



USER 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.

K User Manual

Copyright © 1998 by Kx Systems, Inc.

Edition 1, revision 4. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright owner.

This book is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Kx Systems, Inc. Kx Systems assumes no responsibility or liability for any errors or inaccuracies that may appear in this book. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of this license.

TABLE OF CONTENTS

TABLE OF CONTENTS 3

1: PROGRAMMING ENVIRONMENT 5

- Executing the Examples in the Manual 5
- Atoms and Lists 6
- Errors 7
- Incomplete Expressions 8

2: A TOUR OF K 11

- Introduction 11
- Familiar Symbols with Familiar Meanings 12
- Data Types and Terminology 13
- The Familiar Symbols Applied to Lists 17
- Familiar Functions on Somewhat Familiar Symbols 19
- Join and Enlist 20
- Assignment and Indexing 22
- Where 25
- Order of Evaluation, and the Way to Read Expressions 26

3: PREPARING THE SAMPLE APPLICATION 29

- Data Organization and Display 30
- The K Name Space 34
- Cross-Sectional Indexing 38
- Symbolic Indexing 40
- Attributes 44
- The Sample Application Revisited 47

4: MORE PRIMITIVE FUNCTIONS 53

- Atom, Count and Shape 53
- Take 56
- Enumerate 57
- Match 57
- Find 58
- Grade Up and Grade Down 59
- Formatting and Unformatting Data 62

5: OPERATORS AND DEFINED FUNCTIONS 65

- Each-Right 65
- Each-Left 66
- Each and Monadic Case 66
- Definition of Non-Primitive Functions 69
- Over and Scan 72
- Greatest and Least 75
- Some and All 75

Appendix A: K ON UNIX 77

Appendix B: K ON WINDOWS 79

- Running K 79
- Environment and Nonstandard Commands 80
- Error Behavior 81

INDEX 83

PROGRAMMING ENVIRONMENT

The programming environment is an interactive session manager. When the session manager starts up, an introductory message is displayed and the cursor is placed on a new line, indented two spaces. The user can type a statement, and after pressing Return the input is evaluated, its result (if any) is displayed, and the cursor is placed on the next line, indented two spaces and waiting for the next user entry. By a *statement* we mean an expression in the K language or a command; the latter resembles an operating system command and affects the working environment. A K program is a function definition, which is just an expression of a special form.

To exit from the session, simply type `\\` on the input line and press Return.

Script files that define toolkits and applications are simply text files containing statements. When the command to load a script is executed the effect is to evaluate every line in the script from top to bottom as if they were entered by hand in that order. The script can be edited when changes are needed and then loaded again. Any statement that can be entered by hand in a session can also appear in a script, including statements to load other scripts.

Executing the Examples in the Manual

Examples in this manual are displayed as if the evaluations were carried out in an interactive K session. That is, expressions whose results are to be displayed are indented two spaces, and their results are displayed just below. For example:

2 + 3	the user enters this
5	K displays this

Annotations like “the user enters this” and “K displays this” are often added to examples in the text as part of the discussion, and are not part of the simulated interaction with K. They are always set off to the right and are typeset in the font for text rather than the one for expressions, so it should always be easy to pick them out. If you have access to a session and are evaluating the expressions in the manual, do not include the annotations.

All displays of K statements and output in this manual are in a monospaced font, as if they were typed on a computer keyboard. The monospaced font is also used within the text for expressions, file names, etc. Sometimes an expression is displayed on a separate line simply for emphasis and is not meant to be executed. The context should make these cases clear.

A log accumulates vertically; an expression is entered, its result is displayed below it, the prompt for the next input appears below that result, and so on. In this manual, however, significant space is sometimes saved by putting two or three input expressions on one line and their results immediately below, as in:

$2 + 3$	$4 - 1$	$6 * 5$
5	3	30

This display cannot be reproduced in the interactive session. If you want to run the examples yourself you must enter them one at a time on different lines, or use the list notation discussed in the following section.

Atoms and Lists

Individual numbers and characters are called atoms in K. There are also lists of atoms, lists of lists of atoms, and so on. All lists are ultimately composed of atoms, but can have many levels between the top and the atoms at the bottom. Lists can be entered as a series of expressions separated by semicolons and surrounded by parentheses, as in $(2 + 3; 4 - 1; 6 * 5)$. This list is said to have three items, the first being $2 + 3$, the second $4 - 1$, and the third $6 * 5$. It is displayed as follows:

$(2 + 3; 4 - 1; 6 * 5)$	the user enters this
5 3 30	K displays the result

This list is called an integer list because its items are all integer atoms. Note the display of the result in the last example, with the items simply separated by spaces. Constant numeric lists can be entered in this way as well, one atom after the other, separated by spaces. For example:

```
10 34 -5 67          a constant expression is entered
10 34 -5 67          the result (its value) is displayed
```

Constant character lists are entered and displayed with the characters between double-quotes, as in:

```
"abcdeghqwe"
"abcdeghqwe"
```

Constant integer and character lists always display like they are entered, although extra spaces between the integers on entry do not appear in the displays.

Errors

If execution of an expression causes an error then an error message is printed and the input cursor appears on the next line, indented two spaces, the same as if the error had not occurred. For example:

```
"a" + "c"
term: type
           the cursor is now on this line
```

The `term:` part of the error message indicates that keyboard input is the source of the error. The `type` part indicates a data type error, in this case that Plus does not apply to characters.

This is the behavior when `\e 0` has been entered, setting the session Error Flag to zero, which is convenient for new users and casual use. The alternate behavior, for debugging, is for execution to suspend after the error message is printed so that the state of the computation at the time of the error can be examined. Execute the following command:

```
\e 1
```

Now when an error occurs not only is an error message printed, but the expression is displayed with a caret under it to indicate the function in the expression where the error occurred, and execution is suspended. The prompt now becomes > followed by 2 spaces. If another error occurs before this one is cleared the prompt will become >> followed by two spaces, and so on. To clear the last suspension enter \ following the prompt and Return. To clear all suspensions, repeat this as long as there are > characters in the prompt. You cannot clear more than one suspension at a time. For example:

```

    "a" + "c"
type error          argument types for + must be numeric
"a" + "c"
  ^
> "abc"[3]         note the > in the prompt
index error        the valid indices of "abc" are 0, 1, and 2
"abc"[3]
  ^
>> \              the prompt >> indicates two suspensions; enter \
> \                one suspension is cleared; enter \ again
                   the prompt is two spaces; the suspensions are cleared

```

The purpose of suspended execution is to determine the source of an error. Values of variables can be examined and execution can be resumed with a specified value for the function that caused the error. See the chapter Controls and Debugging in the K Reference Manual for more information.

Incomplete Expressions

No matter if the Error Flag is 0 or 1, a prompt with at least one > appears after entry of an incomplete expression. This is not the same situation as suspended execution when the Error Flag is 1. In that case new expressions can be entered while suspended, which is the common thing to do when you are trying to determine the cause of the error. However, if after pressing Return the expression is incomplete, everything you enter from now on becomes part of the expression until it is either completed or aborted. For example, the expression (1 2 3; "abcd") can be entered on two lines by breaking the entry at the semicolon, as follows:


```

(1 2 3
> "abcd")
(1 2 3
"abcd")

```

leave off the ; and press Return
the prompt is > ; you enter the second part
the result is displayed

the prompt is now the normal “space over 2”

As long as the closing right parentheses has not been entered, every new line will be an input line and no results will be printed. All input will become part of the expression that started with (1 2 3. However, it is always possible to abort the expression using \, the same command used in the previous section to clear an error suspension.

The behavior is similar when there is a missing right bracket or right brace, or a missing closing double-quote. Here is an example of the latter case:

```

"abcd
> "
"abcd\n"

```

prompt is > ; enter " to complete the constant
the result is displayed
the prompt is now the normal “space over 2”

The two-character sequence \n at the end of the above constant denotes the new-line character caused by pressing Return before the closing " was entered. If we had continued to enter expressions instead of the closing " , the expressions would simply have been appended to the character constant until finally a closing double-quote or the abort command is entered.

There can be more than one source of incompleteness, and then there are more than one > characters in the prompt. Repeating the first example:

```

(1 2 3
> "ab
>> cd")
(1 2 3
"ab\n cd")

```

the prompt is > ; continue entry
now the prompt is >> ; continue entry

the result is displayed; note the \n
the prompt is now the normal “space over 2”

If on the third line the entry had been cd" without the) , then the next line would have been an input line with one > in the prompt. Input mode would have continued until the expression that started with (was complete.

Sometimes an incomplete expression is unintentional, and completing it produces an incorrect expression that results in an error message. That's OK; no harm is done, and the expression must be completed or aborted before you can continue. Even entering `\\` to end the session will not work within an incomplete expression.

Statements are sometimes broken at semicolons for readability in this manual, but the `>` characters in the prompt are not included unless there is a reason to do so. However, you will see them when you enter the broken statements in a session.

A TOUR OF K

Introduction

This tour of K is just that: a general overview of the language and its application development features. As we will see, the principal application components — graphical user interface, file I/O, interprocess communication — are very straightforward and easy to use, and therefore most of this tour is about the core of the language. The first part of the tour is designed to give you some familiarity with K. After that, we will set up a simple, yet meaningful, application and add to it as the tour proceeds.

First of all, the language has the traditional control constructs `If`, `Do` and `While` as well as a conditional evaluation construct similar to the one in the C language. A programmer could get by with these and very little else beyond the utility libraries, and you may choose to do so at first. However, that would not be an effective use of the language in the long run. The way to use K effectively is through its primitive functions. The compound data structure in K is the list, and K provides a well-designed set of interacting primitive functions for their manipulation. All the primitive functions are associated with symbols, and all these symbols can be used in expressions in the common way that arithmetic symbols are used, e.g. `a + b * y`. A large part of the challenge of programming in K is constructing concise expressions, and for successful programmers, it is also a large part of the pleasure.

Substantial use of symbols is not uncommon in programming languages. All the non-alphabetic and non-numeric characters on the standard US keyboard have at least one meaning in K, and all except `@`, `$` and ``` have at least one meaning in C. In fact, the total number of meaningful symbols and symbol-pairs in C is greater than

in K. However, K, like APL, associates more functionality with each symbol than C, principally by assigning two primitive functions to most symbols in the tradition of the symbol “-” in arithmetic, which is used for the two functions Minus ($a - b$) and Negate ($- x$).

All the examples in the manual are displayed as they would appear in an interactive session. Before starting the tour, read the chapter Programming Environment for an introduction to using K. Even if you do not have access to K, that chapter will explain the format of the examples in the tour. If you do have access to K, you should execute the examples as you go through the tour, and by all means make up some of your own. Whenever you want to exit from a K session, simply type `\\` alone on a line and press Return.

Familiar Symbols with Familiar Meanings

All symbols on the standard US keyboard are significant in the K language and, more often than not, have more than one meaning. Consequently there is a lot to learn, but not an overwhelming amount unless you try to learn it all at once. The tour begins with some familiar symbols that have well-known meanings.

As in arithmetic, `+` denotes addition and `-` denotes subtraction. For example:

```
5 + 12
17
5 + 12 - 4
13
```

Multiplication is denoted by `*`, which is common among programming languages:

```
5 * 12
60
```

The relational functions are `<` for Less, `>` for More (or Greater), and `=` for Equal. The result of a relational function indicates whether or not the relation is true for the arguments, with 1 for True and 0 for False.

```
5 < 12           5 > 12           5 = 12
1                0                0
```

Relational functions also apply to non-numeric data:

"s" < "h"	"s" > "h"	"s" = "h"
0	1	0

Each of the functions illustrated so far is dyadic, meaning that it has two arguments. The symbol `-` can also be used monadically, i.e. with one argument, which is placed to the right of the symbol, as in `- 3`:

- 3		
-3		
27 + 5 * - 3	27 Plus 5 Times Negate 3	
12		

Why does the symbol `-` in the last expression denote Negate and not Minus? Because the symbol `*` is to its immediate left, not data that would serve as the second argument.

Finally, the monadic form of the symbol `~` denotes Not, or logical negation. That is, `~1` is 0 and `~0` is 1:

~ 1	~ 0
0	1

The function Not in conjunction with the three relational functions above gives three other common relational functions:

~ x < y	x More or Equal y
~ x > y	x Less or Equal y
~ x = y	x Not Equal y

Data Types and Terminology

All the examples so far use individual integers and characters, which are atomic data called —not surprisingly— atoms. There are other atomic data types, and there are compound data structures called lists. Two other atomic types are floating-point numbers and symbols. Floating-point numbers are specified in the usual ways, in both decimal and exponential format. Symbols consist of one, two, or more characters, much like integers are composed of one, two, or more decimal digits. Symbol constants are denoted by back-quote followed by their character contents. For example:

<code>`abc</code>	an atom whose contents are three characters
-------------------	---

There are other data types as well, but these four are the basic ones; the others will be introduced later.

You have already encountered lists in the discussion of data display formats in the chapter Programming Environment. To recall, a constant list of atomic numeric items can be entered simply by entering the items separated by at least one space, as in:

```
1 -2 0 45 9
1 -2 0 45 9
```

which is a list of five integers, and:

```
"axw eql"
"axw eql"
```

which is a list of seven characters. Note that more than one space between the numeric items in the first example are redundant, while the space between the characters “w” and “e” in the second example is the blank character; it is part of the data. Moreover, when only one character appears between double-quotes it is an atom, not a one-item list (see Join and Enlist for a discussion of one-item lists). In addition:

```
1 3.4 5 2.97
1 3.4 5 2.97
```

is a list of four floating-point numbers and:

```
`ibm `sun`apple
`ibm `sun `apple
```

is a list of three symbols. Note that spaces between the symbol items are not necessary, but one space is used in the display format for readability.

In the preceding list of floating-point numbers, it was not necessary to give the whole number items 1 and 5 in a floating-point format, nor are they displayed in a floating-point format. As long as one item in a numeric list of this form is given in floating-point format, all items will be floating-point numbers. A special display format is used when all the items in a list of floating-point numbers are whole numbers:

```
1 -3e0 6 7 2
1 -3 6 7 2.0
```

As before, if one item in this input form is in decimal or exponential format then all items become floating-point. And in this particular case, the last item is displayed in decimal format to indicate that it is a list of floating-point numbers.

Items of lists are not necessarily of the same type, although they will be for constant lists defined in the manner of the preceding examples. More generally, lists can be defined by surrounding all items with parentheses and separating items with semicolons, as in:

```
(761; "a"; `sym; ; 123.82)
(761;"a";`sym;;123.82)
```

This is a list of five atomic items of different types: an integer followed by a character, followed by a symbol, followed by a special atom called *nil*, followed by a floating-point number. The display is like the input, except that redundant blanks are removed. The *nil* value is implied by the absence of any value between the third and fourth semicolons.

More generally still, the list items can be other lists, and their items can also be lists, etc. For example:

```
(3 4 71 -4; "abcdemnklop"; (`abc; -5.6 7.03))
(3 4 71 -4
 "abcdemnklop"
 (`abc
 -5.6 7.03))
```

The first item of this list is the list of four integers 3 4 71 -4, the second item is the list of eleven characters "abcdemnklop", and the third item is a list with two items: the first the symbol atom ``abc` and the second the list of two floating-point numbers -5.6 7.03. The display format now shows the items vertically. Observe that the indentation of each item indicates its level of nesting, or depth.

With regard to terminology, lists whose atoms are all of the same type are called *homogeneous* lists, i.e. all atoms are characters, or all integers, or all floating-point numbers, or all symbols. In particular, a homogeneous list whose atoms are characters is called a character list. If all the items in a character list are atoms then the list is called a character *vector*. If the value of a variable *x* must be either a charac-

ter atom or character list, we say simply that x is character. Similarly for integers, floating-point numbers, and symbols. For example, "abc" is a character vector, 1 2 3 4 is an integer vector, 1.2 3e4 is a floating-point vector, and `a `bb `ccc is a symbol vector. A list whose items are all atoms, but not necessarily of the same type, is called a *simple* list.

Character vectors are also called character strings, or simply *strings*. A character list with the property that every atom is an item of a string is called a string list, and one whose items are all strings is called a string vector. For example:

```
("xyz"; "$2.03")           this is a string vector
("ab"; ("xyz"; "$2.03"); "123.34") this is a string list
("ab"; ("z"; "$2.03"); "123.34")  not a string list
```

The last example is not a string list because the atom z is not an item of a string, as it was in the first example.

The general parentheses-semicolon form of entry can also be used for vectors. For example:

```
("a"; "b"; "c")           input
"abc"                       the display form indicates character vector
(`a; `bb; `ccc)           input
`a `bb `ccc                 the display form indicates symbol vector
(1; 2; 3; 4)               input
1 2 3 4                     the display form indicates integer vector
(1.5; 2.0; 5e1)           input
1.5 2 50                   the display form indicates floating-point vector
```

Observe the second item in the last example, which was entered as 2.0 but displayed as 2. Eliding the decimal format in the display is permissible because the presence of 1.5 as the first item and the format of the display is enough to indicate floating-point vector (the entry 2.0 would have displayed as 2.0 in a context that required decimal format). However, eliding the decimal format in the input would have produced a different result.

Unlike the rule described earlier for entering floating-point lists as numbers separated by spaces, if in the last example the second item had been entered as an integer, the result would not be a floating-point vector, but rather a list whose first and third items are floating-point and whose second item is integer:

(1.5;2;5e1)	input
(1.5;2;50.0)	not a floating-point list

A list whose atoms are either integers or floating-point numbers is said to be a numeric list, but a numeric list whose items are all atoms is not necessarily called a numeric vector; that term is reserved for a list that is either an integer vector or floating-point vector.

You have most likely observed by now that K displays some lists vertically and some horizontally. In general, lists are displayed vertically down to atoms and vectors, while vectors are displayed horizontally no matter how long, if necessary wrapping around onto more than one line.

The Familiar Symbols Applied to Lists

Before introducing lists in the previous section we discussed some familiar symbols and illustrated their meanings with examples using atomic arguments, such as $3 + 4$. In this section we will see how these functions apply to lists. For example:

3	4	5	+	2	0	9
5	4	14				

The two lists are added item-by-item; the first item of the one on the left is added to the first item of the one on the right to produce the first item of the result ($3 + 2$ gives 5), and similarly for the second items ($4 + 0$ gives 4) and for the third items ($5 + 9$ gives 14). Plus applies independently to the atoms in its arguments, and for that reason is called an *atomic function*.

If one argument to Plus is an atom and the other is a list, then the atom argument is paired with every atom in the list argument:

10	+	6	3	2	9	3	-4	5	+	1	
16	13	12	19	4	-3	6					

Not any pair of numeric lists can be used as the arguments to Plus. They must have the same number of items so that the items match in the manner illustrated above. Even when the lists are not vectors, the item-matching rule still applies to every pair of items, and every pair of items within these, and so on down to atoms. Two numeric lists that are a valid pair of arguments to Plus are said to be *conformable*.

Consider the example:

```
(3 4 8; 4; 2 7) + (10; 30 20 50 40; 200 300)
(13 14 18
 34 24 54 44
 202 307)
```

This example shows that the structure of the result is not necessarily identical to that of either argument, but is something like the “least common multiple” of the two. Both arguments have three items, and Plus is applied to the three pairs of corresponding items from the two, as in:

```
    3 4 8 + 10
13 14 18          item 0 Plus item 0
    4 + 30 20 50 4
34 24 54 44      item 1 Plus item 1
    2 7 + 200 300
202 307          item 2 Plus item 2
```

The first item of the result, 13 14 18, has three items itself, like the first item of the left argument, because the first item of the right argument is an atom. The second item of the result, 34 24 54 44, has four items, like the second item of right argument, because the second item of the left argument is an atom. Finally, the third items of both the left and right arguments have two items, and therefore so does the third item of the result. In general, the arguments can be deeper numeric lists, but the application of Plus to them can be traced in essentially the same way. Note that at any point in the descent through the arguments, if an atom is encountered in one of the arguments the corresponding item in the other one can be a numeric atom or any numeric list.

All the other dyadic functions associated with familiar symbols are atomic functions like Plus: Minus, Times, Less, More and Equal. The two monadic functions, Negate and Not, are also atomic functions. There is no notion of conformable arguments in the monadic cases because there is only one argument. Rather, a monadic atomic function is simply one that applies independently to all atoms in its argument. For example:

```

- (1 2; 5; -8 4 1 2)
(-1 -2
-5
8 -4 -1 -2)

```

The structure of the result is identical to the structure of the argument.

Familiar Functions on Somewhat Familiar Symbols

The other atomic functions are Divide and Reciprocal, Max, Logical Or, Min, Logical And, Power and Floor. These are found in most programming languages, and sometimes they are associated with symbols. The symbols used in K reflect the symbols used in other languages, or are suggestive of what the functions do. For example, Logical Or is denoted by | and Logical And is denoted by & in both K and C. Both Divide and Reciprocal are denoted by % for “x divided by y”, and Power is denoted by ^ for “x raised to the power y”, both in K and Mathematica.

Logical And and Or apply to boolean atoms 0 and 1. Since they are atomic functions they can be evaluated for all combinations of atomic arguments at once:

```

0 0 1 1 | 0 1 0 1
0 1 1 1

```

All combinations of boolean atomic arguments for Logical Or

```

0 0 1 1 & 0 1 0 1
0 0 0 1

```

All combinations of boolean atomic arguments for Logical And

An examination of these results shows that the Logical Or of any two boolean atoms is also the maximum value of the two atoms, and the logical And is also the minimum value. Consequently the symbols for Logical Or and Logical And are also used for Max and Min of non-boolean arguments, respectively. For example:

```

5 -4 | 2 7
5 7

```

```

5 -4 & 2 7
2 -4

```

Reciprocal is a monadic function that can be defined in terms of the dyadic function Divide, just as the monadic function Negate can be defined in terms of the dyadic function Minus:

% x equals 1 % x just as - x equals 0 - x

Floor of x, which is a monadic function denoted $\underline{_}$ x, is the largest integer less than or equal to x. For example:

```

  2.5 7 0 -2.5 -7
  2 7 0 -3 -7

```

Floor is used in the monadic expression $\underline{_}0.5+$ to round its argument to the nearest integer:

```

  0.5 + 2 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9
  2 2 2 2 3 3 3 3 3

```

Be careful with the use of space characters around Floor. K permits the underscore character to be part of names (though only reserved names may have underscore as their first character). Thus, to be safe Floor should not have letters nor digits immediately adjacent to it.

This concludes the tour of the atomic functions. Note that of all the symbols introduced so far, only $-$ and $\%$ have both a monadic and dyadic use as atomic functions.

Join and Enlist

The remaining primitive functions are not atomic functions, i.e. do not simply apply independently to the atoms in their list arguments. Two of these are the fundamental list constructors called Join and Enlist. All lists can be built by repeated applications of these two functions to atoms.

Join is the dyadic function denoted by comma, and Enlist is the monadic function. Join creates a new list from its two arguments by appending the right argument onto the end of the left argument:

```

  1 2 4 3 , 10 -4 1 9 6 5
  1 2 4 3 10 -4 1 9 6 5

```

Notice the items in the result corresponding to the left argument have the same position in both lists, while the items corresponding to the right argument are shifted over in the result by the number of items in the left argument. In this example, for instance, the third item of the result is 4, which is also the third item of the left argument, while the ninth item of the result, 6, is the fifth item of the right argument, and 9 is 5 plus 4, where 4 is the number of items in the left argument.

There are no restrictions on the arguments to Join: any two lists can be joined together, and any atom can be joined to any list.

Enlist is the monadic function denoted by comma that creates a one-item list with its argument as the single item. For example:

```
, "a"  
, "a"
```

In this simple example, Enlist is applied to a character atom to make a one-item character list holding that atom. Observe that the comma is used in the display format to indicate a one-item list. This is the first one-item list we have encountered; there has been no way to create or display a one-item list up to now.

The following illustrates how any list can be built with Join and Enlist. Consider:

```
("abcd"; `s `t; 1 3 2)  
("abcd"  
 `s `t  
 1 3 2)
```

Each of the items can be built with Join, as follows:

```
"a", "b", "c", "d"  
"abcd"  
 `s, `t  
 `s `t  
 1, 3, 2  
1 3 2
```

However, these three components cannot simply be joined together, for that would form a nine-item atomic list.

```
("a", "b", "c", "d"), (`s, `t), (1, 3, 2)  
("a"; "b"; "c"; "d"; `s; `t; 1; 3; 2)
```

Instead, each of the components must first be enlisted, and the resulting one-item lists are then joined together to form a three-item list:

```
(, ("a", "b", "c", "d")), (, (`s, `t)), (, (1, 3, 2))
("abcd"
`s `t
1 3 2)
```

Read this expression carefully and make sure you know which of the commas denote Join and which denote Enlist. (If there is data immediately to the left it is the dyadic case Join, and otherwise it is the monadic case Enlist). Note that no commas indicating one-item lists appear in the display format of this list; even though Enlist was used to form the list, none of the items is a one-item list in the result.

Assignment and Indexing

We are now at a point where it is useful to be able to assign values to names for future reference and index into lists to select or replace specific items. Assignment is denoted by the colon. In its simplest form a name appears to the left of the colon and a value to the right, and the effect of the colon is to associate the value with the name. The value of a name can be displayed simply by entering the name alone on a line. For example:

```
aBc: 1 2 3           the name aBc gets the value 1 2 3
aBc                 enter the name alone to get its value
1 2 3               the value of aBc
```

Observe that no value was printed after the assignment statement was executed. This is always the case when the last thing done on a line is assignment with a colon; the return value is nil.

As is common, a name and its data value are often called a *constant* or a *variable* depending on the context in which they are used, where the value is fixed or changing.

There are several forms of selection-by-index in K, but we will use only the bracket form for now because the use of brackets or parentheses for indexing is common and likely to be familiar.

Any items of any list can be selected by indexing. The first item is selected by index 0, the second item by index 1, and so on. The largest valid index depends on the list being indexed, and is one less than the number of items in that list. For example:

```

"abcdefghijk"[0]      "abcdefghijk"[1]
"a"                   "b"
"abcdefghijk"[10]
"k"

```

The index itself can also be a list, and in fact any list whose atoms are all valid indices of the list being indexed, i.e. from the set $\{0, 1, \dots, n-1\}$, where the list being indexed has n items. The items of the index list need not be in any particular order, and duplicates are allowed. For example:

```

"abcdefghijk"[10 0 1]
"kab"

```

and somewhat more complicated:

```

"abcdefghijk"[(0 1; 10; (6 2 6 7; 8 3 4))]
("ab"           selected by the first item of the index, 0 1
 "k"           selected by the second item of the index, 10
 ("gcgh"       selected by the first item of the third item of
 "ide"))       the index, 6 2 6 7
                selected by the second item of the third item
                of the index, 8 3 4

```

The structure of the result $x[i]$ conforms to that of the index i . In fact, the result can be produced by copying the index, and replacing each atom in the copy with the item of x it selects. For instance, the following display shows the result and the index list of the last example side-by-side:

```

("ab"           (0 1
 "k"           10
 ("gcgh"       (6 2 6 7
 "ide"))       8 3 4))

```

Replace each integer on the right with the character in "abcdefghijk" that it selects, and result will be the list whose display is on the left.

In the examples so far the lists being indexed are simple lists, but any list can be indexed, and all the general rules illustrated above apply. For example:

```

(1 2 3; `a `v `sde `j; "wor")[(1 0; 2 2)]
((`a `v `sde `j           selected by the first item of the first item of

```

1 2 3)	the index, 1
("wor"	selected by the second item of the first item of the index, 0
"wor"))	selected by the first item of the second item of the index, 2
	selected by the second item of the second item of the index, also 2

The items of any variable whose value is a list can be replaced by the combination of bracket indexing and assignment called index-assignment. For example:

x: "abcdefghijk"	
x[2]: "C"	replace the item at index 2 with "C"
x	
"abCdefghijk"	lower case "c" has been replaced with upper case "C"

As before, the index can have any structure so long as its atoms are all valid indices of the list to be modified. The value to the right of the colon must conformable with the index, much like the left and right argument of Plus must be conformable. For example:

```

x[0 1 0 4]: "AKE"
x
"AbCdEfghijK"
x[(0 1;2 0)]: ("X";"YZ")
x
"ZXYdEfghijK"

```

In the last example the first replacement to occur is for the index 0 in the first item 0 1 of the index, and the first item "A" of x is replaced by "X"; the second replacement is for the index 1 in 0 1, and "b" is also replaced with "X"; the next replacement is for the index 2 in 2 0, and "C" is replaced with "Y"; and finally, the last replacement is for the index 0 in 2 0, and "X", which had previously replaced "A", is replaced with "Z".

The only time that the order in which the replacements are carried out matters is when there are repeated atoms in the index — e.g., 0 in the above example — which cause repeated replacement of the same items. Indexing order is left-to-right, therefore in such cases the last of the repeated replacements are the ones that count.

There are no restrictions on the items that can be replaced or on the values of their replacements:

```
x[(1;8 3)]: ("0123";(`a `bc;9 2.3 8))
x
("Z"
"0123"           the "x" in x was replaced by "0123"
"Y"
9 2.3 8         the "d" was replaced by 9 2.3 8
"E"
"f"
"g"
"h"
`a `bc         the "i" was replaced by `a `bc
"j"
"K")
```

We will return to assignment and indexing further along in the tour.

Where

How would you select all the items in a floating-point list whose values are greater than 10? In more conventional languages you might start with a result list of the same size and a result count of 0, loop through the floating-point list looking for items that satisfy the condition, and when one is found, insert it in the result list and increment the result count by 1. In K all items can be tested at once, the resulting boolean list is turned into a list of indices, and the floating-point list is indexed all at once to produce the result list. In particular, the result count is simply the length of the result list and does not have to be computed separately. The primitive function that turns boolean lists into lists of indices is called *Where*, and is the monadic function of the symbol `&`. For example:

```

nl: 1 47 -34.6 67.021 0 -2
nl > 10                the boolean test
0 1 0 1 0 0           where items of nl are greater than 10
& nl > 10             the indices where the test is True
1 3
nl[& nl > 10]         the items for which the test is True
47 67.021

```

Order of Evaluation, and the Way to Read Expressions

We have already encountered quite a few expressions involving two or more primitive functions, but have not discussed the order in which these primitives are evaluated within the expressions. Most programming languages follow the lead of conventional mathematical notation and assign a precedence for evaluation among their primitive functions. For example, in evaluating the following mathematical expression:

$$1 - 2x + 4x^2$$

the power function “x squared” is evaluated first, then the two multiplications “2 times x” and “4 times the result of x squared”, and finally the subtraction and addition.

There is no precedence for primitive function evaluation in K; no function is evaluated before another independent of their relative positions in an expression. Instead, K employs the simple rule that in the absence of parentheses, all evaluation is strictly right to left. For example, copy the above mathematical expression item-by-item into a K expression as follows:

$$1 - 2 * x + 4 * x ^ 2$$

Does this expression give the same value? No. Following the K evaluation rule, “x squared” is evaluated first because it is the right-most function. The next evaluation is “4 times the result of x squared”, because it is the next function from the right. The next evaluation is “x plus the result of 4 times the result of x squared” — not “2 times x” as in the mathematical expression — because it is the third function from the right. The next evaluation is 2 times the result of the previous evaluation, and the last is 1 minus the result of that evaluation. The fully parenthesized equivalent expression is:

$$1 - (2 * (x + (4 * (x ^ 2))))$$

Parentheses must be used to reflect the mathematical conventions of higher precedence for exponentiation and multiplication, as well as left-to-right associativity of addition and subtraction. The K expression that gives the same result as the mathematical expression is:

$$(1 - (2 * x)) + (4 * (x ^ 2))$$

or more simply:

$$(1 - 2 * x) + 4 * x ^ 2$$

The evaluation rule means that in the absence of punctuation, the right argument of a dyadic primitive function or the (only) argument of a monadic primitive function is everything to the right of the symbol. For example, in the last expression the right argument to Plus is everything to its right, while the right argument to Times in $2 * x$ is simply x , because that x is followed immediately by a right parenthesis. Another example is:

$$\& n1 > 10$$

from the preceding section (Where), where the argument to $\&$ is $n1 > 10$, which is everything to its right. This “everything to the right” rule is more formally known as *long right scope*.

Because K has long right scope, the most generally effective way to read K expressions is from left-to-right, in the opposite order from which they are evaluated. For example, $\&n1>10$ reads quite naturally as “where $n1$ is more than 10”; $n1 [\&n1>10]$ as “the items of $n1$ where $n1$ is more than 10”; and $(1-2*x)+4*x^2$ as “1 minus 2 times x , plus 4 times x to the power 2.” It should prove worthwhile to use this reading technique during the rest of the tour.

PREPARING THE SAMPLE APPLICATION

It is always better to learn about a language by using it in real situations, and so in this chapter we'll set up a sample application that will be developed further as the rest of the tour proceeds. For various reasons related to the limited amount of data base information that can be displayed on the printed page, the amount of data we will use is very small in comparison to that of the actual application. But no matter, because the application developed here will run as is with realistic amounts of data, and realistic transaction rates as well. This scalability is one feature of K that makes it so useful in practice. Later on you will see how to manufacture large amounts of test data for this application.

Two new things are introduced in this chapter, commands and directories. Commands are statements outside the core language that have a variety of different meanings, much like operating system commands; the only thing they have in common is syntax. (They are outside the core language for two reasons: one, they do not have explicit results, and therefore cannot be used like primitive functions to form expressions; and two, they do not apply to atoms and lists.) There is no separate tour section for commands; they are simply introduced as needed. Also, it is assumed that the general idea of directories is understood, and even though K gives them their own flavor, the term is used here with minimal introduction. Nevertheless, they do require some discussion eventually, and that occurs in a later section called The K Name Space.

The sample application is an ATM system, where customers use ATM machines to carry out simple banking transactions. During the interaction with a customer the ATM communicates with a central server that determines whether or not a particular transaction can be carried out, and if so, records the transaction. Transaction locks and logs are maintained in the server.

The sample application will consist of two separate processes, one representing the central server and the other an ATM machine. Inserting a credit card in the machine and entering the PIN will be simulated by simply typing the credit card number in an appropriate screen window. A list of transactions will then be displayed. For simplicity the only active transaction will be the most common one, withdrawing cash from checking accounts. When that transaction is selected a window will be displayed for entering the amount of the withdrawal. The ATM process will communicate with the server to determine whether or not the withdrawal can be done. If it can, the server logs the transaction, updates its database, and sends a message to the ATM to dispense the cash. Otherwise, the server sends a message to that effect to the ATM process, which will then display its own message explaining why the transaction failed.

Data Organization and Display

For the moment we do not need to run two separate processes; it is enough to keep the code for the two processes separate in one process. We will do this by allotting the code for each process a separate place in K's hierarchical name space (see the section The K Name Space that follows). We will start with the server as follows:

```
\d .server
```

This is a command (as indicated by the initial `\`); it creates a top-level directory named `server` (if it does not already exist) and makes `.server` the so-called working directory, a term that will be explained in the next section. All global variables for the server will be placed in this directory.

The server has access to the central records of the bank, which we will simulate as 10 records, each with a name field, a credit card number field and a field for the start-of-day checking account balance. We will call this table `customers`, and the fields will be `name`, `number`, and `balance`. In order to create this data, begin with:

```
\d customers
```

which creates the directory `customers` as a subdirectory of `.server`, and makes it the working directory. Within this directory the fields are defined as follows:

```
name: ("Jones, B.G."; "Thomas, I.B."; "Roe, E.P."
      "Layne, B.F."; "Irwin, H.G."; "Marshall, C.J."
      "Blackwell, E.W."; "Rowan, A.M."
      "Rogers, F.M."; "Crosley, F.C.")
number: 6358607 4153802 5251491 5667632 4031685
number: number, 7313067 9371526 9855533
number: number, 8757384 4927607
balance: 633.58 105.77 533.87 468.00 988.97 310.16
balance: balance, 722.63 927.68 554.54 6.60
```

All three lists could be entered on one line, although that line may wrap around more than one physical line, but were entered on more than one line here to improve readability. (Breaking the definition of `name` into several lines is explained in *Incomplete Expressions* in the chapter *Programming Environment*.) The vectors `number` and `balance` are entered on more than one line, but there are no syntactically incomplete statements involved. Instead, each is initialized to part of its final value in one statement, and then the remainder is catenated on in a second statement. You could define these vectors in a way similar to `name`, as in:

```
balance: (633.58; 105.77; 533.87; 468.00; 988.97; 310.16
         722.63; 927.68; 554.54; 6.60)
```

but this seems somewhat inconvenient, particularly for long vectors. Also, you must be careful with floating-point vectors. In the vector form above, the whole number 486.00 could have been entered as the integer 468 and `balance` would still have been a floating-point vector, but not in the semicolon-parentheses form, where `balance` would then become a numeric list with one integer atom among floating-point atoms.

Here's a little exercise: what does the following definition of `balance` yield?

```
balance: (633.58 105.77 533.87 468.86 988.97 310.16
         722.63 927.68 554.54 6.60)
```

Returning to the sample application, we will simulate a database by writing this table to a file. First, go back up one level from the `customers` directory to the `.server` directory:

```
\d ^
```

and then write the table to a file:

```
"customerDb" 1: customers
```

Almost any data object can be written to a file, and directories are data objects. The symbols `1:` denote a dyadic function that writes an object to a file. Just as both `Minus` and `Negate` are denoted the same, there is a monadic function denoted by `1:` as well, which will be used later in this section.

To begin work on the ATM side, define a new top-level directory and make it the working directory:

```
\d .atm
```

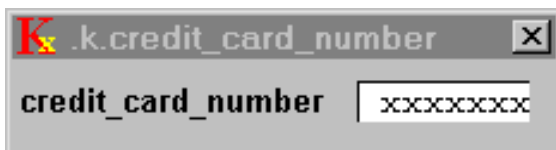
We need a screen display for customers to enter credit card numbers, another for the transaction list, and a third for customers to enter the amount of the withdrawal. In the first and last cases we'll start with character vectors that suggest the location of the entry area in the screen displays:

```
credit_card_number: "xxxxxxx"  
cash_amount: "xxx"
```

Either of these objects can be displayed on the screen, simply by saying “show it”, as in:

```
`show $ `credit_card_number
```

The display will look like:



Customers can type in the entry fields, and the object automatically takes on the entered value. This is true of any global variable displayed on the screen as data: type a new value on the screen and the object changes accordingly. Or, change the value of any object with an assignment expression and its screen display changes. In K applications, what you see and interact with on the screen is simply a visual manifestation of a K data object.

The display of any object can be removed from the screen by saying “hide it”, as in:

```
`hide $ `credit_card_number
```

If you are working alongside in a K session, define `credit_card_number` and show it as above, then type something in its entry field and press Return, compare that entry with its value by typing its name alone on a line (and pressing Return), and then redefine its value by typing in an assignment statement and watch its screen value change.

We will save the initialization of the ATM process in what is called a script file, which is simply a text file holding K expressions and programs (which have yet to be discussed). When a script file is loaded into a K process, the lines of text are executed from top to bottom as if they had been entered at the keyboard. Script files have the extension “.k” or no extension.

Create a text file named `atm.k` and enter the following four lines:

```
\d .atm
credit_card_number: "xxxxxxx"
cash_amount: "xxx"
`show $ `credit_card_number
```

Save the file. When the load command:

```
\l atm
```

is entered, the two variables will be created and `credit_card_number` will appear on the screen, as in the above display. Or, the name of the script can be given on the command line when K is started, as in

```
k atm
```

Returning to the server part of the application, a script file must eventually be created for the server as well. Let’s start now. Create a text file named `server.k` with just one expression in that file, which loads the database:

```
\d .server
customers : 1: "customerDb"
```

To understand this expression, read it from the left: the variable `customers` is assigned the value of the expression `1: ".k.customerDb"`. The symbols `1:` denote a monadic function that is applied to the character vector `"customerDb"`

and brings the data object stored in the file named `customerDb` into the current session. The dyadic use of `1 :` is for writing data objects to files (see above). Save the file.

The last thing to do in setting up this application is to prepare for communication between the two processes (see the chapter Programming Environment). You will need two consoles or shells for running two K processes. The server process must specify a communication port number in its startup command which the ATM process uses to initiate communication with the server. To start the server process, enter the following command in one of the two consoles or shells:

```
k server -i 1234
```

and then enter

```
k atm
```

in the other to start an ATM process. Do this now to make sure your script files are defined correctly.

Developing this application further amounts to appending variable and function definitions to the two script files and then restarting the server and ATM processes with the above commands. The transaction list will be constructed later.

The K Name Space

K has a hierarchical name space, with directories containing other directories and non-directories, all under one root directory, which is called the K name space or K-tree, or when the context of the discussion is clear, simply the name space or the tree. Directories are ordinary data objects in K, with their own data type, and therefore anything in the name space is a valid data object. Every object in the name space has a unique compound name of the form:

```
.simpleName1.simpleName2.—.simpleNameN
```

A simple name contains one or more alphabetic, numeric, or underscore (`_`) characters and must begin with an alphabetic character. (See the section Attributes for the meaning of names containing optional dots). Then all simple names in a compound name are names of directories except, possibly, the one furthest to the right. When two simple names are adjacent to one another in a compound name the one on the

left is said to be the parent of the one on its right, which in turn is a child or entry of the one on its left. If the two simple names are not necessarily adjacent their relationship is ancestor-descendent instead of parent-child. In the compound name `p.q.r`, the name `p` is the parent of `q` and an ancestor of both `q` and `r`, while `r` is an entry in `q` and a descendent of both `p` and `q`.

The left-most dot in the compound name displayed above refers to the root directory of the name space, and the entries in the root directory such as `simpleName1` are said to be top-level objects. The above compound name is called an absolute name because it provides the full path to the object `simpleNameN`. A simple name identifies an object relative to its parent directory, and a compound name that does not begin with a dot identifies an object relative to the parent directory of its left-most simple name; consequently any name that does not begin with a dot is called a relative name. For example, the object referred to by the absolute name `.p.q.r.s` is also referred to by the relative name `s` (relative to its parent `.p.q.r`), by `r.s` relative to `p.q`, and by `q.r.s` relative to `p`. All objects with simple names relative to a given directory are called entries in that directory.

We will create a sample name space for experimentation by starting a new `K` process and loading both the server and atm scripts. Enter:

```
k
```

on the command line of a shell, which starts a `K` session, and then enter the following `K` command in that session:

```
\d
.k
```

This command displays the so-called *working directory*, in this case the default `.k`. It is called the working directory, and often the *current directory*, because any objects with relative names created from now until the working directory is changed will be relative to this directory. For example, if:

```
a: 1 2 3
b.x: `xyz `abcd
```

then `.k` is the parent of both `a` and `b.x`, and `.k.a` and `.k.b.x` are the absolute names of these objects:

<code>.k.a</code>	<code>.k.b.x</code>	absolute names
<code>1 2 3</code>	<code>`xyz `abcd</code>	
<code>a</code>	<code>b.x</code>	relative names
<code>1 2 3</code>	<code>`xyz `abcd</code>	

The following command lists all entries in the current directory:

```
\v
a b
```

Note that in creating the object `b.x` above, the directory `b` was also created.

The working directory can be changed with the command `\d name`, where *name* is any simple or compound name, and identifies the new working directory. If *name* is relative then the new working directory is a descendent of the current one, but if it is absolute then there is no necessary relation between the two. For example, enter:

```
\d c.d
```

and the working directory becomes `.k.c.d`. Enter:

```
a:"abcd"
```

and there are now two objects in the name space with the simple name `a`, namely `.k.a` and `.k.c.d.a`. Now enter:

```
\d .k
```

which makes `.k` the working directory again, and:

```
\v
a b c
```

Observe that when we created the directory `c.d` with the command `\d c.d`, the intermediate directory `c` was also created.

The entries in any directory can be listed with the command `\v name`:

<code>\v b</code>	<code>\v c</code>	<code>\v c.d</code>
<code>x</code>	<code>d</code>	<code>a</code>

Now change the current directory to `.k.c` and add the the code for the sample application:

```
\d c                it was .k; now it is .k.c
\l server
\l atm
```

Recall that the script `server.k` creates the directory `.server` and makes it the working directory, and analogously for `atm.k` and `.atm`. Now enter:

```
\d
.k.c
```

Even though the working directory is changed within each script file load, it reverts to the directory `.k.c` in effect before the loads were done. This is always the case when scripts are loaded. Now enter the following command:

```
\d .
```

which makes the root directory the working directory. Make sure you understand each of the following:

```
\v
k server atm
\d server
\v
customers
\d customers
\v
name number balance
\d
.server.customers
\d .atm
```

If you are following along at a computer, you should continue to explore this name space.

Two important points to remember are that absolute names can be used within any working directory and that compound names can be used wherever simple names are used. For example, in the section *Data Organization and Display*, when defining entries in the working directory `customers` we reset the current directory to its parent so that we could write `customers` to a file using its simple name. However, we could have remained in the `customers` directory and written it to a file from there, using its absolute name `.server.customers`, as in:

```
"customerDb" 1: `.server.customers
```

Secondly, earlier in this section we set the working directory to `.k` and then defined `b.x`, which had the effect of creating the directory entry `b` in `.k` and the symbol vector entry `x` in `b`. We can access this `x` by its absolute name from anywhere, as in:

```
.k.b.x[0]  
`xyz
```

or from any ancestor with the appropriate relative name:

```
\d .k  
b.x[1]: `klmn  
\d b  
x  
`xyz `klmn
```

Cross-Sectional Indexing

So far we have concentrated on accessing the items of a list, but the items themselves can have items, which in turn can have items, and so on. In this section we will see how to select and replace items at depth within a list.

Consider the string vector

```
o: ("abcdef"; "ABCDEFGH"; "0123456789")
```

The character "D" in the second item can be selected as follows:

```
o[1; 3]  
"D"
```

Or, the characters "G" and "D" from the second item, in that order:

```
o[1; 6 3]  
"GD"
```

Or, the characters at index positions 6 and 3 in both the second and third items:

```
o[1 2; 6 3]  
("GD"  
"63")
```

This form of indexing is called *cross-sectional indexing*, as the latter example suggests.

If an index position is left blank, it is treated as if all valid indices had been specified. For example:

```
o[;3]
"dD3"
```

selects the item at index 3 from all three items of `o`, and

```
o[2;]
"0123456789"
```

selects all items of the item at index 2. That is, `o[2;]` is identical to `o[2]`.

Observe that `o[1 2;2 0]` is not the same as `o[(1 2;2 0)]`. In the latter case the parentheses serve to form a single list with two items, and this expression is of the form `o[i]`. Consequently four items of `o` are selected:

```
o[(1 2; 2 0)]
("ABCDEFGH"
 "0123456789")
("0123456789"
 "abcdef")
```

On the other hand, there are two index lists in `o[1 2;2 0]`; this expression is of the form `o[j;k]`, and selects the items with indices 1 and 2 from `o`, and then the items with indices 2 and 0 from each of those:

```
o[1 2;2 0]
("CA"
 "20")
```

As in the earlier section on indexing (see *Assignment and Indexing*), this form is completely general. The items of the indexed arrays do not have to be similar in any way, and the selected items at depth do not have to be atoms. Moreover, there can be as many indices separated by semicolons as there are levels to be indexed. For example:

```

m: (1 4 -2; (`x `yz `q_r; "a"); 2.3)
m[1;0;2]
`q_r

```

Observe here that if the first index is 0 then only one other index can be given, which would select items of `m[0]`. Also, if the first index is 2 then the atom 2.3 is selected and no further indices can be specified.

Index assignment extends to this form of indexing: any item at any depth can be replaced by anything, multiple items can be replaced at once, and duplicate atoms within any index cause items to be replaced more than once:

```

m[1;0]
`x `yz `q_r
m[1;0;2 1 2]: ("two"; ("one";1); "TWO")
m[1;0]
(`x                               unchanged
 ("one";1)                       item 1 of item 0 of item 1 is replaced with ("one";1)
 "TWO")                           item 2 of item 0 of item 1 is replaced with "TWO"
                                   (after first being replaced with "two")

```

Symbolic Indexing

When setting up the sample application we defined a directory `customers` in the server directory with three entries, and saved the directory and its contents in a file. That directory is a K data object, as are all directories, and as such has a data type, called *dictionary*. The reason for using two different names for the same thing will be discussed later.

A directory can be indexed in much the same way as a list, except the its entry names — as symbols — are used as the indices. The result consists of the values of those entry names. For example, define a new top-level directory `p`:

```
\d .k.p
```

We are now in the `p` directory. Define the following entries:

```

a: 1 3 5 7
b: "abcdefgh"
c: `x `y `z

```


Now return to the parent directory of p:

```
\d ^
```

Select the value of a from the directory p by using a symbolic index:

```
p[`a]
1 3 5 7
```

Select the values of the entries b and a, in that order:

```
p[`b`a]
("abcdefgh"
 1 3 5 7)
```

Select the items at indices 3, 0 and 1 from both b and a:

```
p[`b`a;3 0 1]
("dab"
 7 1 3)
```

Replace the items of a at indices 3, 0 and 1:

```
p[`a;3 0 1]: 70 10 20
p[`a]
10 20 5 70
```

Indexing dictionaries is exactly like indexing lists except that the indices are symbols holding entry names instead of integers.

Directory entries can be accessed by their compound names as well as by indexing, and compound names can be used in every way that simple names are used. For example, use Assignment to create a new entry in p:

```
p.d: (1 2; `x `y)
```

Now use indexing to reference this entry:

```
p[`d]
(1 2
 `x `y)
```

Directory Assignment and Structure

Directories are ordinary data objects and can be assigned their contents in the ordinary ways. For example, a new directory q can be formed from p simply by assigning q the value of p :

```
q: p
q ~ p
1                                q and p are identical
```

This form of assignment can also be used to reassign the contents of an existing directory, but here we must be somewhat cautious. It often occurs that we want to reassign the values of all entries in a directory but not change the names of the entries, and later on we will see examples of this. We will also see how to establish spreadsheet-like formulas among global variables that automatically maintain the relationships among variables when some of them change values. Re-assigning an existing directory with a simple assignment statement like the one above has the effect of invalidating any spreadsheet-like relationships involving its entries, and thereby leaving the application in an inconsistent state. Consequently we need a way to safely replace the values of all entries in a directory.

We have seen previously that eliding an index means that all possible indices, in their natural order, are used. For example, all values of a directory are selected by:

```
q[]
(1 3 5 7
"abcdefgh"
`x `y `z
(1 2
`x `y))
```

and all values can be replaced, item-by-item, by the corresponding assignment statement:

```
q[]: ("xw$3"; `a `bb `ccc `d; 1 2.4 -4 6; 5 3 1 0)
```

The number of entries of q cannot change as a result of this form of specification. The new value must be a list with as many items as q has entries, or an atom, in which case every entry of q takes on the value of the atom.

(This form of assignment applies to lists as well as directories; for a list x and $x[]:b$, the number of items of x cannot change, if b is a list then $\#b$ must equal $\#x$, and if b is an atom then after the specification, every item of x equals b .)

Here is the way the directory q is displayed in the session log:

```
q
. (( `a; "xw$3";)
  (`b
   `a `bb `ccc `d
  )
  (`c
   1 2.4 -4 6
  )
  (`d
   5 3 1 0
  ))
```

This is the first display of a directory value in the tour, so let's look at it in detail. First of all, the leading dot (all the way to the left on the line under the name q) is like the leading comma in the display of one-item lists. The dot denotes a monadic primitive function that creates a dictionary data object from its list argument. We will discuss this function later on, and dictionaries as well. (Directories are global variables whose values are dictionaries.)

The list that starts to the right of the dot has four items, one for each entry in q , and we will refer to these items as the *entry items*. Each entry item (always) has three items. The first item of each entry item is a symbol holding an entry name, and you can see that q has entries a , b , c , and d . The second item of each entry item is the value of the corresponding entry, and these values are the ones assigned above. Finally, each entry item has a third, empty (nil) item. The third item of a valid dictionary entry will always be either nil , or a dictionary called the *attribute dictionary* of the entry. In the example, they do not yet exist because no attributes have been assigned values. Attributes are discussed next.

Attributes

Every global variable carries along a set of auxiliary information known collectively as its attribute set. For example, a global variable can be displayed on the screen in one of several ways that are referred to as display classes; the display class is specified by setting the display class attribute of the variable to a particular value. The attributes of a global variable all reside in a directory associated with the variable called its *attribute directory*. The name of the attribute directory is the name of its associated variable followed by a single dot. For example, `p.` is the attribute directory of `p` and `x.` is the attribute directory of `x`, etc. Any name that ends in a dot must be the name of an attribute directory.

The compound name `p.x` refers to the entry `x` in the directory `p`; in effect, the compound name is created by appending a dot to the right of directory name and then appending the entry name to the right of that. The same rule applies to entries in attribute directories: append a dot to the right of the attribute directory name, and then append the attribute name. For example, the name of the attribute `x` of the global variable `b` is `b` followed by dot (signifying the name of the attribute directory), followed by dot (signifying that an entry name follows), followed by `x`, as in `b..x`.

In general, whenever a pair of dots appears in a compound name, the segment to the right of the pair is an attribute of the global variable named by the segment to the left.

Can more than two dots appear together in a compound name? No. That would mean that attribute directories could themselves have attributes, and that is not allowed. Except for this restriction, attribute directories are like any other, and in particular are ordinary data. Moreover, the individual attributes are ordinary data objects in every way, and in particular can have attributes themselves. For example, `x..y..z` is the name of the attribute `z` of the attribute `y` of the variable `x` and `x..y.` is the name of the attribute directory of the attribute `y` of the variable `x`.

Some attributes have reserved meanings, such as the display class attribute described above, whose name is `c`. All other entries in an attribute dictionary can be user-defined. However, you should consider all one-character attribute names to be reserved, even if they presently have no defined meaning.

Attributes are associated with names of global variables, not with values. For example, if `x` and `y` are global variables and `y` is assigned the value of `x`, as in:

```
y: x
```

then `y` has not inherited the attributes of `x`. A separate specification of attribute dictionaries is required if you want the attributes of `y` to be the same as those of `x`:

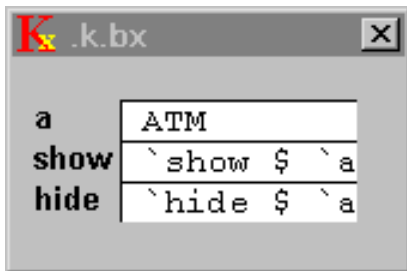
```
y.: x.
```

In this section we will introduce two reserved attributes, the one named `c` for display class mentioned above and the one named `a` for arranging the entries of a dictionary displayed on the screen. In the next section the one named `l` for labeling screen displays will be introduced. This will allow us to create the transaction list for the ATM process in the sample application, but first, let's look at a simple example. We will create a screen display with two buttons, where pressing one button will display the contents of a variable named `a` on the screen, while the other will remove the display of `a` from the screen.

```
\d .k.bx
a: "ATM"
show: "`show $ `a"
hide: "`hide $ `a"
```

If the directory holding these variables is displayed on the screen without setting its display class attribute or that of any of its entries, it will be displayed in default format, which looks like:

```
\d ^
`show $ `bx
```



(Observe that we changed the working directory to the parent of `.k.bx` so that we could refer to `bx` in the show expression by its simple name. However, we could just as well used its absolute name and executed ``show$`.k.bx` without changing the working directory.)

Display class attribute settings affect the appearance and functionality of displays. Any global variable whose value is a character string holding an expression can be displayed as a button; the expression is evaluated when the button is pressed. The global variables `show` and `hide` have this property. To display them as buttons, simply set their display class attribute to ``button`, as follows:

```
bx.show..c: `button
bx.hide..c: `button
```

If either of these settings had been made before `bx` was shown, the display would have the appearance of a layout, or form. (In general, whenever the class attribute of a dictionary entry is set to a value other than the default ``data`, the default display class of the dictionary becomes ``form`). Since `bx` is already displayed, set its class attribute to ``form`, and if you are working along in a session, you will see the screen immediately change:

```
bx..c: `form
```

The variable `a` can be left in the default display class, which is ``data`. (Even though there is a default class known internally to `K`, the attribute `a..c` does not have that value. Instead, `a..c` has no value because none has been set. However, `a..c` can be explicitly set to the default value, for instance when you want to return to the default after setting the display class attribute to something else.)

The screen display will now look like that shown on the left, below:



We want only the entries `show` and `hide` to be displayed in `bx`, because the other entry, `a`, will be displayed separately as a result of the button actions. (The default is to show all entries in a vertical list.) Which entries are shown and how they are arranged is controlled by the arrangement attribute of `bx`, which is named `a`:

```
bx..a: `show `hide
```

The display will now look like that on the right, above. If the `bx` display is in the upper left corner of the screen, move it aside and then press the show button. You should now see the display of `a` in the upper left corner. Press the hide button and that display will go away.

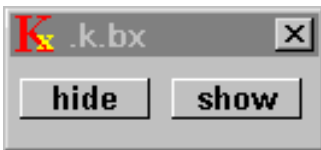
This display can be rearranged simply by re-specifying the order of the entries in the arrangement attribute. For example, set:

```
bx..a: `hide `show
```

and the buttons will immediately change places. Or, set:

```
bx..a: ,`hide `show
```

and the buttons will appear side-by-side; as in:



(Why the latter expression should have this effect is hard to explain because `bx` has so few entries on the screen. We will return to this point in the next section.)

The Sample Application Revisited

Let's return to the ATM process in the sample application and set up the transaction list. There will be four transactions: get cash from checking, get cash from savings, deposit to savings, and balance inquiry. Each transaction will be represented by a button, so that when a button is pressed the corresponding transaction is executed. Since we have no expressions to evaluate right now, the values of all transaction buttons will be the empty character string `"`. Later on we will redefine the button for cash from checking, but the others will remain as is. The transaction list itself will be a directory holding the transaction buttons as entries:

```
\d .atm.transaction_list
cc: ""                cash from checking
cs: ""                cash from savings
ds: ""                deposit in savings
```

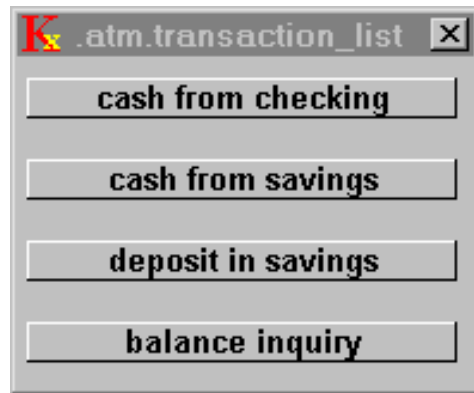
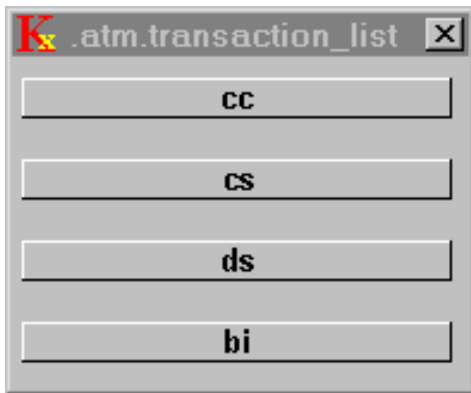
```

bi: ""                balance inquiry
cc..c: cs..c: ds..c: bi..c: `button
                        set all display classes to `button

\d ^
transaction_list..c: `form
`show $ `transaction_list

```

The display of `transaction_list` will look like that on the left, below:



In the previous section the variable names `show` and `hide` were meaningful as the labels when the variables were displayed as buttons. The variable names in the sample application are not so meaningful, but meaningful names are usually long and it is cumbersome to work with long variable names. Fortunately, the text on a screen display can be assigned independently of the variable name, as the value of variable's `label` attribute. If you are working along in a session, you will see each label change as you do the following assignments:

```

\d transaction_list
cc..l: "cash from checking"
cs..l: "cash from savings"
ds..l: "deposit in savings"
bi..l: "balance inquiry"

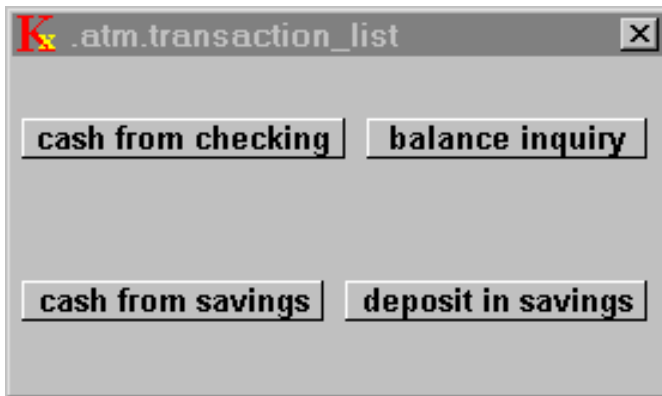
```

Now, the display looks like that on the right, above.

The arrangement attribute of `transaction_list` can be used to rearrange this display. For example, cash from checking can be put above cash from savings in a column on the left, with balance inquiry above deposit in savings in a column on the right, as follows:

```
.atm.transaction_list..a: (`cc `bi; `cs `ds)
```

The display becomes:



You can experiment with other arrangements. Also, `transaction_list` is a long name, and could be shortened to `tl`. The labels at the top of the above displays would then be `.atm.tl`. In that case, it would be possible to get the same look as shown by simply setting the label attribute:

```
.atm.tl..l: ".atm.transaction_list"
```

For any global dictionary `x`, the top level of `x..a` specifies a vertical list in the display whose *i*th row from the top contains the entries named in `x..a[i]`. Each item contributes a horizontal list; the *j*th column from the left in the *i*th row from the top contains the entries named in `x..a[i;j]`. And so on. Each successive level in `x..a` contributes vertical and horizontal lists alternatively. For example, if there are six entries ``a` through ``f`, then the following values of the arrangement attribute have the given meanings:

```
`d`e`f `a`b`c          a vertical list of 6 rows in the specified order
(`a`b; `c`d; `e`f)     a vertical list of 3 rows with 2 columns each
```

`(`a`b`c`d; `e`f)`

a vertical list of 2 rows with 4 columns in the top row and 2 in the bottom row

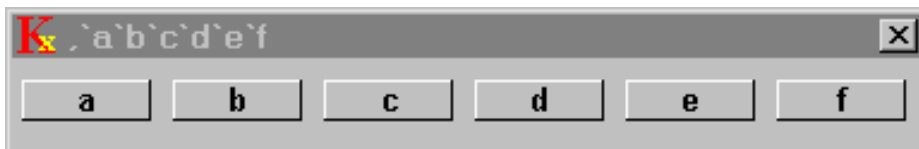
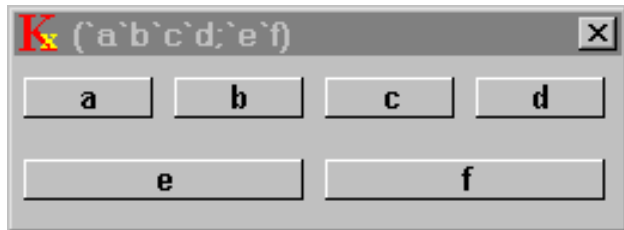
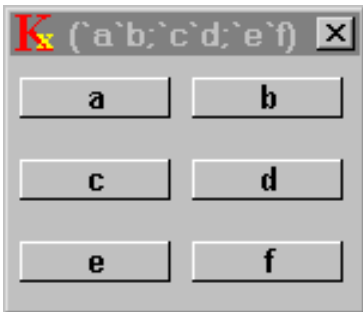
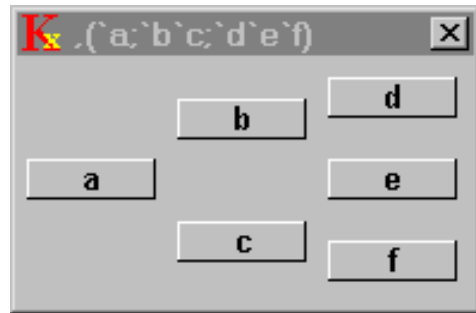
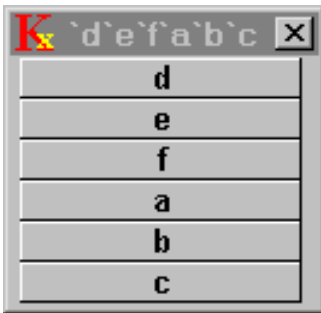
`,(`a; `b`c; `d`e`f)`

a horizontal list of 3 columns with 1 row in the first column on the left, two rows in the second, and three rows in the third

`,`a`b`c`d`e`f`

a horizontal list of all entries (like the hide-show example in the previous section)

In the screen displays below, the label attribute `.l` has been set so that the arrangement of each object is in the title area, and the height attribute `.y` of the first has been set to zero so that the height of the object is minimized.



You can experiment with the arrangement rule by constructing a directory with six or so entries and setting its arrangement attribute to various values. Be sure to set the display class attribute of the directory to ``form`.

The format attribute is used to control the appearance of numeric and character data on the screen. Examples will be given later in the tour.

MORE PRIMITIVE FUNCTIONS

Atom, Count and Shape

We will now return to the primitive functions for awhile. K data objects are self-describing, in that they carry along information about themselves, such as their type, whether they are atoms or lists, and in the case of lists, their item count. There are several primitive functions which produce this information as their results. For example, the monadic function denoted by the symbol @ is called Atom, and its result is 1 if its argument is an atom and 0 if it is a list (i.e., not an atom). For example:

```
@ "a"           @ ""           @ "abc"
1                0                0
```

which illustrates the fact that one character between double-quotes denotes an atom and any other number denotes a list. The monadic function denoted by the symbol # is called Count and its result for a list argument is the number of items in the list; the result is 1 for any atom. For example:

```
# "a"           # , "a"           # "abc"
1                1                3
```

The count of a list is often referred to as its *length*, particularly for character strings.

The monadic function denoted by the symbol ^ is called Shape. It is an extension of Count that includes information about the rectilinearity of its argument. The result of Shape is always an integer vector. For example:

```

^ (1 2 3 4; "abcd"; `w `x `y `z)
3 4
^ (1 2 3 4; "abc"; `w `x `y `z)
,3

```

The argument to Shape in the first expression defines a list of count 3, where all three items have count 4. The shape of this list is the integer vector 3 4. The argument in the second expression also defines a list of count 3, but its items do not all have the same count. The shape of this list is the one-item integer vector , 3.

In general, the shape of a list always has at least one item, and the first item of the shape is the count. If all items of the list are lists with the same count, then the shape has at least two items, and its second item is that common count. If all items of items are lists with the same count, then the shape has at least three items and its third item is that common count, and so on. As long as all items at each successive depth are lists with the same count, those counts will appear as items in the shape.

A matrix is defined to be a list whose items are vectors that all have the same count. The list in the first example above is a matrix, the one in the second example is not. Creating an expression that tests whether or a not a list is a matrix is an interesting exercise. The items of the items of a matrix may all have the same count or not. Consequently a matrix is a list whose shape has at least two items, which means that the count of the shape is at least 2. That is, the data object x is a matrix if the result of the expression:

```
# ^ x
```

Read this expression from left to right:
the count of the shape of x

is at least 2. For example:

```

# ^ (1 2 3 4; "abcd"; `w `x `y `z)
2
# ^ (1 2 3 4; "abc"; `w `x `y `z)
1

```

The expression

```
~ 2 > # ^ x
```

Read this expression from left to right:
not 2 is more than the count of the shape of x

is a test of whether or not `x` is a matrix, in that the value of the expression is 1 if it is a matrix and 0 if it is not.

Observe that reading the last expression from left to right produces the awkward phrase “not 2 is more than...”. In this and like cases, it is best to transpose the `~` and `2` when reading so that the phrase becomes “2 not more than...”.

Finally, there are lists with no items called *empty* lists, whose generic form is denoted `()`. Each type of empty vector has its own specific notation: `!0` for integer, `0#0.0` for floating-point, `0#`` for symbol, and `""` for character. (These notations are explained in the next two sections.) The count of any empty list is 0.

```

# ()          # !0          # 0#`
0            0            0

```

The empty list is an ordinary list that can occur as both function arguments and results. For example, the primitive function `Where` — the monadic function denoted by `&` that was introduced previously — produces a list of indices where the items of a boolean vector equal 1. The boolean vector most likely comes from testing some condition on the items of another list, and if no item satisfies the condition, the list produced by `Where` will be the empty integer vector. For example:

```

& 10 < 1 -12 5 6      read this expression from left to right:
!0                    where 10 is less than 1 -12 5 6

```

There are no items of `1 -12 5 6` greater than 10. A test that at least one item satisfies the condition is:

```

0 < # & 10 < 1 -12 5 6
0

```

We will see another form of this test later.

Some final comments to this section; we said above that the shape of any list has at least one item. The shape of an empty list is naturally enough the one-item list `,0`. In contrast, the shape of any atom is the integer form of the empty list:

```

^ ()          ^ 3          ^ `shape
,0           !0           !0

```

Take

The primitive function `Take` is the dyadic function denoted by `#`. Its purpose is to create lists of specified counts and shapes. It is the most basic list constructor after `Join` and `Enlist`.

The left argument specifies the count or shape of the result and the right argument provides the material that fills it in. For example, an integer atom left argument specifies the count of the result:

```
8 # `a           an atom is replicated as many times as necessary
`a `a `a `a `a `a `a `a
8 # 1 -3 4       the items of a short list repeat as necessary
1 -3 4 1 -3 4 1 -3
8 # "abcdefghij... some items of a long list are not used
"abcdefgh"
```

An integer vector left argument specifies the shape of the result. As in the case of an integer atom on the left, the contents of the right argument are used cyclically to fill in the specified shape:

```
3 4 # `a
(`a `a `a `a
 `a `a `a `a
 `a `a `a `a)
3 4 # 1 -3 4
(1 -3 4 1
 -3 4 1 -3
 4 1 -3 4)
3 4 # "abcdefghij...
("abcd"
 "efgh"
 "ijkl")
```

If the left argument is 0, the resulting list has a count of zero. This helps explain the notation for the empty symbol and floating-point vectors, `0#`` and `0#0.0` respectively, introduced in the previous section.

Enumerate

Another important list constructor is the monadic primitive `Enumerate`, denoted by the exclamation mark. For a non-negative integer atom `x`, `Enumerate` produces a list of the integers from 0 to `x-1`. These integers are valid indices into lists of length `x`. For example:

```
!10                                (!10) [!10]
0 1 2 3 4 5 6 7 8 9              0 1 2 3 4 5 6 7 8 9
```

`Enumerate` also produces symbol vectors from dictionaries and directories, by returning all their entries. Again, there is the connection to indexing, since each item in the return value is a valid symbolic index into the argument. Returning momentarily to the example in *The Sample Application Revisited* from last chapter:

```
\d .atm
!transaction_list
`cc `cs `ds `bi
```

We have already seen `Enumerate` in the notation for the empty integer vector `!0`. Now it is apparent why this notation is used; `Enumerate` always produces an integer list for non-negative integer arguments, in this case a list of length 0.

Match

The primitive called `Match` is the dyadic function denoted by the symbol `~`. The result of `Match` is 1 if its two arguments have identical values, and otherwise it is 0. For example:

```
1 2 3 ~ `a `b                1 2 3.0 ~ 1 2.0 3
0                               1
1 2 3 ~ 1 2.0 3              1 2 3 = 1 2.0 3
0                               1 1 1
```

`Match` is not an atomic function like `Equal`. `Match` indicates whether or not its two arguments have identical values, whereas `Equal` provides a boolean list indicating where corresponding atoms in its two arguments are identical. Note that `Match` distinguishes integers from floating-point numbers. The above example on the lower left of the four tests whether or not the integer vector `1 2 3` matches the floating-

point vector 1 2 . 0 3; it does not, even though, as the example on the lower right (above) shows, the two vectors are item-by-item equal. Similarly, the different forms of the empty list do not match each other.

Find

Searching and sorting are the bread and butter of many computer applications as well as generally useful tools for programmers. K provides several primitive functions of these kinds, all with highly efficient implementations.

The basic search primitive is the dyadic function denoted by the symbol ? and called Find. Find is based on Match. The left argument can be any list l. The right argument r can be anything, and the result of the function is the smallest index i for which l [i] matches r. For example:

```

    `a `xx `r_q ? `xx
1          `xx matches item 1 of the left argument
    1 -4 3 5 2 3 ? 3
2          3 matches both items 2 and 5 of the left argument;
           the result of Find is the smallest index

```

The items of the left argument do not have to be atoms. The only restriction on the arguments is that the left argument must be a list. For example:

```

    ("abc";"defg";1 2.3 4) ? 1 2.3 4
2          1 2.3 4 matches item 2 of the left argument

```

What if the right argument does not occur among the items of the left? The largest valid index of a left argument l is the count of l minus 1, or (#l) - 1, which is the index of the last item. Consequently, the smallest invalid index is #l, and that value is defined to be the result of Find when the right argument does not occur among the items of the left. For example:

```

    1 3 -4 5 2 3 ? 7
6

```

The right argument 7 does not match any item of the left argument, whose count is 6. A test that the right argument r occurs among the items of the left argument l is that the result l ? r is less than the count of l:

```

    (#l) > l ? r

```

In the sample application, an ATM process will send a customer's credit card number, say `n`, to the server process for validation. The customer's records can be located with `Find`, as in:

```
i: number ? n
```

If `i` is less than `#n` then an OK message should be sent to the ATM process, and otherwise an "Unrecognized Card" message.

We will look at high performance searching later on.

Grade Up and Grade Down

There are two primitive sorting functions, one for sorting in ascending order and the other descending. Both are monadic, and both sort the items of their arguments. The function for sorting in ascending order is called Grade Up and is the monadic function denoted by `<`, while the other one is called Grade Down and is denoted by `>`. For example:

```
<"asedg"          >"asedg"  
0 3 2 4 1        1 4 2 3 0
```

Neither sort function produces the rearrangement of its argument as its result. Instead, each one produces a vector of indices which can be used to rearrange the items:

```
"asedg"[0 3 2 4 1]    "asedg"[1 4 2 3 0]  
"adegs"              "sgeda"
```

The reason for this is that the list being sorted often has related lists that should be reordered to match the order of the sorted list. In the sample application, for instance, suppose that we want to display the database `customers` sorted in the order of decreasing account balances. If we execute:

```
\d .server  
i: > customers.balance
```

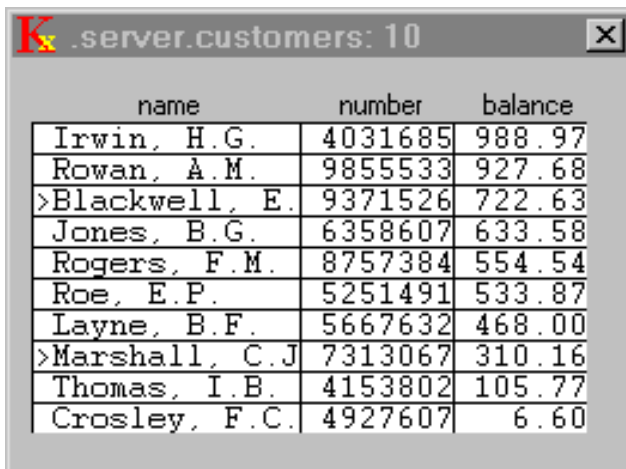
then the items of `customers.balance[i]` are in decreasing order, and the items of `customers.name[i]` and `customers.cc_number[i]` correspond to this order. The values of all entries in `customers` can be reordered at once with `customers[;i]`, and all values can be reassigned with:

```
customers[]: customers[;i]
```

To see the effect — literally — of this expression, first display the table on the screen with:

```
`show $ `customers
```

Now when you execute the above expression you will see the rows in the table change order. (In the next section we'll see how to properly format the `name` column.)



The screenshot shows a terminal window titled ".server.customers: 10". The window displays a table with three columns: "name", "number", and "balance". The rows are sorted by the "balance" column in descending order. The data is as follows:

name	number	balance
Irwin, H.G.	4031685	988.97
Rowan, A.M.	9855533	927.68
>Blackwell, E.	9371526	722.63
Jones, B.G.	6358607	633.58
Rogers, F.M.	8757384	554.54
Roe, E.P.	5251491	533.87
Layne, B.F.	5667632	468.00
>Marshall, C.J	7313067	310.16
Thomas, I.B.	4153802	105.77
Crosley, F.C.	4927607	6.60

Any list can be put in sort order, although it will rarely make sense to do so when the items are very different in type and structure. The most common cases where sorting is useful are integer and floating-point vectors, symbol and character vectors, and string vectors, which are lists of character vectors. We have already seen examples of sorting character vectors and floating-point vectors. For the remainder of this section we will be concerned with symbol and string vectors.

Symbols sort alphabetically, like the entries in dictionaries. For example, the symbol items in the following vector hold the last names of all the customers:

```
a: `Jones `Thomas `Roe `Layne `Irwin `Marshall
a: a, `Blackwell `Rowan `Rogers `Crosley
```

This list is sorted alphabetically as follows:

```
a[< a]
`Blackwell `Crosley `Irwin `Jones `Layne `Marshall `Roe
`Rogers `Rowan `Thomas
```

(Note that lines printed in the session log that are too wide wrap around to the next line.)

Now consider the same information in a string vector:

```
c: ("Jones"; "Thomas"; "Roe"; "Layne"; "Irwin";
    "Marshall"; "Blackwell"; "Rowan";
    "Rogers"; "Crosley")
```

If this listed is sorted in the same way as a we get:

```
c[<c]
("Roe"
 "Irwin"
 "Jones"
 "Layne"
 "Rowan"
 "Rogers"
 "Thomas"
 "Crosley"
 "Marshall"
 "Blackwell")
```

This is not quite alphabetical order, but almost. The result is as if the argument list was first sorted in the order of increasing string lengths, and then alphabetically within each subgroup of strings with the same length. This list would have been sorted alphabetically if all the strings had the same length, which could be arranged by padding the shorter ones with blanks. Or it could be sorted alphabetically if the strings could be converted to symbols. Simple expressions for both solutions will be given in the next section.

Formatting and Unformatting Data

The two functions denoted by the symbol `$` provide the means for converting numeric and symbol data to and from character strings. These functions are used for controlling the appearance of data in reports and on the screen.

The monadic function denoted by `$` is called `Format` and produces a character string representation of any atom. For example:

```
$ 234.37          $ `axyz
"234.37"         "axyz"
$ -23012         $ "a"
"-23012"        , "a"
```

`Format` applies to a list by applying independently to every atom in the list, just like a monadic atomic function such as `Negate`:

```
$ (234.37; `axyz)
("234.37"
 "axyz")
```

In particular, the `Format` of a character string is identical to itself. Technically, `Format` is not an atomic function because it produces character strings, not atoms, when applied to atoms.

More control is provided by the dyadic function denoted by `$`, also called `Format`. The length of the resulting string can be specified with an integer left argument:

```
8 $ (234.37; `axyz; -23012; "abd_3")
(" 234.37"
 "   xyz"
 "-23012"
 "  abd_3")
```

The resulting strings will be left justified if the left argument is negative.

A floating-point left argument can be used to specify the number of decimal digits that should appear in the result as well as the length, as in:

```
8.2 $ 92.71542
" 92.72"
```

Note that we now have a way to sort a list of character strings of different lengths if we know the length of the longest one. In the example from the preceding section, the longest name is Blackwell, with 9 characters. The name of that list is `c`; shorter names are padded on the right with blanks by `-9 $ c`; the sort order of the resulting list is `< -9 $ c`; the list `c` is sorted in alphabetical order by:

```
c[< -9 $ c]
("Blackwell"
 "Crosley"
 "Irwin"
 "Jones"
 "Layne"
 "Marshall"
 "Roe"
 "Rogers"
 "Rowan"
 "Thomas")
```

This solution leaves an interesting question, which we will eventually answer: How is the maximum length of a list of character strings computed?

The dyadic function denoted by `$` can also convert character strings to other specified data types, which in effect unformats data that has been formatted to character strings. In this role the function is called `Form`. The left argument is a prototypical value representing the data type of the result: `0` for integer, `0.0` for floating-point, ``` for symbol, and for completeness, `" "` for character atom or string. For example, the following produce an integer value and floating-point value:

```
0 $ "123"           0.0 $ "123.45"
123                123.45
```

while the next example produces a symbol:

```
` $ "Jones, Ann"
`"Jones, Ann"
```

The last result is something we haven't seen before. This is the general form of entering and displaying symbols. The double quotes around the contents of a symbol can be elided if the contents are a valid name, simple or compound. For instance, ``xyz` and ``"xyz"` have the same meaning and both are valid for entry, but ``xyz` will be displayed in either case:

```

`xyz                `"xyz"
`xyz                `xyz

```

We now have the second solution to the problem from the last section, and this one leaves no unresolved issues. Namely, `` $ c` converts the list of character strings `c` to a list of symbols, and therefore `c[< ` $ c]` sorts `c` alphabetically.

The dyadic function `Form/Format` is like a dyadic atom function, in that a list containing prototypical values and format specifications can appear on the left and a corresponding list on the right, or either the left or right can be an atom.

OPERATORS AND DEFINED FUNCTIONS

Each-Right

There are times when you would like to modify the way a function applies to its arguments without redefining it. One of the best examples is `Find`, which searches for its right argument among the items of its left argument (see `Find` in the previous chapter). In many cases it is not the right argument itself, but every item of the right argument, that you want to find among the items of the left. For example, how would you use `Find` to look up several names in a master list of names?

```
master: ("tom"; "alice"; "mike"; "george"; "mary")
x: ("mary"; "bill"; "mike")
```

To look up every item of `x` in `master`, the expression `master?x` will not do, because it will look up the entire list `x` in `master` (and since `x` is not among the items of `master`, the result will be 5). `K` provides several operators that modify the way functions apply to their arguments, among them one called `Each-Right` that is helpful in this example.

`Each-Right` is denoted by `/:` and like all the operators, applies to the function immediately to its left. For example, `?/:` is the application of `Each-Right` to `Find` that produces a new function called `Find-Each-Right`. This new function searches separately for every item of its right argument — not the entire right argument — among the items of the left argument.

```
master ? x
5                the entire x is not found in master
```

```

master ?/: x
4 5 2

```

x[0] matches item 4 of master,
x[1] is not found in master, and x[2]
matches item 2

In general for a dyadic function f , the function denoted by $f/ :$ and called f -Each-Right is also a dyadic function; it requires a list for its right argument y , its result $x f/ : y$ is a list of the same count as y , and for every index i the following relation holds:

$$(x f/ : y)[i] \text{ matches } f[x;y[i]]$$

That is, the value of the i th item of $x f/ : y$ is identical to f applied to x and the i th item of y . A function produced by applying an operator to another function, as in $f/ :$, is called a *derived function*.

Each-Left

Analogously, there is Each-Left, which is denoted by $\backslash :$ and has the same effect on the left argument of its derived functions as Each-Right has on right arguments. For example, the library function `_in` is a search function like `Find`, but searches for its entire left argument among the items of its right argument and produces 1 if it is found, and 0 otherwise. The operator Each-Left can be applied to `_in`, as in `_in\ :` or In-Each-Left, to search for each item of the left argument among the items of the right. Repeating the above `Find` example for `_in`:

```

x _in master
0

```

the entire x is not found in master

```

x _in\ : master
1 0 1

```

items x[0] and x[2] are each found
in master, but x[1] is not

Each and Monadic Case

Each-Right applies the function it modifies to the left argument paired with every item of the right argument, and Each-Left similar. As you might guess from the title of this section, the third operator in this group, called `Each`, applies the function it

modifies to every item of the left argument paired with every item of the right. Each is denoted by the single quote '. For example, here are lists of various counts and contents created by Take-Each (i.e., #:):

```

3 5 4 #' (1 2; "abcd"; `a `b `c `d `e `f)
(1 2 1                                     this is 3#1 2
"abcd"                                       this is 5#"abcd"
`a `b `c `d)                               this is 4#`a`b`c`d`e`f

```

Note that like a dyadic atomic function, an atom can be used in either argument of a dyadic function derived from Each, as in:

```

5 #' (1 2;"abcd"; `a `b `c `d `e)
(1 2 1 2 1
"abcd"
`a `b `c `d `e)

```

Each-Left and Each-Right only apply to dyadic functions because their definitions require two arguments. The Each operator, however, is different. It applies just as well to monadic functions, where its meaning is to apply the function it modifies to every item in its argument. Before we give examples, there is one thing you must know, which we will come back to later for an explanation. Whenever an operator is applied to the monadic function denoted by a symbol, the symbol must first be modified by a colon. For example, # denotes both monadic Count and dyadic Take. The symbols #' denote Take-Each, not Count-Each. Count-Each is denoted by #:'. For example, if we try to use #' for Count-Each in order to get the count of each item in master, the following occurs:

```

#' master
term: valence error

```

A valence error message appears, meaning that an attempt was made to use a function — in this case #' — with the wrong number of arguments. (See the chapter Programming Environment for the meaning of error messages). The point is that #' is only Take-Each, not both Take-Each and Count-Each. The colon is needed to denote Count-Each:

```

#:' master
3 5 4 6 4

```

Remember the problem of alphabetically sorting the list `c` of last names from the list `customers` in the sample application (see *Grade Up* and *Grade Down* in the last chapter)? One solution required padding the names with blanks in order to make them all the same length, and we used `-9 $ c`, where `$` denotes `Format`. That solution is very specialized, working only for character strings to be padded with blanks. What if for some reason we wanted to pad with asterisks? Or pad a numeric list with 0's? Here is an (almost) general solution. Since 9 is the maximum length of the strings in `c`, the items of `9-#: 'c` give the amount of padding required for each string:

```

n: 9 - #: 'c          n is 9 minus the Count-Each of c
4 3 6 4 4 1 0 4 3 2

```

We will pad with the `*` character for the visual effect. The padding, and then the padded names are:

```

pad: n #' "*"          pad is n Take-Each "*"
("*****";"*****";"*****";"*****";"*****";"*****";"*****";"*****";"*****";"*****")
c , ' pad             Join c and pad item-by-item
("Jones*****"
 "Thomas****"
 "Roe*****"
 "Layne*****"
 "Irwin*****"
 "Marshall*"
 "Blackwell"
 "Rowan*****"
 "Rogers***"
 "Crosley**")

```

There is still the question of finding the maximum value in the list of counts `#: 'c`.

A Note on Syntax

You may have noticed that you are generally free to have extra blanks between function symbols and their arguments or between items of a list, but you cannot have blanks between the symbols for an operator and the function it modifies. For example, `#\:` is correct syntax but `# \:` will result in a parse error.

```

3 5 4 # \: "abc"
      ^
term: parse error

```

note the ^ pointing to the source of the error

Definition of Non-Primitive Functions

K programs are non-primitive functions, i.e. functions that are not denoted by symbols. More often they are called user-defined functions, and even more often *defined functions* for short. A non-primitive function definition consists of a sequence of expressions separated by semicolons and surrounded by braces. The individual expressions in a function definition are executed from left to right, i.e. starting with one nearest the left brace, and the result of the one nearest the right brace is by default also the result of the function. A non-primitive function can have any number of arguments, including zero. The arguments are specified with bracket-semicolon syntax that comes between the left brace at the beginning of the definition and first expression. This syntax mimics the way non-primitive functions are evaluated, with the argument names replaced by corresponding values. For example, the following function definition produces the list of padded names in the example near the end of the last section, when applied to the arguments `c` for `nl`, `9` for `max`, and `"*"` for `fill`:

```
{[nl;max;fill] n:max - #:' nl;pad:n #' fill;nl , ' pad}
```

Function definitions can be broken into more than one line. Each break must occur at a semicolon and can either be at an expression separator like those above or within an expression, as we have seen previously. They must occur at semicolons and the semicolons at the breaks should be elided. For example:

```
{[nl; max; fill] n: max - #:' nl
      pad: n #' fill
      nl , ' pad}
```

is equivalent to the first form. This is convenient for large functions, but for small ones like this one there are other ways to reduce their sizes, and at the same time make them more readable.

First of all, we can cut this definition down by using single letter names, and even further by using `x`, `y`, and `z` as the argument names because the bracket-semicolon argument specification can be elided in the following cases: for functions with no

arguments; for functions with one argument if its name is `x`; for functions with two arguments named `x` and `y`; and for three arguments named `x`, `y` and `z`. This function has three arguments, so we'll use `x` for `nl`, `y` for `max`, and `z` for `fill` (`x` must be the first argument, `y` the second, and `z` the third):

```
{n: y - #: ' x; pad: n #' z; x , ' pad}
```

The use of the names `n` and `pad`, which are local to the function definition, are of little value because they are used only once — their definitions could be used in their place just as well. The definition now becomes:

```
{x , ' (y - #: ' x) #' z}
```

(This is a good example to try reading from the left. Sometimes this process is aided by first replacing the symbols with names, as in `{x Join-Each (y Minus Count-Each x) Take-Each z}`).

Function definitions are ordinary data, the same as string vectors and integer atoms, and do not need to have names. For instance, this definition can be applied with the variables in the previous section as arguments without giving it a name:

```
{x , ' (y - #: ' x) #' z}[c; 9; "*"]
```

(The display of the result, which can be found near the end of the previous section `Each` and `Monadic Case`, has been omitted.)

Of course, if the function is to be used more than once, or if there are readability issues with long definitions, it may be convenient to give it a name and call it with that name, as in:

```
pd: {x , ' (y - #: ' x) #' z}      this is ordinary assignment
pd[c; 9; "*"]                    execute the function
```

(Once again, the display of the result has been omitted.)

Operators apply to function definitions in the same way as to primitive functions, and this provides us with further opportunity to simplify this definition. Namely, instead of applying `Each` to the various primitive functions within the definition, apply `Each` directly to a simpler function definition that pads only a single character string, not every character string in a list. For instance, the function:

```
{x , (y - #x) # z}
```

pads a single character string, as in:

```
{x , (y - #x) # z}[c[3]; 9; "*"]  
"Layne****"
```

and therefore:

```
{x , (y - #x) # z}'
```

pads every character string in a list. That is:

```
{x , (y - #x) # z}'[c; 9; "*"]
```

produces the same list of padded character strings as the previous expressions. One thing we see from this expression is that Each can be applied to functions of any number of arguments, not just one or two. The arguments can be atoms or lists, but if two or more are lists they must have the same length. For example, there are ten names in c, and they could be padded with different characters as follows:

```
{ x , (y - #x) # z}'[c; 9; "*.-_:*. -_:"]  
("Jones****"  
"Thomas..."  
"Roe——"  
"Layne____"  
"Irwin:::."  
"Marshall*"  
"Blackwell"  
"Rowan——"  
"Rogers____"  
"Crosley::")
```

As for giving this function a name, either the new function definition without the application of Each can be given a name and then applied with Each, as in:

```
pd1:{x , (y - #x) # z}  
pd1'[c; 9; "*"]
```

or the derived function can be given a name and applied:

```
pdAlt: {x , (y - #x) # z}'  
pdAlt[c; 9; "*"]
```

In the next section we see how to compute the maximum string length 9 from c.

There is one thing to be careful about with function definitions. If you define a function that spans several lines and is supposed to have a result, be sure that the right brace is on the end of the last line and not on a separate line by itself. Otherwise, the function has a nil result — in effect no result —, which is not we want in this case. For example, if the definition at the beginning of this section had been:

```
{[nl; max; fill] n: max - #' nl
                    pad: n #' fill; nl , ' pad
}
```

where the right brace is now on a separate line, the result would be the empty expression to the left of the closing right brace.

Over and Scan

Consider the following problem: the library function `_dv` is a dyadic function that deletes all occurrences of its right argument from the items of its left argument. For example:

```
1 3 9 -2 3 7 4 -2 3 9 _dv 3
1 9 -2 7 4 -2 9
```

The question is: is there a way to use `_dv` to remove more than one value, say both 3 and -2, in this example? Offhand you might try Each-Right:

```
1 3 9 -2 3 7 4 -2 3 9 _dv/: 3 -2
(1 9 -2 7 4 -2 9
 1 3 9 3 7 4 3 9)
```

The result is two copies of the left argument, one with the value 3 removed, and the other with -2 removed. However, what we want is one copy with both values removed. Each-Right is a parallel operator, in that it applies independently to the items of the right argument paired with the left argument. What we want, in effect, is an iteration operator that accumulates successive applications of `_dv` in one result. There is such an operator, which is called Over and is denoted by `/`. Using Over instead of Each-Right in the above example yields:

```
1 3 9 -2 3 7 4 -2 3 9 _dv/ 3 -2
1 9 7 4 9
```


The way Over works is like this. Think of the left argument as the initial state of the final result. Over first applies `_dv` to this initial state and the first item of its right argument. The result of that evaluation is an intermediate state of the result, which is:

```

1 3 9 -2 3 7 4 -2 3 9 _dv 3
1 9 -2 7 4 -2 9

```

Over then applies `_dv` to this intermediate state and the second item of the right argument, yielding:

```

1 9 -2 7 4 -2 9 _dv -2
1 9 7 4 9

```

In this example the right argument has only two items, so the result of the second evaluation is the result of the expression. If the right argument had more than two items, the result of the second evaluation would be the new intermediate state and `_dv` would be applied to it and the third item, and so on.

There is a companion operator to Over called Scan, which is denoted by `\`. The definition of Scan is essentially the same as Over, except that its results are lists holding all the intermediate states and the result of the corresponding application of Over. Repeating the above example for Scan:

```

1 3 9 -2 3 7 4 -2 3 _dv\ 3 -2
(1 3 9 -2 3 7 4 -2 3 the initial state matches the left argument
 1 9 -2 7 4 -2          3 has been removed from the previous state
 1 9 7 4)                -2 has been removed from the previous state

```

Scan has uses in its own right, but it is also helpful in understanding applications of Over because you see the eventual result as it develops.

Over and Scan control iterative processes, but it is not always necessary to think in terms of the precise way their results are created. For example, the dyadic function `+/`, called Plus-Over, adds each item of its right argument to an accumulating sum that starts with its left argument, as in:

```

0 +/ 1 3 5 7          0 +\ 1 3 5 7
16                    0 1 4 9 16

```

The Over evaluation is actually equivalent to:

```

    (( (0 + 1) + 3) + 5) + 7
16

```

but the way the sum is accumulated doesn't matter; $0 + 1 + 3 + 5 + 7$ would do just as well. Consequently, we simply say that $0 +/ x$ is the sum of the items of x and $0 +\ x$ is a list of partial sums.

Of course the left argument can be any numeric list that conforms with the right argument, not just 0.

As in the case of Each, Each-Left and Each-Right, a function produced by applying either Over or Scan to another function is called a derived function. Over and Scan applied to dyadic functions are special because there are two derived functions, one monadic and one dyadic, much like there are two functions represented by the symbol $-$. So far we have demonstrated only dyadic derived functions. In the monadic case, where there is no left argument, the initial state is then taken to be the first item of the right argument, and the first iterative step applies the function to this initial state and the second item of the right argument. For example:

```

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

```

This is the simplest form of Over and Scan. Summing is a common task, and $+/$ is a common phase in K; over time you may find yourself using this expression without thinking in terms of Over. Here is a little quiz: What well-known function is expressed by $\{ (+/x) \%#x \}$, where the argument is an integer vector, floating-point vector, or other simple numeric list?

The results of any dyadic function derived from Over or Scan can be produced by its companion monadic function. Repeating the first example in this section:

```

    _dv/ (, 1 3 9 -2 3 7 4 -2 3) , 3 -2
1 9 7 4

```

That is, for any dyadic function f , the dyadic expression $x f/y$ is equivalent to the monadic expression $f/(, x) , y$.

Greatest and Least

The applications of Over to Max and Min, the dyadic functions denoted by `|` and `&`, are also among the most common uses of this operator. The result of `|/v` for an integer vector or floating-point vector `v` is the value of the greatest item in `v`, while `&/v` is the least. For example:

```
|/ 1 -2.3 5 10 -5 6      &/ 1 -2.3 5 10 -5 6
10.0                      -5.0
```

It might be helpful to step through these examples. In the case of Greatest, the one on the left, the initial state of the result is 1. Let's refer to the developing result as `r`. The first evaluation is `r|-2.3`, which is 1 and leaves `r` unchanged. Then `r|5` is evaluated and `r` becomes 5; then `r|10` is evaluated and `r` becomes 10; then `r|-5` is evaluated and `r` remains unchanged; and finally the evaluation of `r|6` also leaves `r` unchanged, so the final result is 10 (actually, 10.0 because the argument is a floating-point vector). The result of Least can be analyzed in the same way.

At the end of the section Definition of Non-Primitive Functions, we had completed our solution of padding each character string in a list so that in the result they all had the same length, except for one thing; we had no expression to compute the length of the largest string in the original list. Now we do. It is:

```
|/ #c
9
```

Some and All

Early in the tour (the section Familiar Functions on Somewhat Familiar Symbols) we noted that when Max is restricted to boolean arguments it is equivalent to Logical Or, and Min is equivalent to Logical And when similarly restricted. What about `|/` and `&/` when restricted to boolean arguments? Under this restriction `|/v` is 1 if there is at least one 1 in `v` and `&/v` is 0 if there is at least one 0 in `v` (just think in terms of Max and Min, and the fact that the largest and smallest boolean values are 1 and 0).

A list of boolean values often represents the result of testing a list of conditions for True and False. For this representation, to say `|/v` is 1 — and therefore at least one item in `v` is 1 — now means that some of the conditions are true. Similarly, to say

$\&/\vee$ is 0 means some of the conditions are false, or equivalently $\&/\vee$ is 1 only if all the conditions are true. Consequently $|/$ is called Some when applied to boolean lists and $\&/$ is called All.

At the end of the section Atom, Count and Shape in the chapter More Primitive Functions, we used the following expression to test whether or not any item of `1 -12 5 6` is greater than 10:

```
0 < # & 10 < 1 -12 5 6
0
```

The test goes like this; by applying Where (monadic $\&$) to the expression `10 < 1 -12 5 6` we get an integer list holding the index of every item in the expression with value 1, or true. Consequently, if the count of this integer list is greater than 0 it must contain at least one item, and therefore at least one item of `1 -12 5 6` must be greater than 10. A more efficient and straightforward test is simply whether some of `10 < 1 -12 5 6` are true:

```
|/ 10 < 1 -12 5 6
0
```

K ON UNIX

A K process is started in Unix with a K command, which in its simplest form is:

```
k
```

The command can also specify a script to be loaded, which will be done immediately after K is initialized and is the way applications are started. All scripts have the extension “.k”, but the extension need not be included in the command. For example, if the name of the script is `prices.k` then the command

```
k prices
```

will cause a K process to start up and immediately load the script `prices.k`.

If interprocess communications are to be used then it may be necessary to specify a communication port number in the startup command by which other K processes identify this one. This number is a four-digit integer with values greater than 1000, although your Unix system may hold some numbers in this range reserved. The port number parameter is `-i`. For example:

```
k prices -i 1234
```

will give the `prices` application the communication port number 1234. Port numbers need only be specified for server processes. If the `-i` option is used, it must be after the load script, if the latter is present. Command line parameters occurring after the port number and load script are optional, and are accessible from within the K session as a string list, via `_i`.

Normally a K session permits keyboard input, but some applications, in particular end-user applications, need to disallow it. For that purpose the K command that started the application should be followed by `</dev/null` to its right, as in:

```
k prices -i 1234 </dev/null
```

This K process cannot receive keyboard input, but any screen objects the application creates will be active, as well as interprocess communications.

The fonts used for screen displays are specified in Unix by the environment variables KFONT and KLFONT. The font named in KFONT is for data and should be monospaced; the one named in KLFONT is for labels and its point size should not exceed that of the font used for data. The default values are:

```
KFONT                *courier-medium-r*-14*
KLFONT                *helvetica-bold-r*-12*
```

Any fonts named in the output of the Unix command `xlsfonts` can be used. For application portability it is best to specify only the font name and size, and use the wild card `*` where minor variations in the fonts are acceptable, as in the above default specifications. See the X Windows documentation for a discussion of the font description format.

It is possible to adjust the colors used by the display hardware to show K data objects, using the KCOLOR environment variable. The default is black, white, and a light gray:

```
KCOLOR                0 -1 808080
```

These values refer to the foreground, background, and panel midground. See the K Reference Manual for a description of the color codes.

K manages its own storage for data objects. This backing store is comprised of files created and maintained by K, which are located in the directory designated by the environment variable KSWAP. If this variable is not set, then `/var/ktmp` is used. For performance reasons the designated directory should be on the local disk of the workstation, and there must be sufficient storage available in the directory, i.e. at least 500 megabytes. If your installation provides a more appropriate directory with these characteristics, set the environment variable KSWAP to the name of that directory before entering K.

K ON WINDOWS

The Windows version of K is designed to run under Microsoft Windows NT 4.0 and most features are also functional in Windows 95. In both cases, the working environment is largely similar to that of K on Unix. Some K characteristics have been augmented and others added in order to take advantage of features available in Windows.

There are actually two K programs (`.exe` files), both of which use the same K dynamic load library (`.dll` file). The first, named simply `k`, is for developers, while the second, `kr` or “K runtime”, is intended for end application users. This latter version does not provide the K console familiar to developers. The current discussion focuses on the main console program.

Running K

K may be executed in several different ways. The traditional method involves typing in a command shell’s console:

```
k
```

The command shell would probably be *command* (more common on Windows 95) or *cmd* (a more powerful shell, usually available only in Windows NT). Certain versions of the emacs editor are also supported. In this scenario, K takes over the console until the session is terminated. Alternatively, the user can simply click on an icon representing the K program. In this case, a brand new console is created, which lasts until the end of the K session.

K is also started up when an icon is dragged and dropped onto the K icon. In this case, the file represented by the first icon is taken to be the first argument to a K session (which would imply that the file is a K script; see below). Finally, a K session is started if the OS registry has been configured to execute K when K scripts are opened. End users would have kr set up to run this way.

One or more arguments may be provided on the command line, or installed as implicit arguments via the system registry (e.g., for use in opening K scripts). The first argument to a K session is the name of a file to be loaded as a K script (`.k` suffix). One may also use the `-i` option to specify a port number, when the K session is intended to act as a server process, or the `-h` option for an http port. Following arguments may be any number of valid strings, which are accessed within the K session via the `_i` pseudo-function. For example, the following command:

```
k prices -i 408 34 67 date "15 July 1997"
```

results in the script `prices.k` being loaded, the session becomes a server process available at TCP/IP port 408, and `_i` has the value of the string list (`"34";"67";"date";"15 July 1997"`). Loading multiple scripts must be done by including `\l` commands in one main script.

One cannot redirect stdin in K on Windows; run-time license operability is provided by the companion program, kr. However, both stdout and stderr are redirectable where the command shell permits this. For example:

```
k -i 408 2> logfile.txt
```

will cause all stderr output to be redirected to the file `logfile.txt`.

Environment and Nonstandard Commands

The k user can use several commands which duplicate the functionality of command line options and environment variables. The nonstandard options are all available via the command `\m`. They are:

- (0) `\m` - list available nonstandard commands
- (1) `\m i` - set the server port number within a K session (e.g. `\m i 1234`)
- (2) `\m h` - set the http port number
- (3) `\m c` - set the KCOLOR variable (e.g. `\m c 0 -1 009999`)

(4) `\m f` - set the `KFONT` variable (e.g. `\m f Courier New-8`)

(5) `\m l` - set the `KLFONT` variable (e.g. `\m l Helvetica-7`)

The latter three `\m` commands that affect K environment variables `KCOLOR`, `KFONT` and `KLFONT` are relevant to K visual displays, and are only effective if executed before the first instantiation (`\show$`) of a K data object. These variables can also be set outside the K session in the standard way:

```
> set KCOLOR="1 -1 656565"  
> set KFONT="Courier-9"  
> set KLFONT="Arial-bold-8"
```

The double-quote characters are optional in most command shells. Within a K session, they should be avoided.

The specification of fonts is similar to the method used in X Windows, in order to increase portability of k scripts. (However, note that the font family names under Windows NT are likely to be different.) In a specification such as those in the examples above, one *must* provide the font family name (e.g., Arial, or Courier New). Other characteristics are delimited by the `-` and `*` characters, and may include pointsize (an integer), an italics flag (`i` or `italic`), and a density (one of thin, extralight, light, normal, medium, semibold, bold, extrabold, or heavy). The default fonts are the fixed width and proportional width defaults provided by the Windows OS.

Colors used in the K graphical user interface are specified in the same way as on Unix, using `KCOLOR` (see K on Unix). Note that if a Windows display screen is limited to 256 colors, only the 20 “system” colors are available to the K GUI. If more flexibility is required, the display settings should be adjusted so that 16 bits or more are available to each screen pixel.

Error Behavior

In the K working environment, errors may arise from console commands, from interprocess communication, or from K graphical objects. They are reported to the user via the graphical interface or the console, depending on various conditions. Entry into data widgets of badly typed data will result in that k widget assuming the title “Error”. This type of error may be cleared by pressing the escape key.

If the Error Flag $\backslash e$ is set to 1 in a session, all other types of errors are reported to the console. If $\backslash e$ is 0, the errors are reported to the console if they originate at the console and there are no K widgets associated with the session. Otherwise, errors are reported via the k “Error” widget in the GUI; see below.

When $\backslash e$ is set to 1 and an error occurs, user input is required while the session is in suspended mode. $\backslash e$ is 0 by default in kr.

When the K error widget comes up in the screen display (because an error has occurred and $\backslash e$ is 0), it displays an error message. Until the user deactivates the error window by clicking with the mouse, all K widgets belonging to that particular K session are suspended. Input and output to the console are also suspended, though the console can still be manipulated (moved, iconified and so on) in the desktop.

INDEX

Symbols

! monad 57
dyad 56
monad 53
\$ dyad 62, 63
\$ monad 62
% 19
& dyad 19
& monad 25, 55
&/ 76
' 67
* dyad 12
+ dyad 12
+/ 73
, 20
- dyad 12
- monad 13
-h option 80
-i option 77, 80
.. 44
.a attribute 45
.c attribute 45
.k file 33, 77, 80
.l attribute 45
/ 72
/: 65
: 22
< dyad 12
< monad 59
= dyad 12

> dyad 12
> monad 59
? dyad 58
?/: 65
@ monad 53
[: 42
\ command 9
\ operator 73
\: 66
\| 5, 12
\d 35
\e 7, 82
\l 33, 37, 80
\m command 80
\v 36
^ dyad 19
^ monad 53
_ 20
_dv 72
_i variable 77, 80
_in 66
`button class 46
`data class 46
`form class 46
`hide 33
`show 32
| 19
|/ 76
~ dyad 57
~ monad 13
1: dyad 32

1: monad 33

A

abort command 9
absolute name 35, 38
addition 12
All 76
ancestor 35
annotation 6
APL 12
append 20
application 5
 end-user 78
 sample 29
argument name 69
arithmetic 12
arrangement 46
Assignment 22
assignment
 directory 42
ATM example 30
Atom 53
atom 6, 13
atomic function 17
attribute 44. *See also* .a, .bg
 etc.
 inheritance 45
 reserved 44
attribute dictionary 43

B

back-quote
 denoting symbol 13
blank. *See also* space
 where permitted 68
boolean 12, 19, 57, 66, 75
 list 25, 55
brace character 69
bracket form 22, 69
button class 46

C

C language 11
caret
 indicating error 8
change directory command 36
character list 7
clear
 suspension 8
colon 22
 symbol modification 67
color 78, 81
command 5, 29
 startup 34
command line 80
communication
 interprocess 34
communication port 34, 77
compound name 34
conditional evaluation 11
conform 17
 in indexing 24
console 34, 79, 81
constant 22
control statement 11
corresponding items 18
Count 53
cross-sectional indexing 39
current directory 35
cursor 5, 7
cyclic 56

D

data class 46
data font 78
data object 32, 34, 78
data type 13
 dictionary 40
 error 7
database 33
debug 7
default argument names 69
default color 78
default directory 35
default display class 46
default font 78, 81
defined function 69
derived function 66
descendent 35
dictionary 40
directory 29
 current 35
 default 35
 entry 35
 index 40
 root 34
 top-level 30
 working 35
display
 class 44
 of lists and vectors 17
display directory command 35
Divide 19
Do 11
dot
 attribute directory 44
 make dictionary 43
double-quote 81
dyadic function 13, 74
dynamic load library 79

E

Each 66, 71
Each-Left 66
Each-Right 65

emacs 79
empty expression 72
end-user application 78, 79
Enlist 20
entry 35
Enumerate 57
environment 5
environment variable 78
Equal 12
error 7, 81
Error Flag 8, 82
escape key 81
evaluation 5
evaluation order 26
execution 69
 of an expression 7
 resume 8
exit 5
exponential format 13
exponentiation 19
expression 5, 6
 incomplete 8
 mathematical 26
extension 77

F

false 12
file 78
 .exe and .dll 79
file extension 33, 77
file I/O 11
file read 34
file write 32
Find 58, 65
Find-Each-Right 65
floating-point
 number 13
 vector 31
Floor 20
font 6, 78, 81
Form 63
form class 46
Format 62
function

derived 66
non-primitive 69

G

global variable 30
Grade Down 59
Grade Up 59
graphical user interface. *See*
 GUI
greatest 75
GUI 11, 81, 82

H

hide 33
hierarchy 34
homogeneous list 15
horizontal display 49
http 80

I

icon 79
If 11
immediate left rule 13, 22
In-Each-Left 66
incomplete expression 8
indent 5
index 22
 assignment 24, 40
 blank/empty 39
 cross-sectional 39
 directory 40
index error 8
integer list 7
interactive 5
interprocess communication
 11, 34, 77
item-matching rule 17
items at depth 38
iteration 73

J

Join 20

K

K console 79
K language 5, 11
K name space 34
K process 35, 77
K program 5
K Reference Manual 8, 78
K runtime 79
K script 80
K session 35
K-tree 34
KCOLOR 78, 81
KFONT 78, 81
KLFONT 78, 81
KSWAP 78

L

label attribute 48
language
 K 11
layout 46
Least 75
least common multiple 18
left justification 62
left-to-right 25, 69
length 53
Less 12
list 6, 11, 13
 constructor 20, 56
 entries of directory 36
 floating-point 14
 index 25
 length/count 53
 numeric 14, 17
 one-item 21
 simple 16
load 5, 33, 37, 77
Logical And 19
logical negation 13
Logical Or 19
long right scope 27

M

Match 57
matrix 54
Max 19, 75
Microsoft Windows 79
Min 19, 75
Minus 12
monad
 enforce with colon 67
monadic function 13, 74
monospace font 78
More 12
multiplication 12

N

name space 34
Negate 13
nil 15, 22, 72
non-primitive function 69
nonstandard command 80
Not 13
NT 79
number 6
numeric list 7, 14, 17
numeric vector 17

O

object edit 32
one-item list 14, 21
operator. *See* adverb
operator precedence 26
order
 in indexing 25
 of evaluation 26
Over 72

P

pairing 17
parent directory 35
parentheses 26
 closing 9
parse error 68
path 35

Plus 12
Plus-Over 73
port 34, 77, 80
Power 19
precedence 26
primitive function 11
programming environment 5
prompt 8

Q

quote 67

R

read file 34
Reciprocal 19
rectilinearity 53
redirection 80
Reference Manual. *See* K
 Reference Manual
registry 80
relational function 12
relative name 35
replication 56
result
 function 69
right-to-left evaluation 26
root directory 34
rounding 20

S

scalability 29
Scan 73
scope
 long right 27
screen 81
screen display 32, 78
script 5, 77, 80
script file 33
search 58
selection-at-depth 39
selection-by-index 22
self-describing data objects 53
server 34, 77, 80

Shape 53
shell 34, 79
show 32, 81
simple list 16
simple name 34
Some 76
sort 58
 of strings 61
 of symbols 60
sorting function 59
space 7, 14
 where permitted 68
spreadsheet 42
startup 77
startup command 34
state
 computation 7
statement 5
stdout and stderr 80
storage 78
string 16
subdirectory 31
subtraction 12
suspension 7, 82
symbol 11, 13
 as index 40
 general form 64
syntax 68

T

Take 56
TCP/IP 80
terminology 15
text file 5
Times 12
toolkit 5
top-level directory 30
top-level object 35
top-to-bottom execution 33
transaction 30
tree 34
true 12
type error 8

U

Unix 77
user-defined function 69

V

valence 67
variable 22
vector 15
 numeric 17
 of indices 59
vertical display 46
visual display 81
visual object 32

W

Where 25, 55
While 11
wild card 78
Windows 79
working directory 30, 35, 45
write to file 32

X

xlsfonts 78