k;…)` with the corresponding items-at-depth of y. In the next two cases modify those same items-at-depth with the function f and, if present, items-at-depth of y. In the last three cases replace or modify the list whose name appears in place of v itemwise with f and, if present, the atom or list y.

Arguments

The first argument v is a place holder for the name of an existing variable. The value of that variable is like the first argument of the verb form of Amend, except that a handle is not permitted. In the first three cases the list `(j;k;…)` is like the second argument of the verb form. The items of `(j;k;…)` must satisfy the same conditions as the verb form. The third argument f is any monadic or dyadic function; the second and fifth of the above expressions correspond to dyadic f and the third and sixth to monadic f. The argument y, if present, is any atom or list that conforms with the index list in the manner described under the verb form.

There must be at least one semicolon in the index list specification `[j;k;…]`, or no entries at all, or else this is Item Amend.

Definition

In every use of Amend, the name of an existing variable appears in place of v. The value of that variable serves as the first argument d in the verb form of Amend. In the first three cases the list `(j;k;…)` serves as the index list i in the verb form, and in the last three cases the empty space between brackets represents the Enlist of nil, and therefore these cases correspond to the verb form with the index list i equal to `,_n`. See Facts about Amend in the chapter Verbs for the behavior of Amend for this i.

In all cases the variable whose name appears in place of v is modified, and in that sense these forms of Amend are like the verb form with a handle first argument holding this name. However, the result of this form consists only of the new values of the modified or replaced items-at-depth; it is neither the handle nor the entire modified value of the verb form. For example:

```
  a:(1 2 3; 3 4 5)
  r: a[1 0;0 2] +: 100     r is the result of the Amend
  a
(101 2 103
 103 4 105)
  r
(103 105                   r has the modified items-at-depth of a
 101 103)
```

The last paragraph in the definition of the verb Amend applies equally to this form.

Error Reports

Index Error if any path in the path list `(j;k…)` is not a valid path of the left argument.

Length Error if the path list `(j;k;…)` and the last argument y , if present, are not conformable in the manner described under Amend in the Verbs chapter.

Length Error in the last three cases if v and y are lists with different counts.

Type Error if any atom of `(j;k;…)` is not an integer, symbol or nil.

Apply

```
f[j;k;…]
f[]
```

Description

Apply the function f to the argument list within brackets.

Arguments

f is a function and the objects within brackets are proper arguments of f.

Definition

`f[j;k]` evaluates dyadic f;

`f[j;k;l]` evaluates f with valence 3 (triadic), etc.

`f[]` evaluates niladic f.

(See Apply Monad for `f[j]` .)

For functions of valence at least two, when any of the argument positions are left blank, i.e. those arguments are unspecified, the effect is to project f onto the non-blank, specified arguments. For example:

```
  f:{x+y+z}
  f[1; 2; 3]
6
  g: f[1;; 3]                g is monadic, the projection of f onto
                             its first and third arguments
  g[2]                       apply g to the argument 2
6
```

When fewer argument positions than the valence of f appear between the brackets, say n argument positions and valence m, the last `m-n` arguments are considered to be unspecified and the function application is a projection onto the specified arguments among these n argument positions. Continuing the above example:

```
  h: f[1]                    one argument is specified, so h is a dyad
  h[2; 3]
6
```

```
  e: f[;2]                         two of three argument positions, one
                                   unspecified, makes e a dyad
  e[1; 3]
6
```

Note that if all argument positions are unspecified the resulting function is indistinguishable from f. See Projection; Fixing Function Arguments in the chapter Functions.

Error Reports

Valence Error if the function is called with too many arguments.

Execute

`d[s]`

Arguments

The argument d is any dictionary and the argument s is a character string.

Definition

`d[s]` is identical to `d @ s` (see Value / Execute in the chapter Verbs). Note that `. s` is identical to `_d[s]`.

Error Reports

See Item Index.

Index

```
d[j;k;…]
d[]
```

Description

In the first case select items-at-depth from the list or dictionary d, as given by the index list `(j;k;…)` . In the second case select all of d.

Argument

For the first expression, the argument d is like the left argument of the verb form of Index and the list `(j;k;…)` is like the right argument. The second case is like that verb form with d as left argument and nil as right argument.

There must be at least one semicolon in the index list `(j;k;…)` or this is Item Index.

Definition

`d[]` is identical to `d . _n` , or equivalently `d@!#d` for lists and `d@!d` for dictionaries;

`d[j;k]` is equivalent to `d .(j;k)` ;

`d[j;k;l]` is equivalent to `d .(j;k;l)` , and so on.

Facts about Index

`d[]` is a list of values of all entries in a dictionary d.

For atoms j and k, `d[j;k]` is identical to `d[j][k]` , and this extends to longer index lists.

Error Reports

Index Error if any atom in `(j;k;…)` is not a valid index to the item-at-depth in d.

Type Error if any atom of `(j;k;…)` is not an integer, symbol or nil.

Item Amend

```
v[i] : y
v[i] f: y
v[i] f:
```

Description

Modify the entries of the dictionary or items of the list whose name appears in place of v at indices i with f and, if present, the atom or list y.

Arguments

The first argument v is a place holder for the name of an existing variable. The value of that variable is like the first argument of the verb form of Amend Item, except that a handle is not permitted. The second argument i is either a nonnegative integer atom or list, or a symbol atom or list. The argument f is any verb; the second of the above expressions corresponds to the dyadic case of f and the third to the monadic case. The argument y, if present, is any atom or list; i and y must be conformable in the sense described for the verb form of Amend Item, and if f is present, items-at-depth in y corresponding to paths in i must be valid right arguments of f.

Definition

In every use of Item Amend, the name of an existing object appears in place of v. This name must be a valid entry in the current directory (see the introductory remarks to this chapter). The value of that variable serves as the first argument d in the verb form of Amend Item. The remaining arguments i, f, and y serve in their same roles as in the verb form. For example:

```
  a: 3 4#!12
  a
(0 1 2 3
 4 5 6 7
 8 9 10 11)
  a[1] : (`ab;"cde")          replace item 1 with (`ab;"cde")
  a
(0 1 2 3
 (`ab
  "cde")
  8 9 10 11)
  a[0] +: 10                add 10 to item 0
  a
(10 11 12 13
 (`ab
  "cde")
 8 9 10 11)
  a[2] -:                   negate item 2
  a
(10 11 12 13
 (`ab
  "cde")
-8 -9 -10 -11)
```

In all cases the variable whose name appears in place of v is modified, and in that sense these forms of Item Amend are like the verb form with a handle first argument holding this name. However, the result of this form consists only of the new values of the modified or replaced items; it is neither the handle nor the entire modified value of the verb form. For example:

```
  a: 1 2 3 4
  r: a[0 2] +: 100      r is the result of the Item Amend
  a
101 2 103 4             the new a
  r
101 103                 r holds only the changed items
```

The last paragraph in the definition of the previous form of Amend applies equally to this form of Amend.

Error Reports

Index Error if any atom of the index i is not a valid index of the object being modified.

Length Error if the index i and the last argument y are not conformable.

Type Error if any atom of the index i is not an integer, symbol or nil.

Item Index

```
d[i]
```

Arguments

The argument d is any list and the argument i is a nonnegative integer atom or list, or the argument d is any dictionary and the argument i is a symbol atom or list.

Definition

`d[i]` is identical to `d @ i`. See Index Item in the chapter Verbs.

Error Reports

Index Error if any atom of i is not a valid index of d.

Rank Error if d is an atom.

Type Error if any atom of the index i is not an integer, symbol or nil (or character string; see Execute).

Apply Monad

```
f[i]
```

Arguments

f is a monadic function and i is a proper argument of f.

Definition

`f[i]` evaluates the monad f with argument i.

CHAPTER 7

FUNCTIONS

A function can be defined by entering its defining expressions in order from left to right separated by semicolons, with a left brace (`{`) to the left of the first expression and a right brace (`}`) to the right of the last expression. The first expression may be preceded by a bracketed list of names, as in `[n1;n2;…;n3]`, which is the argument list when the function is applied. For example, if the function `interest` is defined as follows:

```
interest: { [p;r;t] p * r * t }
```

then when it is called it will have three arguments, which are the values for `p`, `r` and `t`, in that order, as in `interest[100;0.075;1]` . An empty argument list, i.e. `[]`, is used for a niladic function, i.e. one with no arguments. If there is no argument list then the following default arguments are assumed:

- if the name z appears in the expression then the argument list `[x;y;z]` is assumed, no matter whether x or y appears or not;
- if y is present but z is not, the argument list `[x;y]` is assumed;
- if x is present but y and z are not, the argument list `[x]` is assumed;
- if x, y and z are not present, the argument list `[]` is assumed.

The value of the right-most expression is the default explicit result of the function (see Return in the chapter Controls and Debugging for overriding the default). Functions always have explicit results. When there is nothing between the right-most semicolon and the right brace, or just blank space, the result of the function is nil.

For example, the result of `f[]` for the nilad `f:{a: 10; b: 20; a + b}` will be 30 because the default result line is `a + b`, but the result of `g[]` for the nilad `g:{a: 10; b: 20; a + b;}` will be nil because the result line is empty.

Projection; Fixing Function Arguments

Suppose that at a certain point in an application the function Plus is always called with the same fixed left argument, as in `3+a-b`. This occurrence can be viewed as the monadic function `3+x` applied to `a-b`. The expression `3+` denotes that monadic function, and the above expression can be written as `(3+)[a-b]`. It is said of the sub-expression `(3+)` in `(3+)[a-b]` that the left argument of Plus is *fixed*, or that 3 is the fixed left argument of Plus. In addition, the monadic function `3+` is called a *projection of* `+` *onto the left argument*.

It is not necessary for fixed arguments to be constant, but in most uses they tend to be fixed relative to other arguments. For example, consider an application of the adverb Each Right to a dyadic function f, i.e. `x f/:y`. When evaluated, the function f is applied `#y` times, each time with the same left argument x but a different right argument, the ith item `y[i]` for every index i. The value of x may change from one evaluation of `x f/:y` to the next, but during any particular evaluation f is applied with a fixed left argument x and a varying right argument. Note that the functionality of `x f/:y` can be obtained by fixing the left argument x to produce a monadic function, which is expressed as `f[x;]`, and applying Each to that monadic function with argument y, as in `f[x;]'y`. That is, `x f/:y` is `f[x;]'y`.

The previous example suggests how Each Right and Each Left can be generalized to functions of three or more arguments so that Each is applied item-wise to some arguments but not others. For example, if f has four arguments a, b, c and d and f is to be applied to all of a and c with each item of b and d, then fix arguments a and c and apply Each to the resulting dyadic function, as in either of the following two expressions:

```
b f[a;;c;]'d
f[a;;c;]'[b;d]
```

It is possible to fix arguments using function expressions, but less convenient. The first and third arguments of the above function f are fixed in the expression `{f[a;x;c;y]}`, and the above application of Each can now be written as either of the following:

```
  b{f[a;x;c;y]}'d
  {f[a;x;c;y]}'[b;d]
```

It is not possible to fix the right argument of Plus in the way that the left argument was fixed above. That is, `(3 +)` is valid, but `(+ 3)` is not. However, `+[3;]` also expresses fixing the left argument, and the analogous expression for the right argument, `+[;3]` , is a valid expression for fixing the right argument.

When a function with fixed arguments is assigned to a name, the fixed arguments maintain their current values in the object named on the left. For example, suppose:

```
  A: 3                 A is assigned 3
```

and f is defined as follows:

```
  f: A +               f is assigned Plus with fixed left argument
  f 5                  evaluate f 5
8
  A: 6                 change A
  f 5
8                      f is unchanged; it does not use the new value of A
  (A +) 5
11                     the fixed left argument in A +  is the new value of A
```

Just as when the adverb Over is applied to f and the resulting function is called f-Over, a projection of f is called f-Onto, as in f-Onto the second and fourth arguments for `f[;a;;b]`. See Apply in the chapter Amend, Index, Apply and Assign.

Localization

A name that begins with a dot is called an *absolute referent* , and one that begins with an alphabetic character is called a *relative referent* .

All relative referents assigned a value via single colon Amend of the form `a:b` in a function expression are local names. The names of the arguments are also local. In fact, local names are strictly local, in that their values cannot be seen outside the function expression or inside any function called within the function expression.

Use double colon Amend or the handle case of Amend, Amend Item or Item Amend to assign global variables with relative referents within function expressions. In the case of double colon assignment, the relative referent will be resolved in the directory that was the current directory when the function expression was defined. In the cases where a handle is used, the relative referent that is the contents of the handle will be resolved at run-time, in the directory that is current when the function is called.

Global variables identified by absolute referents can be assigned values by any of the various assignment methods within function expressions, i.e. single and double colon Amend and any of the handle cases.

Local Functions

Suppose that the function g is defined within the body of another function f and uses the variable x in its definition, where x is local to f. Then x is a constant in g, not a variable, and its value is the current one when g is defined. For example, if:

```
  f:{b:3; g:{b}; b:4; g[]}
```

The value of f is the value of the local function g, which turns out to be 3, the value of b when g is defined, not the subsequent value 4.

```
  f[]
3
```

CHAPTER 8

ATTRIBUTES

An attribute is a global variable with a special association to another global variable. Attributes are either *primitive* , i.e. part of the K language definition, or user-defined. The association between a variable and one of its attributes is expressed in their names. For example, for the variable named v its format attribute is named `v..f`.

The *attribute dictionary* of v , denoted by `v.` , contains all attributes of the variable v. Since attributes themselves are true variables they also have attributes, and these in turn have attributes themselves, and so on. In practice, however, it is rare to go beyond attributes of attributes of ordinary variables. And attribute dictionaries, which are also true variables, do not themselves have attributes.

The name of an attribute of a variable is formed by the variable name, followed by a dot to signify the attribute dictionary of the variable, followed by another dot to signify an entry in that dictionary, followed by the attribute name. For example, `v..f` is the format attribute of v.

Primitive attributes have special effects on the variables they modify defined by the K language (see the definitions below), while user-defined attributes have user-defined meanings. Typically, a user-defined attribute would be auxiliary information about a directory that organizationally does not belong among the directory entries. For example, a directory might represent a view of a relational table with its entries as fields, and the name of the view's base table(s) could be a user-defined attribute of the view.

Primitive attributes have implicit default values, but not explicit ones. That is, referencing an attribute for a variable when that attribute has not been given a value will not necessarily give the default value.

Arrangement

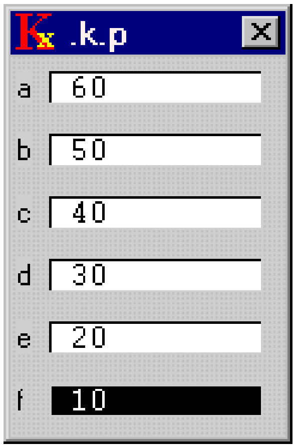
`v..a`

If v is a dictionary that is classified as `` `form``, then `v..a` is a symbol list whose atoms are entries in v, such that the arrangement of the atoms in `v..a` specifies the arrangement of the entries on the screen when v is displayed. The depths of the atoms in `v..a` indicate whether the corresponding entries are within vertical or horizontal sections of the display. For example:

```
\d p                    create a dictionary p, and then create six entries
a: 10 + b: 10 + c: 10 + d: 10 + e: 10 + f: 10
\d ^                    go up one level
p..c: `form             make p a form
`show $ `p              display it
```

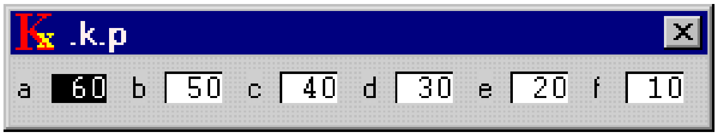
At this point the entries of p are displayed vertically in what looks like random order. (Actually, it is the order in which the variables were created.) To display them vertically in the order a through f, set `p..a` to `` `a`b`c`d`e`f``. To display them horizontally in that order, set `l..a` to `` ,`a`b`c`d`e`f``. The top-level of `p..a` indicates vertical order (one list item), and the next level horizontal (six items). To display a and b horizontally above c, d, e, and f horizontally, set `p..a` to ``(`a`b;`c`d`e`f);`` the top level of `p..a` indicates vertical order (two items) and the next level horizontal (two items in one and four in the other). And so on. The following figures show various arrangements of this form.

```
p..a:`a`b`c`d`e`f          p..a:,`a`b`c`d`e`f
```



.k.p

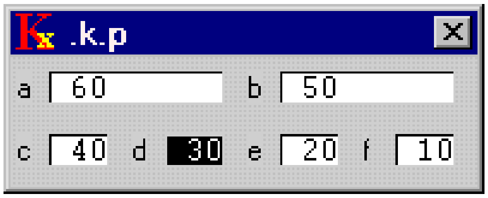
a 60
b 50
c 40
d 30
e 20
f 10



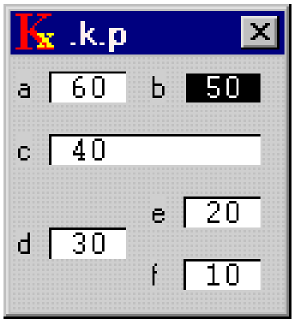
.k.p

a 60 b 50 c 40 d 30 e 20 f 10

```
p..a:(`a`b;`c`d`e`f)
```



```
p..a:(`a`b;`c;(`d;`e`f))
```



Background Color / Foreground Color

```
v..bg
v..fg
```

These attributes specify the background and foreground colors on the object v. The foreground color applies to display of the object's data, while the background color applies to the region in which the data is displayed. Distinct colors are specified as nonnegative integer atoms with at most 6 meaningful digits, arranged as `rrggbb`. That is, if c is such an integer atom then

```
  100 100 100 _vs c
```

consists of three integers with values between 0 and 99 that depend only on the right-most six digits of c. The first of the three integers specifies the intensity of red in the resulting color, from none (0) to maximum intensity (99), the second specifies the intensity of green, and the third, blue. For example, 990000 is pure red, 9900 is pure green, 99 is pure blue, and 990099 is purple formed from equal intensities of red and blue. Also, 0 is black and 999999 is white.

Any integer `c` can specify a color, and if its value is not between 0 and 999999 then the residue `c!1000000` is used. For example, white can be specified as `-1`.

The value of the color attribute can be an integer atom, which applies to every data item on the screen, or an integer list, with (possibly) a different color value for each data item, or a monadic function, whose argument is the value of the data item to be colored, or a niladic function. In the case of a function, the system variables `_v` and `_i` are available. These functions have access to any entry in the K-tree, but cannot modify the value of v.

Class

```
v..c
```

This attribute classifies the format of the display of the associated variable v . The meaningful values are `` `form ``, `` `data ``, `` `chart ``, `` `plot ``, `` `button ``, `` `check ``, and `` `radio `` . The default is `` `data `` , with the following exception: the default display class of a dictionary is `` `form `` if the display class of any entry has been specified, or if some entry is a dictionary whose default display class is `` `form `` . See the chapter Screen Displays for examples of the display classes.

Click / Double Click

```
v..k
v..kk
```

A mouse click event is said to occur on v when a mouse button is pressed while the mouse cursor is in the data area of the screen display of v. A double click event occurs when a second mouse click event on v occurs almost immediately after the first. The click and double click attributes are character strings holding expressions that are executed whenever click and double click events occur. Note that the click attribute expression is executed on the first click in a double click sequence, but not on the second click.

Dependency

```
v..d
```

The value of this attribute is a character vector holding the dependency definition of the associated variable v . See the topic Dependencies in the chapter Terminology . In principle, a dependency should not explicitly set the values of items on the K-tree, but if v happens to be set while the value of `v..d` is executing, that setting will not cause `v..d` to execute again. Cycles in dependency relations are not permitted, i.e. a variable cannot be dependent on itself.

Editable

```
v..e
```

This attribute specifies whether or not the data items of v on the screen can be edited, with a 0 for no and a 1 for yes. The value of this attribute can be an integer atom, which applies to every data item on the screen, or an integer list, with (possibly) a different editable setting for each data item, or a monadic function, whose argument is the value of the data item to be edited, or a niladic function. In the case of a function, the system variables `_v` and `_i` are available and the result indicates whether or not the `_i`th item (or item-at-depth) of the variable on the K-tree named in `_v` can be edited. These functions have access to any entry in the K-tree, but cannot modify the value of v.

Format

```
v..f
```

The value of this attribute is a monadic function used to format the items or items-at-depth in the screen display of the associated variable v . The result of the function should be a character vector. Typically, the function is a projection of the dyadic Format primitive function onto a fixed left argument, as in:

```
v..f: 8.2 $
```

or a more specialized formatting function, say for dates and time.

Help

```
v..h
```

The value of this attribute is a character vector holding descriptive information on the associated variable v , typically for the user of an application. Eventually it will be automatically displayed when the screen display of v has focus and some to-be-defined action is taken.

Label

`v..l`

The value of this attribute is a character vector. For certain display formats, in the absence of a value for this attribute the name of the associated variable is used to label the object; for example, see the earlier figure showing various arrangements of a form under the Arrangement attribute, where the names a through f are used. If this attribute has a value then the text in the character vector is used as the label.

Option List

`v..o`

Option Lists are symbol vectors that are associated with the radio display class. If v is classified as `` `radio `` then its display has n entries, where n is the count of `v..o`, and the label on the ith radio button is the text in the ith item of `v..o`. The value of v must be one of the entries in `v..o`; when the ith radio button is pressed the value of v becomes `v..o[i]`.

Trigger

`v..t`

The value of this attribute is a character vector holding the trigger definition of the associated variable v. See the topic Triggers in the chapter Terminology. The purpose of triggers is to have side effects; a value produced by a trigger expression is ignored. Note that if the value of v happens to be set while the trigger `v..f` is executing, that set will not cause `v..f` to execute again. Neither will a set of v due to the evaluation of its dependency expression `v..d`.

Update

`v..u`

The value of this attribute is a dyadic function used to produce the new value of v when v is modified on the screen. When an item of v is changed on the screen, the validation function (see Validation below) is first called by k to produce the potential new value — say y — of that item from the character string on the screen, and then v is amended by k as follows:

```
.[` v; _i; v..u; y]
```

In particular, the first argument of this attribute is the old value, and the second argument is the new value.

An update function checks that the proposed value is valid, modifies that value or sets other variables on the K-tree as appropriate, and if the proposed value is invalid can signal an error (see Signal in the chapter Controls and Debugging) to abort the update.

When an update function is called, the left argument holds the value to be replaced and the right argument holds the proposed new value. The system variables `_v` and `_i` are available, indicating which item `_i` (or item-at-depth) of which variable `_v` on the K-tree is to be updated.

The default value of `v..u` is ordinary assignment, i.e. in the case when `v..u` is not specified, the amending expression is:

```
.[` v; _i; :; y]
```

Everything done by an update function can also be done by a validation function, but it is often useful to separate the activity of undoing the formatted character string on the screen (validation) from updating the value of a variable, particularly when other settings of the variable must also be monitored. Any explicit setting, i.e. one that appears explicitly in the code, can be done using the same update function in an explicit amend expression, as in:

```
.[` v; i; v..u; y]
```

Validation

`v..g`

The value of this attribute is a monadic function that is essentially an inverse to the monadic function in the Format attribute of the associated variable v. Its argument is a character vector and its result is an appropriate item or item-at-depth in v. When an item on the screen is modified this function is called with the contents of that item as a character vector. A validation function should simply return the appropriate value represented by the character string if the representation is valid, and otherwise signal an error (see Signal in the chapter Controls and Debugging) to abort the change.

The default value of the validation attribute is the dyadic primitive function Form with the left argument appropriate to the data type of v.

Width / Height

```
v..x
v..y
```

The values of these attributes are the minimum width and height of the object v on the screen, where the units of length are, respectively, the average character width and average character height of the data font (see the operating system appendices in the K User Manual for the ways in which the data font can be specified). The settings of these attributes on a form will be ignored if they are inconsistent with the sizes of the entries in the form.

CHAPTER 9

CONDITIONALS

Conditional Evaluation

```
:[cond; true; false]
:[cond1;true1; cond2;true2; …; condN;trueN; false]
```

The first expression is the simplest form of conditional evaluation. All three of `cond` , `true` , and `false` denote expressions. If the result of `cond` is a non-zero integer, the conditional evaluation is the result of `true` , but if the result of `cond` is 0, the conditional evaluation is the result of `false` .

The second expression above denotes the general form of conditional evaluation. If the result of `cond1` is a non-zero integer, the conditional evaluation is the result of `true1` . Otherwise, if the result of `cond2` is a non-zero integer, the conditional evaluation is the result of `true2` . And so on, until finally, if the result of `condN` is a non-zero integer, the conditional evaluation is the result of `trueN`, but if the result of `condN` is 0, the conditional evaluation is the result of `false`.

Note that the meanings of the expressions in a conditional evaluation alternate between conditions and conditionally executed expressions. The result of a conditional evaluation is the result of the conditionally executed expression whose condition is true, or nil if all conditions are false.

Error Reports

Type Error if any of the condition statements do not have integer atom values.

Do

```
do[count; expression]
do[count; expression1;…; expressionN]
```

The effect of the first Do statement is to execute the expression `count` times, while in the second case the expression list is executed `count` times. For example:

```
  i:0
  do[5; i+: 1]
  i
5
```

As with function expressions and conditional evaluation, a list of expressions is executed from left-to-right, or top-to-bottom if the expressions are on separate lines. For example, the following Do statements are equivalent:

```
  do[n; f[x;y]; g[x;y;z]]
```

and

```
  do[n
     f[x;y]
     g[x;y;z]]
```

Do statements do not have results, or rather, their results are always nil.

Error Reports

Type Error if the `count` statement does not have an integer atom value.

If

```
if[condition; expression]
if[condition; expression1;…; expressionN]
```

The effect of the If statement is to execute expressions conditionally depending on the value of the condition expression. In the first case, *expression* is executed if the value of the condition expression is not equal to 0. In the second case, the expression list is executed. For example:

```
  a: 10                        a: 10
  if[3<4; a: 20]               if[3>4; a: 20]
  a                            a
20                           10
```

Like the Do statement, a list of expressions is executed from left-to-right, or top-to-bottom if the expressions are on separate lines. The If statements do not have results, or rather, their results are always nil.

Error Reports

Type Error if the condition statement does not have an integer atom value.

While

```
while[condition; expression]
while[condition; expression1;…; expressionN]
```

The effect of the first While statement is to repeatedly execute the expression as long as the condition has a nonzero value, while in the second case the expression list is executed repeatedly. For example:

```
  i:0
  while[ 5 > i; i+: 1]
  i
5
```

Like the Do statement, a list of expressions is executed from left-to-right, or top-to-bottom if the expressions are on separate lines. The While statements do not have results, or rather, their results are always nil.

Error Reports

Type Error if the condition statement does not have an integer atom value.

CHAPTER 10

CONTROLS AND DEBUGGING

The examples in this chapter assume that the error flag is set to 1, i.e.

```
  \e 1
```

See Error Flag in the chapter Commands.

Abort

```
\
```

Abort causes the most recent suspended execution to be abandoned and its execution stack to be cleared. For example:

```
  5 * 3 + "a"
type error
3 + "a"
  ^
>  \                     Prompt indicates a suspension; abort with \
  3 + 97                 The prompt indicates no suspension; continue
```

Each `\` clears one suspension; it is not possible to abort more than one suspension at a time. For example, if a second error had occurred above, the situation would have been:

```
>>  \                    Abort the most recent suspension
>  \                     One suspension remains; abort it
                         No suspensions remain
```

Abort can also be used to escape early from either loading or step-loading a K script (see Load and Step in the chapter Commands).

Comment

```
/text
```

A comment may be placed at the end of a line by entering the slash character followed by the text of the comment. The slash character must be preceded by a space (or be the first character on a line) in order to denote a comment. Otherwise, K will assume it denotes Over, or perhaps is part of Each Right. For example:

```
  15 * 17 / This should be 255
255
/And it is!
```

Resume

```
:
: x
```

Execution is suspended when a primitive function fails or a Stop control statement is executed (see Stop / Trace). To resume execution after a failed primitive, enter colon (`:`) followed by a value to serve as the value of that failed primitive. For example:

```
  4 # 3 + "a"
type error
4 # 3 + "a"
      ^
>  : 3 + 97
100 100 100 100
```

A colon was entered on the next-to-last line, followed by the expression `3+97` , and execution resumed with 100 as the value of the failed `+`. That value became the right argument to `#` (with left argument 4), resulting in `100 100 100 100` .

If execution is suspended because of a Stop control statement, nothing has failed; it is not necessary to supply a value in order to resume. In this case simply enter colon alone on the input line to resume execution.

Return

```
: x
```

When this expression is executed in the course of executing a function expression, the effect is to end execution of that function and return with the result x. It is most often used with If and Conditional Evaluation (see these topics in the chapter Conditionals).

Signal

```
'
' x
```

The effect of Signal is to cause a K-like error from within a function expression, or to signal one level up the execution stack in immediate execution mode.

Only `'x` , and not quote alone, can be used in a function expression. When executed, the effect is to cause a K-like error to be reported in the K session, as follows:

- x is a symbol or character vector containing the text of the error report;
- the function in which the signal occurred is treated like a primitive, in that the line with the function application is displayed, not the line within the function where `'x` occurs;
- the up-caret (^) is placed beneath the beginning of the function expression, or the beginning of the function name, whichever appears in the displayed line;
- execution is suspended and the prompt > is displayed;
- Resume (See Resume) can be used to resume execution, just as with a failed primitive.

For example, here is a K error report for a failed primitive:

```
  5 * -4.0 ^ 0.5
domain error
5 * -4.0 ^ 0.5
         ^
>  : 4.0 ^ 0.5              Most likely,  -4 should have been  4
10
```

Compare this to an error report for a failed function expression:

```
  SQRT:{:[ ~ x < 0; x ^ 0.5; ' `"domain error"]}
  5 * SQRT -4.0
domain error
5 * SQRT -4.0
    ^
>
```

Resume with the value `SQRT 4.0`:

```
>  : SQRT 4.0
10
```

The value of `SQRT 4.0` was used as the right argument to Times (with left argument 5) when execution resumed.

Both forms of Signal are valid in immediate execution mode. For example, suppose the `SQRT` function had not signalled, but had simply been defined as:

```
  SQRT:{x ^ 0.5}
```

and the same expression as above had been executed:

```
  5 * SQRT -4.0
domain error
{ x ^ 0.5}
    ^
>
```

At this point Resume could be used to continue execution from the point of error. However, in debugging it is often helpful to see the actual application of the function, which does not appear in the error report. This can be accomplished by signalling, which has the effect of passing the error up one level of the execution stack. Continuing the example:

```
>  ' "x^0.5 error"
x^0.5 error
5 * SQRT -4
    ^
>
```

Execution can now be resumed from this point, or abandoned with Abort, or Signal can be used again to signal up another level. If `SQRT` was a more complicated function, then generating an error message like `"x^0.5 error"` to indicate the reason for the suspension might be useful in the session log. However, it is often not necessary, and all one wants to do is signal one level up. This can be done with quote alone (there is no expression to its right), which has the same effect as `' ""` or `' `` ` ``, i.e. signalling with an empty error message.

Stop / Trace

```
\ x
```

When `\` is placed to the left of an expression in a function expression, as if it were a monadic function to be applied to the value of that expression, there are two possible effects: immediately after the expression is evaluated during function execution, either its value is displayed or execution of the function is halted. When values are displayed the effect is called a *trace* , and when execution is halted it is called a *stop* . Whether back-slashes cause traces or stops is controlled by the execution monitor command `\b` , also called the break flag. It is also possible to turn off stops and traces with this same command, and without removing the back-slashes.

CHAPTER 11

I/O AND COMMUNICATION

The definitions of the primitive functions `3:` and `4:` describe the default behavior of interprocess communication among k processes. See the section Interprocess Communication at the end of this chapter for how the default behavior can be modified.

Load / Save Text File

```
0: f
f 0: x
` 0: x
```

Description

Load text files as string vectors, and save such string vectors as text files.

Arguments

The argument f is a symbol atom, character atom or character string, while x is a character atom or string, or string vector.

Definition

In either of the first two cases f holds the name of a file, including any directory path information. The monadic function `0:f` loads the file named in f into the workspace as a string vector. Each byte in the file becomes a character in the result, except for new-line characters, which have the effect of separating one character string from the next.

The dyadic function `f 0: x` saves the string vector x in the file named in f, where `x[0]` becomes the first line in the file, `x[1]` the second, and so on. New-line characters are inserted between successive character vectors. Blanks at the end of character strings are preserved. The result is always nil.

There is a special case of the dyadic function where f is the empty symbol, in which case the contents of what would otherwise have been written to a file are simply displayed in the session log.

Error Reports

Domain Error if the file named in f does not exist or cannot be found, in which case an operating system message may be displayed as well.

Type Error if either f or x are not as described above.

Load Text File as Fields

```
(s;w) 0: f
(s;w) 0: (f;b;n)
```

Arguments

The right argument f is a symbol atom, character atom or character vector, while the left argument is a two-item list whose first item s is a character vector and second item w is an integer vector. The two vectors have the same count.

In the second case, the right argument is a three-item list where f is as described above, and b and n are integers.

Definition

The argument f holds the name of a text file, including any directory path information. Each line of this file is presumably a record of fixed-width fields. The purpose of this function is to load the file into the workspace in such a way that each row is partitioned into fields of widths and types specified by the left argument. The result is a matrix whose items have count equal to the number of rows in the file, and whose ith item holds the contents of the ith field.

The left argument `(s;w)` specifies how the rows are to be partitioned. The first item is a character vector and the second item is an integer vector. The two vectors have the same count, which is the number of logical fields. The ith items of the vectors describe the ith field. Each item in the character vector s is one of `"IFCS "`, where:

- `I` means integer field, which comes into K as an integer atom;
- `F` means floating-point, which comes in as a floating-point atom;
- `C` means character, which comes in as a character vector;
- `S` means symbol, which comes in as a symbol atom;
- blank means skip.

The ith item of the integer vector is the width (in characters) of the ith field.

For example, suppose the file named in f has the following two rows:

```
1050    1.234   abcdef  ghi
234     1e50     gqw    xy
```

where there is a space at the end of the second row. Then:

```
  ("IFCS"; 7 9 10 3) 0: f
(1050 234
 1.234 1e+50
 ("abcdef    "
  " gqw      ")
 `ghi `xy)
```

Note the extra padding on the two character vectors. These could be loaded using width 6 to avoid unnecessary padding, and skipping the 4 blanks that follow:

```
  ("IFC S"; 7 9 6 4 3) 0: f
(1050 234
 1.234 1e+50
 ("abcdef"
  " gqw  ")
 `ghi `xy)
```

The second form of dyadic `0:` permits more flexible loading of exceptionally large files. In this case, f is a file as previously described, b is an offset into that file, and n is a length. Both b and n are in units of bytes.

Error Reports

Domain Error if the file named in f does not exist or cannot be found, in which case an operating system message may be displayed as well.

Domain Error if the contents of a field specified as integer or floating-point field are not a valid representation of that type of number.

Length Error if the length of the lines in the file are not all equal to the total of the field widths in s, i.e. `+/s[1]`.

Load/Save K Data as K Files

```
1: f
f 1: x
```

Arguments

The argument f is a symbol atom or character vector, while x is any atom or list. In the first case, Load K Data (monadic `1:`), f may also be a character atom.

Definition

In either case f holds the name of a file, including any directory path information. The monadic function `1: f` loads that file into the workspace as a data object, which is the result of the function. The dyadic function `f 1: x` saves the data x in a file in the format of a K data object. Presumably the monadic function is applied to a file that has been previously created with the dyadic function.

Note: The actual name of the file referred to by f may differ from the name held in f, typically by the extent `.K` on Unix systems and by `.L` on NT. For example, if f is `"/dir/prices"` then the actual file name would be `/dir/prices.K` on Unix, or `/dir/prices.L` on NT. The full file name including the suffix is permitted for f, but is not portable.

Error Reports

Domain Error if the file named in f does not exist or cannot be found, in which case an operating system message may be displayed as well.

Nonce Error if the file indicated by f exists, but does not contain valid K data.

Type Error if either f or x is not as described above.

Load Binary File as Fields

`(s;w) 1: f`
`(s;w) 1: (f;b;n)`
`c 1: f` or `c 1: (f;b;n)`

Arguments

The right argument f is a symbol atom, character atom or character vector, while the left argument is a two-item list whose first item s is a character vector and second item w is an integer vector. The two vectors have the same count.

In the second case, the right argument is a three-item list where f is as described above, and b and n are integers. In the third and fourth cases, c is a character. Only the options `"c"` or `"i"` or `"d"` for c are allowed.

Definition

The argument f holds the name of a binary file, including any directory path information. The length of the file, L, must be an integral multiple of W, the sum of the field widths specified in the second item of the left argument. The purpose of this function is to load the file into the session as a matrix whose items have count equal to `L%W`, and whose ith item holds the contents of the ith field.

The left argument specifies how the file is to be read. The first item s is a character vector whose count is the number of logical fields, and whose items are one of `"cbsifd CS"` , each of which refers to a C language data type. Specifically:

- `c` means one-byte character field, which comes into K as a character atom;
- `b` means one-byte integer field, which comes into K as an integer atom;
- `s` means two-byte integer field, which comes into K as an integer atom;
- `i` means four-byte integer field, which comes into K as an integer atom;
- `f` means four-byte floating-point field, which comes into K as a floating-point atom;
- `d` means eight-byte double, which comes in as a floating-point atom;
- blank means skip;
- `C` means string, which comes into K as a character string;
- `S` means string, which comes in as a symbol atom.

The second item w is an integer vector of the same count as the character vector, whose items are the widths of the fields. See the example in Load Text Files as Fields.

In the special case of the left argument `"c"`, the entire file (including any null bytes) is read into a character string whose length equals the number of bytes in the file. If the left argument is `"i"` (or `"d"`) the entire file is read into an integer vector (or floating-point vector) with 4 (or 8) bytes to the item; the length of the file must be a multiple of 4 (or 8).

The second form of dyadic `1:` permits more flexible loading of exceptionally large files. In this case, f is a file as previously described, b is an offset into that file, and n is a length. Both b and n are in units of bytes.

Error Reports

Domain Error if the file named in f does not exist or cannot be found, in which case an operating system message may be displayed as well.

Domain Error if the contents of a floating-point field are not a valid representation of that type of number.

Length Error if the length of the file is not an integral multiple of the sum of the field widths in s, i.e. `+/s[1]`.

Copy K Data from K File

`2: f`

Arguments

The argument f is a symbol atom, character atom or character vector.

Definition

The argument f holds the name of a file, including any directory path information. Presumably f has been previously created with the Save K Data variation of dyadic `1:`. Monadic `2:` copies f into the workspace as a data object, which is the result of the function. The significant difference between this function and monadic `1:` (Load K Data) is that in the latter case the file is actually mapped rather than copied, and therefore certain operations on the contained data are restricted. See also the section Load/Save K Data as K Files.

Note: The actual name of the file referred to by f may differ from the name held in f, typically by the extent `.K` on Unix systems and by `.L` on NT. For example, if f is `"/dir/prices"` then the actual file name would be `/dir/prices.K` on Unix, or `/dir/prices.L` on NT. The full file name including the suffix is permitted for f, but is not portable.

Error Reports

Domain Error if the file named in f does not exist or canno t be found, in which case an operating system message may be displayed as well.

Nonce Error if the file indicated by f exists, but does not contain valid K data.

Link Object Code

```
f 2: (e;t)
```

Arguments

The argument f is a character vector, e is a character vector, and t is an integer.

Description

Use compiled code like defined functions.

Definition

This function links the object file named in f into the K process. The name of this file and the way it is created varies with the host operating system; see the K User Manual. The character string e holds the name of a function in the object file to be made into a K function. The result is that K function, which is usually assigned to a name in the K session for future use. The non-negative integer t is the number of formal parameters declared for the external function. The valence of the resulting K function is also t.

The external function described by string e must have return type and parameters which conform to the internal K data type, as specified in the interface description appropriate to the programming language and operating system in which the function is written and compiled.

Error Reports

Domain Error if the file named in f does not exist or cannot be found, or if the function or module e cannot be located within f. In these cases, an operating system message may be displayed as well.

Type Error if any of f, e and t are not as described above.

Communication Handle

`3: (n;p)`

Arguments

The first item of the argument, n, is a symbol atom and the second item, p, is an integer atom.

Description

Identify a communication partner.

Definition

The symbol n holds the name of the machine and the integer p is the communication port number of another K process, presumably one with which this one wants to establish communication. The result is an integer atom, which is the communication handle to be used as a left argument to Remote Get and Remote Set. (The pair `(n;p)` can also be used as a left argument in these cases.)

The symbol n may be the empty symbol `` ` ``, in which case the machine is assumed to be the local host. In other words, the communication partner to be identified is a K process running on the same machine. Four number internet protocol (IP) addresses in the form `` `"999.999.999.999" `` may also be used for n.

Whenever a message is received from a partner, the system variable `_w` holds the communication handle of that partner. Consequently, only the partner who initiates the communication may need to use this function: if the first message is a remote set of a global variable which has a trigger, the trigger can inspect `_w` to get the communication handle of the new partner.

The communication port number p is established in various ways, depending on the host operating system. See the section Interprocess Communication and the K User Manual.

Error Reports

Domain Error if the partner identified by the pair `(n;p)` is not available.

Type Error if n or p is not as described as above.

Close Handle

```
3: h
```

Arguments

The argument h is an integer atom.

Description

End communication with this partner.

Definition

The argument h is a communication handle, e.g. the result of `3:(n;p)` or a value of `_w`. The effect is to close this communication channel.

Error Reports

Domain Error if the argument is not an active communication handle.

Type Error if h is not an integer atom.

Remote Set

`t 3: x`

Arguments

The argument t, which identifies another K process, is described in Communication Handle. The symbol ` is also allowed for t. The argument x can be any atom or list.

Description

Set a value in another process.

Definition

Assume for now that t is not `. For convenience the process in which these expressions are being executed and the process identified by t will be referred to as the current process and the other process, respectively. Also, the right argument is sometimes called a *set message*, and it is said that a set message is sent to the other process.

Expressions are executed in another process for one of two purposes, either to set a value in the other process or to get a value from the other process. If the purpose is to set a value, as it is for Remote Set, then the message is sent asynchronously. That is, execution of `t 3: x` completes immediately, even though the message may still be on its way or is still being evaluated in the other process. In the absence of a message filter `.m.s` in the other process (see Interprocess Communication), when the message x is received it will be processed by the function

```
{:[4:x; .x; .[.;x]}
```

In effect, x can be a valid argument of Value/Execute, or a two-item list whose first (second) item is a valid left (right) argument to Index or Apply, or a three-item or four-item list that corresponds to a valid argument list to Amend. The only restriction is that a function must be a primitive function or a primitive derived function. Even though all such messages will be executed in the other process, the only effective ones are those that set a value, such as `"a:2+3"` or ``(`a;();:;5)``.

When t is ` then x is a character string holding an operating system command, and the effect of this function to have the command executed.

The result of sending a set message is always nil.

Error Reports

Domain Error if the left argument t is an invalid communication handle, or if valid, the partner identified by t is not available.

An error can occur in the other process.

Internal Data Type

```
4: x
```

Argument

The argument is any atom or list.

Definition

The result is the data type of the argument x, as an integer. Data types are as tabulated below.

Data Object	Type
Integer Atom	1
Floating-point Atom	2
Character atom	3
Symbol Atom	4
Integer Vector	-1
Floating-point Vector	-2
Character Vector	-3
Symbol Vector	-4
Other List	0
Dictionary	5
Nil	6
Function	7

Remote Get

`t 4: x`

Arguments

The argument t, which identifies another K process, is described in Communication Handle. The symbol ` is also allowed for t. The argument x can be any atom or list.

Description

Get a value from another process.

Definition

Assume for now that t is not `. For convenience the process in which these expressions are being executed and the process identified by t will be referred to as the current process and the other process, respectively. Also, the right argument is called a *get message*, and it is said that a get message is sent to the other process.

Expressions are executed in another process for one of two purposes, either to set a value in the other process or to get a value from the other process. If the purpose is to get a value, as it is for Remote Get, then the message is sent synchronously; execution of `t 4: x` does not complete until the message has been received, processed, and sent back by the other process, and finally received by the current process and made the result of `t 4: x`. In the absence of a message filter `.m.g` in the other process (see Interprocess Communication), when the message x is received it will be processed by the same function as set messages (see Remote Set), and therefore x can have any of the forms for set messages. However, the effective get messages are not the same as set messages, but instead those that get values from the other process, such as `` `a `` or `(a;!10)`.

When t is ` then x is a character string holding an operating system command, and the effect of this function to have the command executed. Since the operating system is sent a get message, the result of the command is returned as a list of character strings.

Error Reports

Domain Error if the left argument t is an invalid communication handle, or if valid, the partner identified by t is not available.

Domain Error if the other process cannot return a result for this right argument.

An error can occur in the other process.

Executable Form

`5: x`

Argument

The argument x is any atom or list.

Description

Give the character vector form of the default display of the argument.

With the exception of expressions that end in certain forms of Amend (see the chapter Amend, Index, Apply, and Assign), whenever an expression is typed in a K-session its value is displayed. That display can be captured by applying `5:` to the expression. The result is a character vector containing the display; new-line characters are used when the display requires more than one line. The Execute primitive can be applied to the result to reproduce the value. Consequently, this primitive is useful for capturing complicated constants in script files. If c is such a constant, simply write `5: c` to a text file with `0:` and copy that text file into the script file.

Synchronized File Append

```
f 5: c
```

Arguments and Result

The left argument f is a symbol atom, character atom or character vector, and the right argument c is a general list. The result is an integer atom.

Description

The left argument f holds the name of a file created by `f 1: b` for some K object b that is a general list. The effect of this function is to append the general list c to that file. The result is the count of the list in that file after the append takes place.

If the file named in f does not exist then this function is identical to `f 1: c`, except that the result is the count of c.

In all cases, execution of this function does not complete until the updated file is actually written to disk.

Interprocess Communication

A K server process is created using the `-i` command line option. For example, the following starts a server at port number 1234:

```
k -i 1234
```

(See the K User Manual for more information.) Production applications often need ways to monitor clients connecting to servers and the messages clients send. These facilities are provided by the contents of the root directory `.m`, which is present in every k session.

Authorization Vector

```
.m.u
```

The symbol vector `.m.u` contains the names of users that are permitted to connect to the process in which `.m.u` is defined.

Closed Connection Callback

```
.m.c
```

The character string `.m.c` is automatically executed when the connection to another process is broken.

Message Filters

```
.m.g
.m.s
```

Message filters provide the means to monitor messages received from other processes. In the discussion below a get message will always mean one sent to this process by way of `4:`, and a set message will always mean one sent by way of `3:`.

The message filter `.m.g` applies to get messages and is called the get message filter, while `.m.s` applies to set messages and is called the set message filter. The message filters replace the default message evaluation functions. Both filters are monadic functions which are automatically evaluated whenever a message of the appropriate type arrives. The argument to a message filter is always the entire message, i.e. the entire right argument of `3:` or `4:`, no matter what form that argument takes. The result of the get message filter is returned to the process that sent the message; the set message filter has no meaningful result.

Message filters have two general uses. First of all, they permit any form of message to be sent, whereas the default message evaluation functions fail in the general case. Secondly, while `.m.u` allows only authorized users to send messages to a process, not all authorized users are necessarily the same; often some can send messages of particular forms that others cannot. Message filters provide the means to monitor the messages of authorized users.

Synchronous Connection Filter

`.m.h`

This message filter applies to general synchronous interprocess communication provided by the `-h` listening port option (see the K User Manual). It is somewhat similar to the get message filter `.m.g`, but the communication partner (i.e., the client) need not be another K process. The argument to `.m.h` is the data received from the partner, as a character array. As long as `.m.h` returns nil, more data is read, in each instance becoming the next argument to the filter. When the return value is non-nil, it is sent back to the client, the current connection is closed, and the `-h` port returns to its previous listening state.

When `.m.h` is not defined, the default behavior is to echo all received data back to the client.

CHAPTER 12

COMMANDS

Commands are statements that display and set K system parameters, load source code, monitor and control execution, and send non-K commands to the host operating system for execution. Commands do not have explicit results, cannot be part of expressions, and cannot be directly executed within function expressions. However, commands can be executed indirectly by way of the Value/Execute primitive function.

Adverbs

```
\'
```

Online help for adverbs.

Assignment, Functions, Control

```
\:
```

Online help for assignment, functions, and control statements.

Attributes

```
\.
```

This command causes a summary of all primitive attributes and their meanings to be displayed in the session log.

Break Flag

`\b [`*`character`*`]`

The settings of this command distinguish whether all Stop / Trace commands mean stop (`\b s`), trace (`\b t`), or have no effect (`\b n`). If a character is not present, the current setting is displayed.

Commands

`\`

This command causes a summary of all commands to be displayed in the session log.

Console Flag

`\c` *`boolean`*

This flag controls activation of the user console. When `\c 0` is executed, the console is closed. If there are no current communication partners nor active K graphical objects associated with the session, then the parent K process is terminated. If the session remains active after the console is closed, and a signal or error condition arises which requires console input, then the console reappears (see Error Flag).

If `\c 1` is executed while the console is down, the console reappears. Otherwise, the command has no effect. The console may also be retrieved from any K graphical object associated with the session by typing `<ctrl-K>` from within the object.

Data and I/O Verbs

`\0`

Online help for data representation and I/O verbs.

Directory

`\d [`*`name`*`]`

The directory command is used to specify the current directory to be the one named in *`name`*, or display the name of the current directory if *`name`* is not present. The significance of the current directory is that all relative names that are assigned values will be relative to the current directory. This makes it convenient when writing utility programs; simply set the current directory in the utility script to the one in which the utilities will reside in the first line of the utility script, and then relative names can be used instead of fully qualified ones from that point on. Use the absolute name of the utility directory when setting the current directory so that the utility can be properly loaded from anywhere within an application script.

As a convenience, if `^` or `~` appears in place of *`name`*, then the parent or attribute directory of the current directory, respectively, becomes the new current directory.

The default current directory at the beginning of the K session is `.k`.

The setting of `\d` is overridden by the setting of the system variable `_d`. As a consequence, if a script is loaded while an expression is executing, any settings of `\d` in that script will not take effect.

Directory Entries

`\v [`*`directory`*`]`

A list of names of all global variables in *`directory`* is displayed, or the current directory if *`directory`* is not present, or the parent or attribute directory of the current directory if *`directory`* is `^` or `~`, respectively.

Error Flag

`\e [`*`boolean`*`]`

The debug flag controls the behavior of the interactive environment in response to an error in a primitive function or a signal in a defined function. The default behavior, which occurs when the setting is 0, is to report the error or signal in the interactive session and wait, unsuspended, for the next input. When the setting is 1, which is assumed for the descriptions in the chapter Controls and Debugging, the error is reported or the signal occurs, as before, but execution is suspended. In this case, the context of the interactive session is the same as that existing at the time of the error or signal.

If no boolean setting is present, the current setting of the error flag is displayed.

Exit

`\\`

End the K session.

Interrupt

`<ctrl-C>`

Interrupt execution of a running K application.

Invalid Values

`\i [`*`name`*`]`

A list of names of all global variables whose dependency definitions explicitly refer to the global variable named in *`name`* , i.e. all global variables that depend directly on the named global variable. If *`name`* is not present, a list of all currently invalid names is displayed, i.e. all names whose dependency expressions will be evaluated the next time they are referenced.

Load

`\l `*`file`*

The source code of applications and utilities is maintained in one or more files called scripts. In order to run an application, its scripts must be loaded after a K process is started. The effect of this command is to load the runtime program `file.kr`, or if that file does not exist, the script `file.k` .

Even though the current directory can be set in a script file and has the expected effect on subsequent definitions in that file, once the load is complete the current directory is automatically reset to the one when the load command was executed.

OS Command

`\`*`text`*

When the text following the back-slash is not the text of one of the other K commands, it is passed along to the operating system for evaluation.

Print Precision

`\p [`*`digits`*`]`

Print precision is the maximum number of decimal digits that can appear in the default format of a floating-point number. If *`digits`* is absent, the current value is printed out. Otherwise, the print precision is reset to the value of that constant. The default is 7, and the valid settings are the integers 0 through 19. If `digits` has any other value then the setting is left unchanged and the current value is displayed. A value of 0 indicates that all available digits should be used in the display.

Print Precision affects values typed by K in the session log and results of the primitive monadic Format function.

Random Seed

`\r [seed]`

The random seed is the seed of the random number generator used, for example, in the library functions Deal, Random Selection and Random Probability (see Draw in the chapter System Functions). Its purpose is to permit multiple experiments to be run on the same random sequence, by re-initializing the random sequence at the beginning of each experiment, and optionally resetting the random seed. In the absence of the constant signified by `seed`, the random sequence is re-initialized with the current value of the random seed, which is printed out. Otherwise, the random sequence is re-initialized with random seed reset to the value of that constant, which must be an integer.

Runtime Program

`\kr file`

The script `file.k` in the current OS directory is converted into a runtime program `file.kr`. Such programs are compiled and encoded K code, and may be loaded into both the developer and runtime versions of K (see Load). Note that unencoded scripts of the form `file.k` may not be loaded into runtime K.

Set Timer

`\t [digits]`

Set the timer to the number of seconds given by the integer represented by `digits`, or display the current setting if `digits` is not present. If the setting is a positive integer n then the global variable `.t` is assigned the value of `_t` every n seconds; a trigger on `.t` will then be executed every n seconds. No assignments of `.t` occur if the setting is 0, the default.

Step

`\s` *`file`*

The script `file.k` is step-wise loaded line-by-line, using the Return key as the load proceeds. The load may be aborted at any time by entering the back-slash character `\` (see Abort in the chapter Controls and Debugging).

System Names

`\_`

This command causes a summary of the meanings of all system functions and system variables to be displayed in the session log.

Time

`\t` *`expression`*

The execution time of the expression, in milliseconds, will be printed out when execution completes.

```
  \t +/!10000
16
  \t do[100;+/!10000]
750
```

Verbs

`\+`

Online help for verbs.

Workspace Size

`\w`

Displays the space used, space allocated, and space allocated to mapped files.

CHAPTER 13

SYSTEM VARIABLES

System variables are special global variables whose meanings and values are determined by the K process. System variables are always available, and their names, which all begin with underscore (_), never need to be qualified with path information.

Three of the system variables are related to amending global variables. Whenever a global variable is being amended the system variables `_d` , `_v` and `_i` are automatically set to the directory of which the global variable is an entry, the entry itself, and the indices of the items-at-depth being changed. These three values are available within the function argument of Amend and Amend Item and the Update and Validation Attributes.

Current Directory

`_d`

The value of `_d` , like the result of the command `\d` , is the current directory, i.e. the directory in which relative names are resolved. The value is a symbol holding the name of the directory, i.e. a handle. This value is automatically set to the directory whose entry is being modified by the current Amend or Amend Item.

Current Global Set

`_v`

The value of `_v` is a symbol holding the absolute name of the global variable currently being modified by Amend or Amend Item.

Current Time

`_t`

The value is the current time, measured in seconds from some initial point, as a nonnegative integer atom. The base time, i.e. _t is 0, is 12:00 AM, January 1, 2035. See GMT Time Stamp, Local Time Stamp, Dates from Julian Days and Julian Days from Dates in the chapter System Functions.

Host Process (Machine Name)

`_h`

The name of the machine (as a symbol atom) on which the current K process is running.

Host Process (Port)

`_p`

The port number of the current K process, or 0 if no port was identified when the process started.

Items Changed

`_i`

The value of `_i` is the atom or list of indices where the global variable named in `_v` is currently being modified.

Message Source (Handle)

`_w`

The value of `_w` is the communication handle of the current message sent to this process from another K process, or 0 if the source of the current message is the console or screen.

Message Source (User)

`_u`

The value of `_u` is the name (as a symbol atom) of the user of the K process that sent the current set message to this process, or `` ` `` if the source of the current message is the console or screen.

Nil Value

`_n`

The value of nil, which is the value of any unspecified list item. For example, the first and third items of `(; 1 2 3; )` have values equal to nil.

Self Referent

`_f`

A recursive function is one whose definition refers to itself. A simple example is the factorial function `fac[n]`, whose value for a positive integer n is the product of the first n positive integers. Since this product equals `n*fac[n-1]`, and since `fac[1]` equals 1, `fac` can be defined by:

```
fac: {:[x>1; x * fac[x-1]; 1]}
```

Functions are data and in particular, do not have to be named. A question that then arises is: how is an unnamed function referred to in a recursive function expression? The answer is: with the self referent system variable `_f`. The factorial function can be redefined using `_f` by:

```
{:[x>1; x * _f[x-1]; 1]}
```

Of course factorial can also be defined by `fac:{*/ 1 + !x}`.

CHAPTER 14

SYSTEM FUNCTIONS

System functions have names that begin with underscore (_). All system functions are syntactically verbs. For example, applications of `_in` and `_bin` can be expressed in infix notation, e.g. `a _in b` and `a _bin b` .

Binary Search

```
x _bin y
x _binl y
```

Arguments

In the case of `_bin` , the left argument `x` is any list without duplicate items and in ascending order, i.e. x is identical to `x @ < x` , and the right argument y is any atom or list. In the case of `_binl` , both `x` and y can either be an integer vector or floating-point vector, and otherwise x satisfies the other conditions of the left argument of `_bin` .

Description

Find y in x using binary search.

Definition

The function `x _bin y` is defined in the much the same way as same way as `x ? y` , except that the restrictions on the left argument permit y to located in the intervals `(x[i-1], x[i])` , and not just identical to an item of x. Consequently, the result is the largest index i for which the list `(x[i-1]; y; x[i])` is in sort order, i.e. if `<(x[i-1]; y; x[i])` is `0 1 2` . Otherwise, the result is 0 if `(y; *x)` is in sort order, and `#x` if `(*|x; y)` is in sort order.

The restrictions on the left argument are required by the binary search algorithm used in this function, which is more efficient than the general search algorithm used in Find. For the sake of efficiency the restrictions are assumed to apply; the function does not verify them. Consequently, the function will not fail if the restrictions do not apply, but the result will be meaningless.

`x _binl y` is equivalent to `x _bin/: y` wherever it is defined.

Error Reports

Rank Error if `x` is an atom.

Delete Indices

```
x _di y
```

Arguments

Either the left argument x is a list and the right argument y is integer, or the left argument is a dictionary and the right argument is a symbol atom or vector.

Description

Delete items `x[y]` from x.

Definition

The result of `x _di y` for a list x is x with items `x[y]` deleted. For example:

```
  "abcdefghi" _di 3 1 5
"aceghi"
```

In the case of a dictionary x, the result is x with the entry or entries named in y deleted. For example:

```
  \d
.k
  a:2; b:3; c:"abc"; d:7 8
  .k _di `a `c
.((`b;3;)
  (`d
   7 8
   ))
```

If y is a vector of entry names (e.g. `y:!x`), use `x _di y` to delete them all.

Error Reports

Index Error if an atom in the right argument is not a valid index of the left argument.

Rank Error if x is not a list or dictionary.

Delete Value / Delete Value List

```
x  _dv y
x  _dvl y
```

Arguments

In the case of `_dv`, the right argument y is any list or atom, while for `_dvl` the right argument must be a list. In either case the left argument x is a list and the result is a list whose count does not exceed `#x`.

Definition

The result of `x _dv y` is x with any items that match y deleted. Since Match is used, floating-point comparisons are subject to comparison tolerance.

For example:

```
  3 5 1 7 _dv 5                    3 5 1 7 _dv 1 5
3 1 7                            3 5 1 7
```

In the example on the right, the list `1 5` is not an item of the list `3 5 1 7`, so nothing is deleted. The function `x _dvl y` is like `x _dv y`, except that every item of y, not y itself, is deleted from x. That is, `x _dvl y` is equivalent to `x _dv/ y`. For example:

```
  3 5 1 7 _dvl 1 5
3 7
```

Error Reports

Rank Error if x is not a list.

Draw

```
x _draw y
```

Description

Make x random selections from `!y` with replacement if y is positive, from `!-y` without replacement if y is negative, or from the interval `(0,1)` if y is 0.

Arguments

The left argument x is a nonnegative integer atom or vector and the right argument y is an integer atom. If y is negative then `*/x` is less than or equal to `-y`.

Definition

Assume for now that x is an integer. If y is positive the result is an integer list of count x whose items are integers randomly selected from `!y` . This function is sometimes called Random Selection. For example:

```
  5 _draw 3                       5 _draw 7
1 2 2 0 2                       4 5 4 6 4
```

If y is negative the result is an integer list of count x whose items are distinct integers randomly selected from `!-y` . This function is sometimes called Deal, and since the items of the result are distinct, x must be less than or equal to `-y` . For example:

```
  4 _draw -9                      9 _draw -9
3 5 0 4                         5 4 7 0 2 3 6 8 1
```

If y is 0 the result is a floating-point vector of count x whose items are distinct numbers greater than or equal to 0 and less than 1, chosen from a uniform or rectangular distribution. This function is sometimes called Random Probability.

```
  6 _draw 0
0.3655 0.2888 0.2184 0.8171 0.693 0.6323
```

If x is a vector then `x _draw y` is identical to `x # (*/x) _draw y` in all cases.

Error Reports

Length Error if y is negative and `*/x` is greater than `-y` .

GMT Time / Local Time

```
_gtime x
_ltime x
```

Argument

The argument is an integer atom and the result is a two-item integer vector.

Description

Time from seconds.

Definition

Give the time produced by the current time system variable `_t` as a two-item integer vector, where the first item is yyyymmdd and the second is hhmmss. `_gtime` gives GMT and `_ltime` gives local time. For example:

```
  _gtime _t                    _ltime _t
19951001 50000               19951001 0
  _gtime 0
20350101 0
```

The argument does not have to be `_t`, but must represent time duration in seconds starting at the same point in time as `_t`.

Error Reports

Type Error if the argument x is not an integer atom.

Integer from Character / Character from Integer

`_ic x` and `_ci x`

Argument

In the case of `_ic`, the argument is character, while for `_ci` the argument is integer. The result is identical in structure to the argument, but is integer when the argument is character and character when the argument is integer.

Definition

Both `_ic` and `_ci` are atom functions. In the case of `_ic`, the result is just like the argument except that every character atom in the argument is replaced by its ASCII integer value in the result. In the case of `_ci`, every integer atom in the argument becomes the character in the result whose ASCII value is that integer modulo 256 (i.e., `x ! 256`).

Error Reports

Type Error if the argument of `_ic` is not character or the argument of `_ci` is not integer.

Julian Day from Date / Date from Julian Day

```
_jd x
_dj x
```

Argument

The argument and result are integer atoms.

Definition

`_jd` gives a Julian day count for an integer argument of the form yyyymmdd, such as the first item of the result of either `_gtime` or `_ltime`, and `_dj` produces a date from a Julian day count.

These functions are compatible with `_t`, i.e.:

```
  _dj 0
20350101
```

In particular, since this date is a Monday, the day of the week for any time `_t` is:

```
  `Mon`Tues`Wed`Thur`Fri`Sat`Sun @ (_ _t%86400)!7
```

Error Reports

Type Error if the argument is not an integer atom.

Least Squares

```
x _lsq y
```

Description

The least squares solution w of the linear equations (in conventional mathematical notation) *yw* = *x* .

Arguments

The argument x is a floating-point vector or matrix and y is a floating-point matrix. Since y is a floating-point matrix all its items are floating-point vectors of the same count. That count must equal the count of a vector x or the count of every item of a matrix x, and cannot be less than the count of y. Also, the matrix y must be non-singular, i.e. the items must be linearly independent.

Definition

The items of y are considered to be the columns of the matrix. Consequently, for any vector w of count equal to the count of y, the matrix-vector product yw is `+/ y*w` . If w is a solution of the above equation then `+/y*w` equals x, or equivalently, the items of the vector `x - +/y*w` are all 0.0 (in practice, they are approximately 0.0). If w is not a solution, then

```
  +/(x - +/y*w)*2
```

is called the sum-of-squares measure of how close w is to a solution.

The equation may not have a solution when each column of y has more items than y itself. However, if the matrix is non-singular, then there is a unique w for which the sum of squares has the smallest possible value. That w is `x _lsq y` .

Note that if the equation has a solution then it is `x _lsq y` . For example:

```
  y: (1 1 1.0;1 2 4.0)
  x: 1 2 3.0
  w: x _lsq y
  w
0.5 0.6428571
  +/y*w
1.142857 1.785714 3.071428
```

The vector w is not a solution, but is the least-squares solution.

`x _lsq y` is identical to `x _lsq\: y` for a matrix `x`.

Error Reports

Domain Error if the matrix y is singular.

Type Error if either argument is not floating-point.

Math Functions

`_abs x` to `_tanh x`

Argument

The argument of every math function is numeric.

Definitions

Every math function is an atomic function. They are:

`_abs x`	absolute value function, or magnitude function
`_arccos x`	inverse cosine function
`_arcsin x`	inverse sine function
`_arctan x`	inverse tangent function
`_cos x`	trigonometric cosine function
`_cosh x`	hyperbolic cosine function
`_exp x`	exponential function e^x
`_floor x`	the integer part of x as a floating-point whole number.

(This function does not use Comparison Tolerance. Compare with `_ x` .)

`_log x`	natural logarithm function
`_sin x`	trigonometric sine function
`_sinh x`	hyperbolic sine function
`_sqr x`	x-squared, i.e. `x^2.0`
`_sqrt x`	square root function
`_tan x`	trigonometric tangent function
`_tanh x`	hyperbolic tangent function

The result of any math function is a floating-point atom or list.

Error Reports

Type Error if an atom argument x is not numeric.

Domain Error if an atom argument x is not in the domain of the function.

Matrix Functions

```
x _dot y
x _mul y
_inv x
```

Arguments

For the first function, x and y are conformable numeric lists. For the next two, they are numeric matrices. (Some other numeric structures are legal, but of limited practical use.)

Definition

The first function is dot product, the second is matrix multiply, and the third is matrix inverse. Dot product is simply the function `+/*`. Matrix multiply is the function `{x _dot\: y}`. Matrix inverse uses the system function `_lsq` to find the inverse of a square matrix. For example:

```
  2 3 4 _dot 9 8 -2
34
  A: (2 3 4; 1 2 3; 0 2 1)
  B: (1 2; 5 -2; 3 4)
  A _mul B
(29 14
 20 10
 13 0)
  _inv A
(1.333333 -1.666667 -0.3333333
 0.3333333 -0.6666667 0.6666667
 -0.6666667 1.333333 -0.3333333)
```

Inverting a singular matrix results in a matrix filled with `0n` values.

Error Reports

Type Error if arguments are non-numeric, or if the argument for `_inv` is not a matrix.

Length Error if the arguments for `_dot` do not conform, or if the arguments for `_mul` to not conform along the inner dimension. Also, length error if the argument for `_inv` is not a square matrix.

Membership / List Membership

```
x _in y
x _lin y
```

Arguments

In the case of `_in`, the left argument x is any list or atom and the result is an integer atom, while for `_lin` the left argument must be a list and the result is an integer list of the same count. In either case the right argument y is a list.

Description

Is x in y?

Definition

The result of `_in` is 1 is if `x` matches any item of y, and 0 otherwise. Since Match is used, floating-point comparisons are subject to comparison tolerance.

For example:

```
  5 _in 3 5 1 7
1
  9 _in 3 5 1 7
0
```

This function is similar to Find. Namely, `x _in y` is `(y ? x) < # y`.

The result of `_lin` is an integer list whose ith item is 1 if `x[i]` matches any item of y, and 0 otherwise. That is, `_lin` applies `_in` to every item on the left and is equivalent to `x _in\: y`. For example:

```
  5 9 _lin 3 5 1 7
1 0
```

Error Reports

Domain Error if y is not a list.

Scalar from Vector

```
x _sv y
```

Arguments and Result

The left argument x is either a positive integer atom or vector, while the right argument y is integer and if a list, its items must be conformable in the manner described below. If both arguments are lists they must have the same count. The result is integer.

Description

Evaluate y in the radices x.

Definition

Scalar from Vector computes the base value of a vector y in a number system with radices x. An atom x paired with a list y is treated like `(#y)#x` .

```
  10 _sv 1 9 9 5
1995
  2 _sv 1 0 0 1
9
  24 60 60 _sv 1 3 25
3805
```

If the right argument is a matrix then the result of `x _sv y` is a vector with `# * y` items whose ith item `(x _sv y)[i]` is identical to `x _sv y[;i]` . The general case can be seen from the following definition of this function, which is based on the polynomial evaluation method called Horner's method, and which for an integer vector left argument is:

```
  sv: {{z + y * x}/[0; x; y]}
```

The function `{z + y * x}` is atomic, and its first application in an evaluation of `_sv` is:

```
  t1: {z + y * x}[0; x[0]; y[0]]
```

Since 0 and `x[0]` are atoms the result `t1` is identical to `y[0]` in structure. The next application is:

```
  t2: {(z + y * x}[t1; x[1]; y[1]]
```

Since `x[1]` is an atom, `t1` and `y[1]` must be conformable, and therefore `y[0]` and `y[1]` must be conformable. And so on. The items of y must be conformable so that, for example, any Over for dyadic f applied to y, such as `+/y` , is defined.

Error Reports

Domain Error if the left argument x is integer but not positive.

Length error if the arguments are both lists but their counts are different.

Type Error if either argument is not integer.

String Match

`x _sm y`

Arguments and Result

The left argument x is a symbol, string, or list of symbols and/or strings. The right argument y is similar. The result is a boolean list.

Description

Indicate whether a string in x matches one in y.

Definition

String Match is a string-atomic function that yields a 1 wherever a string in x matches one in y, and 0 otherwise. Special wild-card characters may appear in strings of y:

- `*` one or more: `"b*t"` matches `"bet"` and `"beat"` and `"beast"`
- `?` one: `"b?t"` matches `"bat"` and `"bet"` and `"bit"`
- `[…]` one of: `"a[cr]t"` matches `"act"` and `"art"`
- `^` none of: `"ab[^bc]"` matches `"abd"` but not `"abb"` nor `"abc"`
- `-` range: `"[0-2]4"` matches `"04"` and `"14"` and `"24"`

The literal value of a wild-card character is specified by enclosing it in brackets, e.g. `[^]`. Some examples:

```
  files: ("a.c";"foo.h";"bc")
  files _sm "*.[ch]"
1 1 0
  `one `two `three `four _sm "two"
0 1 0 0
  (`one;"one") _sm (("on[^a-z]"; `b); ,"one")
(0 0
 ,1)
```

Error Reports

Type Error if x or y is not as described.

Length error if the arguments are both lists but their counts (down to string-atomicity) are different.

String Search

```
x _ss y
```

Arguments and Result

The left argument is a character string or string list, and the right argument is a string, symbol, or list of strings and/or symbols. The result is a list of nonnegative integers.

Description

Find all occurrences of strings of y in strings of x.

Definition

Like String Match, String Search is a string-atomic function (see String-Atomic Function). If x and y are character strings, then each item in the resulting integer vector is the first index of a unique occurrence of the string y in the string x. That is, if `r: y _ss x` then for every index i of r:

```
  y ~ x[i+!#y]
```

Matching substrings may not overlap. For example:

```
  x:"Mississippi"
  x _ss "issi"
,1
  x _ss "iss"
1 4
  x[1+0 1 2 3]
"issi"
```

If y is not a substring of x, the result is the empty integer vector `!0`.

Wild-card characters defined for String Match are permissible in y, with the exception of `*`. When the right argument y is a symbol, it is treated as a word; that is, a sequence of characters surrounded by non-alphanumerics. For example:

```
  x:"Extract 15 words out of 1015."
  x _ss `"15"
,8
```

Error Reports

Type Error if the argument types are not as described above.

Length Error if the arguments do not conform (see String-Atomic Function), or if y (or an item of y) is the empty string `""`.

String Search and Replace

```
_ssr[x;y;z]
```

Arguments and Result

The first argument x is a character string, and the second argument y is a string, symbol, or character. The third argument z may be an atom, string, or monad. The result is a string.

Description

Find all occurrences of y in string x, and replace with z.

Definition

Each occurrence of the substring y in the character string x is replaced with the value of z. For example:

```
  s:"Adam had a pear"
  _ssr[s;"a";"the"]
"Adthem hthed the pether"   / replace substrings
  _ssr[s;`a;"the"]
"Adam had the pear"         / replace word
```

The result is identical to x if y is not a substring of x.

All wild-card characters permitted for String Search are also valid here.

Error Reports

Type Error if either argument is not a character string.

Vector from Scalar

```
x _vs y
```

Arguments

The left argument x is either a positive integer atom or vector, while the right argument y is an integer atom or list of integers. The result is a list of integers.

Description

Expand y in the radices x.

Definition

Vector from Scalar computes the base representation of y in radices x. For example:

```
  10 _vs 1995
1 9 9 5
  2 _vs 9
1 0 0 1
  24 60 60 _vs 3805
1 3 25
```

If the right argument is an integer vector then the result of `x _vs y` is a matrix with `#x` items whose ith column `(x _vs y)[;i]` is identical to `x _vs y[i]`. More generally, the right argument y can be any list of integers, and each item of the result is identical to y in structure. For example:

```
  a: 10 _vs 1995 1996 1997
  a
(1 1 1
 9 9 9
 9 9 9
 5 6 7)

  a[;0]
1 9 9 5
```

```
  10 _vs (1995; 1996 1997)
((1
  1 1)
 (9
  9 9)
 (9
  9 9)
 (5
  6 7))
```

```
vs:{|(-1 _ i)-c*1 _ i:y (_%)\ c:|x}
```

Error Reports

Domain Error if the left argument is integer but not positive.

Type Error if either argument is not integer.

CHAPTER 15

SCREEN DISPLAYS

Every global variable can be displayed on the screen simply by saying “show it”, as in:

```
`show $ `x
```

for the global variable x. A variable and its screen display are tightly coupled; when the displayed value is edited the value of the variable changes automatically, and when the variable is amended in the workspace the screen display changes accordingly. Consequently, when the screen is edited, a global variable automatically changes value, its trigger – if it has one – fires immediately, and any other global variables dependent on it are marked invalid (and automatically re-evaluated if displayed on the screen, thereby keeping the screen view consistent). In this way the basic interactions with users are handled simply and automatically.

The display of a global variable is removed from the screen by saying “hide it”, as in:

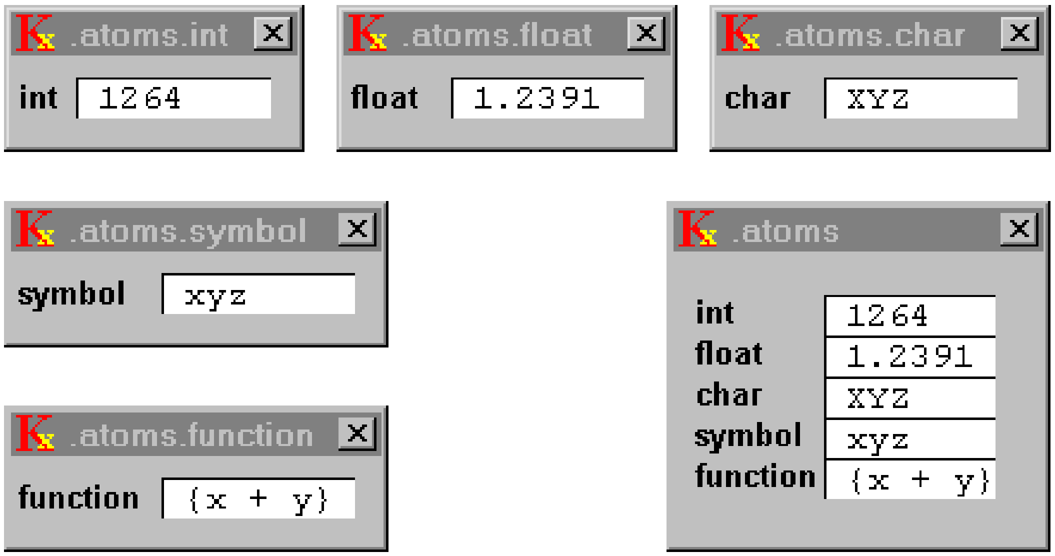
```
`hide $ `x
```

Usually, both `` `show$ `` and `` `hide$ `` are applied to handles, in which case their result is nil. If either monad is applied to an expression, a variable is created for it and its handle returned. These are named `` `s0 `s1 `s2 `` … , and should be considered reserved names since they are used in the given order regardless of any previous assigned value.

Data Presentation

Every atomic K data type has a default display. Examples are shown in the following figure. They were created as follows:

```
  \d .atoms
  int: 1264
  float: 1.2391
  char: "XYZ"
  symbol: `xyz
  function: {x + y}
  `show$' `int`float`char`symbol`function`.atoms
(;;;;;)
```



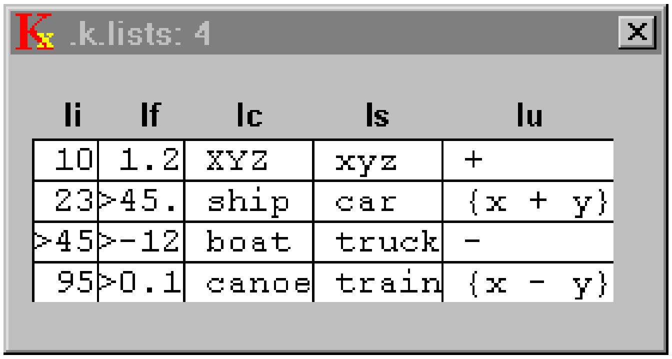
```
int 1264
float 1.2391
char XYZ
symbol xyz
function {x + y}

int      1264
float    1.2391
char     XYZ
symbol   xyz
function {x + y}
```

Note that character strings are atomic for screen displays, rather than just character atoms. There are also default displays for integer, floating-point and symbol vectors, lists of character vectors and functions, and dictionaries whose entries have these types of values. In all cases the displays are column-oriented, i.e. items display as columns. For example, for dictionaries:

```
  \d .k.lists
  li: 10 23 45231 95
  lf: 1.2 45.768 -12.34 0.123
```

```
lc: ("XYZ"; "ship"; "boat"; "canoe")
ls: `xyz `car `truck `train
lu: (+; {x + y}; -; {x - y})
\d ^
`show $ `lists
```

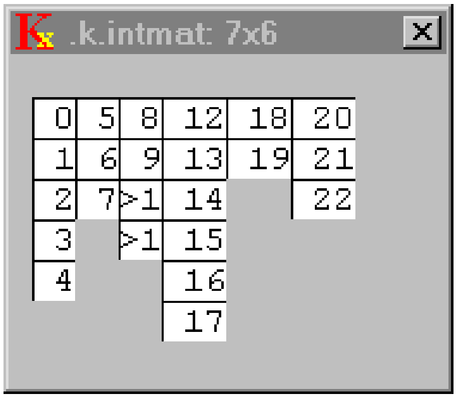


li	lf	lc	ls	lu
10	1.2	XYZ	xyz	+
23	>45.	ship	car	{x + y}
>45	>-12	boat	truck	-
95	>0.1	canoe	train	{x - y}

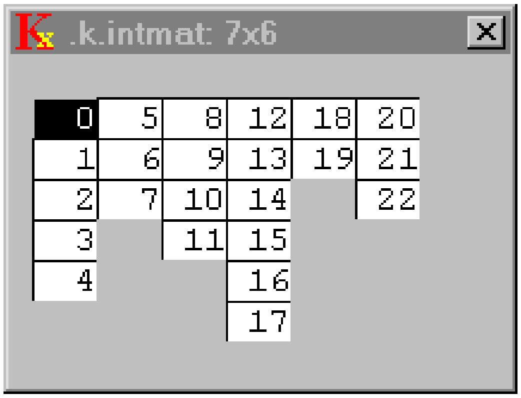
The independent displays of the entries in this dictionary are also column-oriented and look much like the above display, except that they have no column titles.

Finally, there are default displays for matrices of the basic types. Executing the following results in the display shown below on the left:

```
intmat: 0 5 8 12 18 20 23 _ ! 23
`show $ `intmat
```



0	5	8	12	18	20
1	6	9	13	19	21
2	7	>1	14		22
3		>1	15		
4			16		
			17		



0	5	8	12	18	20
1	6	9	13	19	21
2	7	10	14		22
3		11	15		
4			16		
			17		

Note that the items of intmat are the columns of the display. The columns show the application of a default formatting function, as can be seen from the third column where the entries with more than one digit (10 and 11) are not shown. The default formatting can be overridden by setting the Format attribute. For example, evaluating the following line will cause the display to change to the one shown above on the right:

```
intmat..f: 2 $
```

Display Classes

The display class of a variable can be specified with the Class attribute. The default display class is `` `data ``, and all the displays discussed in the previous section are of that class. There are also `` `chart `` and `` `plot ``, `` `check `` and `` `radio ``, and `` `button ``. Finally, the entries of dictionaries of class `` `form `` can be arranged in a variety of ways on the screen. (See Arrangement in the chapter Attributes.)

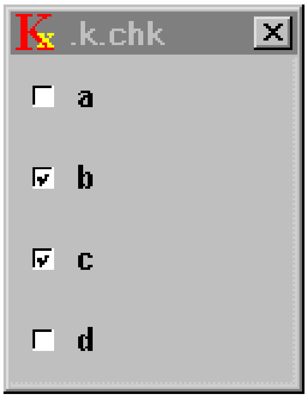
Both the chart and plot classes are for graphs. A numeric vector y can be displayed as a chart, and is plotted vertically against the horizontal coordinates `!#y`. A numeric matrix can also be displayed as a chart, in which case each (vector) item `y[i]` is plotted against `!#y[i]`.

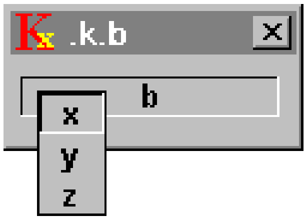
The plot class is like chart, except that coordinate pairs must be supplied. The simplest case is a two-item numeric matrix `xy` whose items have the same count, in which case `xy[1]` is plotted vertically against horizontal `xy[0]`. A list of such two-item lists will produce a set of independent traces on the same graph.

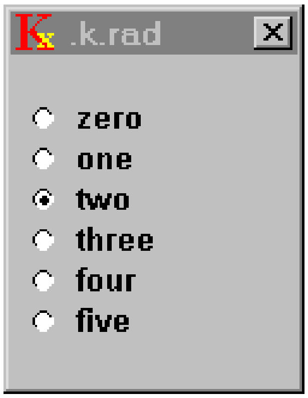
An integer atom whose value is 0 or 1 can be displayed as a check button, and a dictionary whose entries are all atoms with values 0 or 1, and whose display classes are all `` `check ``, will display a series of check buttons. For example:

```
chk:.+(`a`b`c`d;0 1 1 0)
chk[.;`c]:`check
`show $ `chk
```

Since the check buttons are held in a dictionary they can be arranged in many different ways simply by setting the arrangement attribute on the dictionary. Note that the class of the dictionary is automatically a form because the class of its entries was explicitly set (to `` `check ``).







The radio class uses the option list attribute in its display. If a variable is of class `` `radio ``, then its `.o` attribute must be a symbol vector, and its initial value must be one of these symbols. For example, the following code results in the display shown above on the right:

```
rad..o: `zero`one`two`three`four`five
rad: rad..o[2]
rad..c: `radio
`show $ `rad
```

See also Option List in the chapter Attributes for a description of the radio class.

Finally, any character vector can be displayed as a button. The contents of the character vector must be a valid K expression or sequence of expressions, which will be executed whenever the button is pressed.

A dictionary whose entries are valid buttons can be made into a pulldown menu. For example, the following code will generate a pulldown menu with three items:

```
b.x: "2+3"
b.y: "5-2"
b.z: "8*!3"
b..c: `button
`show $ `b
```

Now put the mouse cursor on the button b and press and hold the left mouse button. A display of the buttons x, y and z will appear, as shown in the center of the above figure. While still holding the left mouse button, move the mouse cursor to the button x and release the button. The expression `2+3` will be executed and 5 will appear in your session log. Note that `` `button `` is the class of the dictionary, but not necessarily the class of the entries.

INDEX