

# ngn/k tutorial

razetime

## Contents

<b>Preface</b>	<b>4</b>
<b>1 Introduction</b>	<b>4</b>
1.0.1 <code>,x</code> Enlist . . . . .	7
1.1 Vocabulary from this lesson . . . . .	9
1.2 Exercises . . . . .	9
<b>2 Working with Arrays</b>	<b>9</b>
2.0.1 <code>!x</code> Range . . . . .	9
2.0.2 <code>!x</code> Odometer . . . . .	10
2.0.3 <code>#x</code> Length . . . . .	10
2.1 Exercises . . . . .	12
<b>3 Functions and Defining Variables</b>	<b>12</b>
3.0.1 <code>x!y</code> Create Dict . . . . .	13
3.0.2 <code>\$[;;...]</code> If . . . . .	14
3.1 Scoping . . . . .	14
3.2 Trains . . . . .	15
3.3 Vocabulary from this lesson . . . . .	15
3.4 Exercises . . . . .	16
<b>4 Indexing and Application</b>	<b>16</b>
4.0.1 <code>f@x</code> At . . . . .	16
4.0.2 M-Expressions . . . . .	18
4.0.3 <code>f.x</code> Dot . . . . .	18
4.1 Projections . . . . .	19
4.2 Vocabulary from this lesson . . . . .	20
4.3 Exercises . . . . .	20
<b>5 Adverbs</b>	<b>21</b>
5.0.1 <code>f' x</code> Each . . . . .	21
5.0.2 <code>  x</code> Reverse . . . . .	21
5.0.3 <code>x f' y</code> Each Dyad . . . . .	22
5.0.4 <code>x f\:</code> y Each Left . . . . .	23

5.1	Folds and Scans . . . . .	23
5.1.1	<code>x f/ y</code> Over (Fold from left) . . . . .	23
5.1.2	<code>x f\ y</code> Scan (Scan from left) . . . . .	24
5.2	Iterated Fold and Scan . . . . .	25
5.2.1	<code>f\[x;...y]</code> Iterated Scan . . . . .	25
5.3	Vocabulary from this lesson . . . . .	26
5.4	Exercises . . . . .	26
<b>6</b>	<b>Numbers and Logic</b>	<b>26</b>
6.0.1	<code>x?y</code> Find . . . . .	28
6.1	Vocabulary from this lesson . . . . .	29
6.2	Exercises . . . . .	29
<b>7</b>	<b>More Adverbs</b>	<b>29</b>
7.0.1	<code>x f/ y</code> For . . . . .	29
7.0.2	<code>x f\ y</code> For (Scan) . . . . .	30
7.0.3	<code>f g/ z</code> While . . . . .	30
7.1	Other looping adverbs . . . . .	30
7.1.1	<code>g/ z</code> Converge . . . . .	30
7.1.2	<code>x _ y</code> Drop . . . . .	31
7.1.3	<code>x f': y</code> Eachprior . . . . .	31
7.1.4	<code>x f': y</code> Stencil . . . . .	32
7.1.5	<code>x ': y</code> Window . . . . .	32
7.2	More adverb overloads . . . . .	32
7.3	Not Really Adverbs . . . . .	32
7.3.1	<code>&amp; x</code> Where . . . . .	33
7.4	Vocabulary from this chapter . . . . .	33
7.5	Exercises . . . . .	34
<b>8</b>	<b>Dictionaries, Tables, Strings and other Miscellaneous functions</b>	<b>34</b>
8.1	Dictionary Trivia . . . . .	34
8.1.1	Tables . . . . .	35
8.2	String functions . . . . .	35
8.2.1	The Rest . . . . .	36
8.3	Exercises . . . . .	38
<b>9</b>	<b>Special Forms</b>	<b>38</b>
9.1	Amend (@) . . . . .	38
9.1.1	3 Arguments (@[x;y;z]) . . . . .	38
9.1.2	4 Arguments (@[x;y;z;u]) . . . . .	38
9.2	Drill / Deep Amend (.) . . . . .	38
9.2.1	3 Arguments (.[x;y;z]) . . . . .	38
9.2.2	4 Arguments (.[x;y;z;u]) . . . . .	38
9.3	Try (.[x;y;z]) . . . . .	39
9.4	Splice (?[x;y;z]) . . . . .	39

<b>10 Input and Output</b>	<b>39</b>
10.1 Command line arguments and other console values . . . . .	40
10.2 Vocabulary from this chapter . . . . .	41
10.3 Exercises . . . . .	41
<b>11 Debugging and Errors</b>	<b>41</b>
11.0.1 'parse . . . . .	42
11.0.2 'rank . . . . .	42
11.0.3 'type . . . . .	42
11.0.4 'domain . . . . .	43
11.0.5 'index . . . . .	43
11.0.6 'length . . . . .	43
11.0.7 'value . . . . .	43
11.0.8 'nyi . . . . .	43
11.0.9 'stack . . . . .	43
11.0.10 'io . . . . .	44
11.0.11 'compile . . . . .	44
11.0.12 'limit . . . . .	44
11.0.13 'eoleof . . . . .	44
11.0.14 'oom . . . . .	45
11.1 Finding and throwing errors . . . . .	45
11.2 Inspecting functions internal bytecode . . . . .	45
11.3 Vocabulary from this chapter . . . . .	45
11.4 Exercises . . . . .	45
<b>12 Thinking in an array language</b>	<b>46</b>
<b>13 Writing a Prelude</b>	<b>49</b>
13.1 Math . . . . .	50
13.2 Strings . . . . .	51
13.3 Array primitives . . . . .	52
13.4 Exercises . . . . .	52
<b>14 A Sudoku program</b>	<b>53</b>
14.1 Generating a random board . . . . .	53
14.2 Displaying a Sudoku puzzle . . . . .	55
14.3 Solving a Sudoku puzzle . . . . .	55
14.3.1 Finding valid moves . . . . .	56
14.3.2 Validating a board . . . . .	56
14.3.3 Step 1: Filling in the right places . . . . .	57
14.3.4 Step 2: Filling all possible candidates . . . . .	58
14.3.5 Combining all the steps . . . . .	58
14.4 Creating a valid puzzle . . . . .	58
14.5 Playing the game . . . . .	59
14.6 Exercises . . . . .	59

<b>15 Tables and Libraries</b>	<b>60</b>
15.1 Tables . . . . .	60
15.2 File Model . . . . .	61
15.2.1 <x Open . . . . .	61
15.2.2 >x Close . . . . .	61
15.3 Libraries and Namespaces . . . . .	62
15.3.1 .x Get . . . . .	63
15.4 Serialization . . . . .	63
<b>16 The ngn/k C API</b>	<b>63</b>
16.1 Extending ngn/k with the C Foreign Function Interface . . . . .	64
16.2 Building <code>libk.so</code> . . . . .	64
16.3 C API Datatypes . . . . .	64
16.4 Using functions from <code>libgmp</code> . . . . .	65

## Preface

You may access this book in the following formats:

- Website
- PDF
- EPUB

This book would not have been possible without the help of ngn's dedicated work on maintaining and teaching ngn/k, alongside many of the people in the community who have contributed to the process of documenting and improving K. I would like to thank all the people involved directly and indirectly with this project.

## 1 Introduction

Welcome to K!

K is an array-oriented programming language. Although this term may sound new, K comes from a family of programming languages which date back to APL in the 1960s. You may already know a popular object-oriented language such as Python or Java, so why should you learn K?

Other languages may have array programming capabilities, such as Julia or Python+NumPy, but there are few things which compare to the joy of thinking and using K fluently. K and the APL family of languages push the horizons of what a programming language should look like, and how it should be used. Using a tool that is designed for its job has its own profound benefits, and K proves that rule very well. If you are looking for a new way to look at programming and you want to have fun, you are in the right place.

Part of K's design was to improve upon some of APL's issues. At a glance, the most noticeable improvements are:

- You can type K on an ordinary QWERTY keyboard with no extra software.
- K emphasizes general purpose programming more than mathematics, which simplifies the steep learning curve of APL's more math oriented sides.
- K adds and removes features on every iteration of its design. This lets it refine its design better than languages which are forced to maintain backward compatibility, like APL, or even Python.

Each one of K's major iterations is called a *dialect*. In this tutorial, we will learn about ngn/k, an open source implementation of the K6 dialect. Learning ngn/k will give you the fundamentals required to understand any K dialect.

K takes some time and effort to learn, just like any language. However, K is designed to be simple. Most of K's basic concepts can be finished in an afternoon. Some parts of K will be hard to digest coming from a different language background, but with a little focused practice and this beautiful tutorial, I can assure you that you will be able to do great things with K.

To start with K, first understand how it uses "right to left" evaluation instead of operator precedence rules. Take this simple math expression:

`2 * 2 + 3`

You might think to do `2 * 2` first, then add 3, which equals 7. However, K does not care about primary school rules like PEMDAS. K evaluates from right to left:

`2 * 2 + 3 = 2 * (2 + 3) = 10.`

If you want the result to be 7, then you can add parentheses:

`(2 * 2) + 3`

...or you can reorder the operations so the multiplication is performed first:

`3 + 2 * 2`

The easiest way to run ngn/k is to go to the online interpreter, available at <https://ngn.bitbucket.io/k/>.

I recommend using the online interpreter's REPL available at <https://ngn.bitbucket.io/k/#r> until you get comfortable with K. If you like running things locally, then feel free to build the interpreter from source: <https://codeberg.org/ngn/k/>.

Let's start with a Hello World:

`"Hello, World!"`

K will automatically display the result of every line in your program as K data. This means that you can safely paste the result of a K expression into your program, and use it later.

To prevent K from repeating everything you say to it, you can add a semicolon to the end of your line. For example,

```
"Hello, World!";
```

will do absolutely nothing!

To actually display a string without K's formatting, you can do this:

```
/ an actual hello world program
` 0:"Hello, World!"
```

In this program, `` 0:` is the equivalent of a print statement in any other language. It will print a simple character array given to it with a newline at the end. You can change it to `` 1:` if you want to display without a newline.

The line starting with a `/` is a comment. You can also use comments inline.

Reminder: K's "operators" have no precedence. So the interpreter reads "Hello World" first, then `0:`, then the backtick.

The other major part of our Hello World program is the character array, which brings us to the lego bricks of the array-based language: the arrays.

The things that you can put in an array are called **Nouns**. These can be:

- Numbers, which can be simple integers or floating point numbers.
- Characters, which are enclosed in double quotes
- Symbols, which are a series of alphanumeric or `.` characters preceded by a backtick, like `` .symbol123`.
  - Symbol names cannot start with a number.
  - Symbols can be made from arbitrary strings using double quotes:  
`` "symbol with spaces"`
- Other Arrays
- Functions and dictionaries, which we will discuss later.

**Verbs** are primitive functions, written with a single character. When you put a verb inside an array, K converts it to a noun. A verb can take one or two arguments, and can have multiple meanings based on what data the verb is given. Verbs are equivalent to operators in other languages.

**Adverbs** are primitive higher-order functions, written with one or two characters. An adverb takes one or two verbs on the left and produces a new verb with modified functionality. Some adverbs also behave like verbs if they are not given a verb argument. Adverbs bind first, and they are always evaluated first. Before verbs are executed from right to left, adverbs must be paired with their respective arguments.

The default way of defining multi-element arrays is as follows:

```
(noun1; noun2; noun3)
```

In scripts, you can also use a newline and a space instead of semicolons to separate array elements, so:

```
(noun1
 noun2
 noun3)
```

is also valid array syntax.

A single element array of any type can be defined with the verb **enlist**, which is `,.`

```
,1
,"a"
,`symbol
```

It's important to understand that just `"a"` is a character, not a string. `, "a"`, however, is an array containing the character `"a"`. Now to officially introduce you to your first K verb: `enlist`.

### 1.0.1 `,x` Enlist

**Symbol:** `,`

**Args:** `,` any\_value

**Description:** Create an array containing `x`.

Strings/character arrays can be made by putting multiple characters in double quotes, as you saw in the Hello World example. A single character string cannot be made with double quotes, since a single character in quotes is considered a character, as shown earlier.

```
"Hello, world!"
```

```
"This is a K string"
```

You can escape a double quote inside a character array with a backslash. K supports C style escape sequences in its strings, and you can refer to this table for the basic escapes:

Escape	Character
<code>\\</code>	<code>\</code>
<code>\n</code>	newline
<code>\"</code>	<code>"</code>
<code>\t</code>	tab

Arrays of symbols are created by juxtaposing the symbols together, with no spaces between them.

```
`a`b`c  
`symbol`list
```

A list of numbers separated by spaces is a numeric array. All of the following are numeric arrays:

```
45 56 67  
420 69  
1 2 3
```

Numbers (and **only** numbers) separated by spaces will be treated as numeric arrays. This behaviour in K is called *stranding*.

One special thing about K's arrays is that some operations vectorize by default. Consider this:

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

gives 5 7 9. This is also called a *conforming* operation. Conforming "penetrates" to the deepest subarrays in an array(atoms), so these:

```
1 + 2 3 4 -> 3 4 5  
1 2 3 4 + (1;2 3;4;(5;6 7)) -> (2;4 5;7;(9;10 11))
```

are also valid K expressions. All basic arithmetic operators support conforming operations:

Verb	What it does
+	Add
-	Subtract
*	Multiply
%	Divide

Some primitives in K conform only to one side. The ones that conform to the left are called left atomic, because they penetrate to the atoms of the left argument. The same goes for right atomic primitives.

Every verb in K can take 1 or two arguments. K programmers often call the 2-argument verbs *dyadic* and 1-argument verbs *monadic*. For example, these same tokens do very different things when they are given one argument on their right side:

Verb	What it does
+	Flip (Transpose)
-	Negate
*	First element of an array
%	Square root



You can append a colon `:` after a verb to make sure that it executes with only one argument (monadic).

K has 22 primitive verbs and 6 primitive adverbs. They do different things depending on both the **type** and **number** of arguments given, leading to more than 70 unique functions the K programmer can use. Every primitive has an important usecase, and remembering these can be a challenge when you're starting out.

The overloads of each primitive verb are usually inter-related and are easy to understand. Once you get the hang of them, you will have the tools to create any program you like. We will cover the functionality of K's primitives, their overloads, and their basic usecases in this tutorial.

## 1.1 Vocabulary from this lesson

- Noun: Any defined value in K.
- Verb: A primitive function that can be used infix, and operates on nouns.
- Adverb: A primitive higher order function that creates a new function from a given verb, or function noun. Adverbs are part of K's syntax itself.
- Atom: Any value that is not an array (function, symbol, character, integer, float).
- Monad/Monadic Function: Single argument function.
- Dyad/Dyadic Function: Two argument function.
- Conforming Operation: An operation that works on the lowest level of nesting it can work with. Most of K's verbs perform conforming operations.

## 1.2 Exercises

1. Guess the result of `1 2 3 + 4`, Try it in the REPL. Then try `1 2 3 + 4 5`. What does K say, and why?
2. `ngn/k` has inbuilt documentation, available with `\` in the REPL. Try leafing through each help topic, and don't worry if you don't understand some things. The inbuilt documentation is the most complete and up-to-date cheatsheet on `ngn/k`, which will be very useful in the coming chapters.
3. Define an array with one integer, one character and one symbol.

# 2 Working with Arrays

You know how to make arrays by hand. Now, it is time let the program make them for you.

The most basic and universal array creation verb is *range* (`!`):

### 2.0.1 `!x` Range

**Symbol:** `!`

**Args:** ! number

**Description:** Array with numbers from 0 to  $x-1$ .

Providing range with a single number gives an expected result:

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

But what does that second result mean? ! generalizes to arrays of numbers, and this is called odometer. What this does is generate all indices to index into an array that has the dimensions described in  $x$ . Each column denotes one element.

### 2.0.2 !x Odometer

**Symbol:** !

**Args:** ! numeric\_array

**Description:** Generate an array where each column denotes an index into an array which has dimensions  $x$ , in row-major order.

Odometer always generates an array of the length it is given, or in K's notation,  $\#x$ .

### 2.0.3 #x Length

**Symbol:** #

**Args:** # array/atom

**Description:** Length of the given array. Returns 1 for atoms.

# when called dyadically is one of the most important verbs in K, called *reshape*. Reshape lets you construct arrays with any number of dimensions:

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

When there are not enough elements to fill all dimensions, `#` will repeat the elements of `y` till it is filled.

You can also omit dimensions in reshape:

```
3 0N#!9
(0 1 2
 3 4 5
 6 7 8)
```

Here, `0N` (integer null) denotes the missing dimension. `K` will try to fit all elements of the array in the smallest shape possible using the rest of the dimensions. You can only omit one dimension in reshape.

Arrays are very flexible, first class values in `K`, and the more you buy into that concept, the more fun you will have.

- Concatenation (`x,y`) joins two arrays together, or appends an atomic value to a array.
- Take (`x#y`) takes elements from the beginning or the end of a array
- Drop (`x_y`) drops elements from the beginning or the end of a array
- Cut (`x_y`) cuts (partitions) an array at the indices given.
- Delete (`x_y`) removes an element at a given index.
- Except (`x^y`) removes the elements of one array from another.
- Flip (`+x`) transposes a rectangular array.
- Distinct (`?x`) gets the unique elements of an array.

*Take* is a special case of Reshape where `x` is an atom, and `x` is lesser than the length of `y` (`#y`). It truncates the given array to length `x`. Negative `x` takes from the end.

*Drop* (`atom _ array`) does the same thing as take, except it removes `x` elements from the beginning or end.

*Cut* (`array _ array`) has some special behaviour: it ignores all elements till the first index in `x`, and then performs the cutting operation:

```
0 4_"feedface"
("feed";"face")

3 5_"adfdsfsdfa"
("ds"
 "fsdfa")
```

*Delete* is `array _ index` and it is quite self explanatory.

*Except* (`array ^ array`) works like multiset difference: it removes the elements of `y` from `x`.

*Flip* (`+ array`) will error on non-rectangular arrays. However, a simple list consisting only of atoms will be enlisted.

```
+!4
,0 1 2 3
```

## 2.1 Exercises

1. Convert 23 34 45 56 to (,45; ,56) using a single primitive.
2. Take the first five elements of !9, multiply them by 2, and put them back in the same place in !9.
3. Create the array (6 8 10; 12 14 16) using the primitives from this chapter. Try to find more than one way to do it.

## 3 Functions and Defining Variables

Now that you have an endless supply of array creation power, you now have to find a way to reuse those neat little arrays.

To define a variable, you type its name followed by a colon:

```
one: 1
```

Assigns the number 1 to the variable `one`. You can then use `one` anywhere you like. Yay!

To define multiple variables, you can put the variable names and their values in arrays:

```
(one; two; three): 1 2 3
1 2 3
one
```

If you have already defined a variable, you can then modify its value in the same way with the help of reassignment:

```
one: 2
one
2
```

Or modified assignment:

```
one += 3
one
5
```

Modified assignment happens when a verb is given before the `:` symbol. The above statement is the same as `one: one + 3`.

You can assign any noun to a variable. For example, a dictionary can be named with:

```
dict: `a`b`c ! 1 2 3
```

Creates a dictionary `dict` with key-value pairs ``a:1; `b:2; `c:3`

### 3.0.1 x!y Create Dict

**Symbol:** !

**Args:** array ! array

**Description:** Create a dict with keys `x` and values `y`. `x` and `y` must be of equal length.

You can get values from a dict using square brackets: `dict[`a]` will give you 1.

You can also assign functions to variables. The default method of defining functions in K is to create a lambda. For example:

```
plus:{x+y}
plus[1;2]
3
```

creates a function `plus` and passes the arguments 1 and 2 to it.

Function call syntax applies to Verbs as well, so `+ [1;2]` is perfectly valid K (and a better way to put this).

But where do `x` and `y` come from? Every K function has three default arguments: `x`, `y` and `z`, if the arguments are not defined beforehand. To name the arguments yourself, you can put them in square brackets:

```
{[firstnum;secondnum] firstnum+secondnum}[1;2]
```

which also does the same thing. If you'd like to document your code, named arguments can be quite useful, but K functions are generally short enough to be understood with the default arguments.

Functions can have multiple statements separated by semicolons:

```
{x+y; x-y}[1;2]
-1
```

A function will always return its last expression, unless specifically told to with the help of `∴`. Any expression which starts with a `:` will be returned, and immediately exit from a function:

```
{:x+y; x-y}[1;2]
3
```

A function can reference itself with the special variable `o`. A recursive factorial, for example, is

```
{$(x>1; x*o[x-1]; 1)}
```

Here, `>` is greater than. `<` is lesser than, and `=` is equal to.

So what does the `$` do? `$` is the K equivalent of an if statement. You *must* give `$` at least 3 arguments for a conditional.

effectively, this translates to the following pseudocode:

```
function(int x) {  
    if (x > 1) {  
        return x * o(x - 1);  
    } else {  
        return 1;  
    }  
}
```

### 3.0.2 `$[;;...] If`

**Symbol:** `$`

**Arguments:** Any number

**Description:** K's If statement. `$[condition; true; false]` or `$[condition1; true1; condition2; true2; ...; false]`.

So what is truth, and what is a falsehood?

The following table lists *all* falsy values in K. The rest are truthy.

Type	Falsy
Number	0
Character	"\0", 0x00
Symbol	` (Empty symbol)
Function	(::)
Array	() (Any empty array)
Dictionary	()!() (Any empty dictionary)

Although these may seem convenient, I highly recommend that your condition always returns 0 or 1.

To group multiple statements in an if-else, you can use a `progn` (lisp term), which is a bunch of statements inside square brackets:

```
{ $[1; [a:x+y;a]; 0] } [5; 6]  
11
```

A `progn`, similar to a function, will always return its last statement unless told otherwise with the help of `::`.

## 3.1 Scoping

You can define variables *within* functions as well, and you can use them within the function only. This means that

```
{a:5; {a}[]}
```

will give you an error. Instead, you can define a global variable using `::`:

```
{a::5; {a}[]}
```

However, this means that the entire program will have access to variable `a`. Global variables are best kept to a minimum in K, and we will see more ways to write our programs without them.

## 3.2 Trains

A *train* is a way of representing simple function composition.

There are some important things to remember when writing trains:

- A train is a list of verbs, and their left arguments if applicable.
- A verb with a `:` after it will only be called with one argument. Without the colon, it is always called with two arguments. This applies to all places where you use verbs, but is especially useful in trains.
- A verb which does not have a `:` after it *must* have a left argument specified i.e `(+)` is not a valid train but `(1+)` is, since `+` is given left argument 1 and `+-` is, since `+` only takes a single argument.
- Every primitive in train must have its *arity* specified with the help of `:`.
- The arity of the last verb of a train determines its arity. eg: `(f g h)` translates to `{f [g [h [x;y]]]}`, and `(f g h:)` translates to `{f [g [h [x]]]}`.

Trains are a way of representing really simple functions, where a list of functions is applied to one or two values. For example, here is a function that displays the first `x` odd numbers:

```
odds:{1+2*!x}
```

What this is doing is: a) Generating a range `0..x-1 - !x` b) multiplying it by 2 (conforming) - `2*` c) adding 1 to it (conforming) - `1+`

to convert this to a train, we have to specify arities:

- `!` is called monadically: `!:`
- `*` is called dyadically: `2*`
- `+` is called dyadically: `1+`

So what we finally get is:

```
odds: 1+2*!:
```

## 3.3 Vocabulary from this lesson

- Variable: a name that is assigned a value.
- Lambda: General curly brace syntax for defining a function.
- progn: A block of statements which returns the value of its last statement, unless specified otherwise with `:`.

- Global variable: A variable that is available to the entire program, assigned with `::`.
- Dictionary: A noun that maps keys to values.
- Train: A sequence of composed verbs and their left arguments.

### 3.4 Exercises

1. Define a lambda function that takes two numbers and subtracts 1 from their sum.
2. Define a train which has the same function as the one from Question 1.
3. Create the array `(1 2 3;4 5 6)` with the help of the verbs you learned in the previous chapter. Then, make a dictionary with the first row as keys, and the second row as values.

## 4 Indexing and Application

K is 0-indexed, which means arrays start at 0.

Indexing into an array takes the element at the given index and returns it.

Indexing into a dictionary takes the value that corresponds to a given key, and returns it.

Passing values to a function will return its corresponding result.

So why Arrays, dictionaries and functions together? Because they all do one single thing: They take a set of values, and return a single value. K hence treats them as the same thing: application.

The first method of application is called **at**.

#### 4.0.1 f@x At

**Symbol:** @

**Args:** (func/array/dict) @ (array/atom)

**Description:** Applies a noun **f** given on its left to a single argument **x** given on its right.

For a function, @ treats **x** like it is a single value, and provides it to **f**.

`(-:) @ 1` gives `-1`.

For a dictionary or an array, @ can take a single index:

`('a`b`c!1 2 3) @ `a -> 1`

`67 78 89 @ 1 -> 78`

or an array of indices:



```
(`a`b`c!1 2 3) @ `a`c`b -> 1 3 2
67 78 89 @ 0 2 -> 67 89
```

All forms of application behave in this manner i.e. they will always index into atomic values and preserve the shape of the indices.

If you index outside of an array's length, you will get a null value corresponding to the type of the array:

```
67 78 89 @ 56 -> 0N
```

**4.0.1.1 Implicit At Indexing** Implicit At Indexing is what happens when K sees two noun values in a row. The simplest way to understand this is with functions:

```
f:{1+x}
f 2
```

Here, K sees two nouns: `f` and `2`, so it applies `f` to `2`. `{1+x}2` and `{1+x}@2` do the exact same thing.

With arrays, you can expect to see stuff like this in a lot of K code:

```
"abcde"1 2
```

This is the same as `"abcde"@1 2`.

Many beginners find this ambiguous as they are not fully clear on when K decides to do this. Let's say that we want to apply `{1+x}` to the square root of `2`. `%x` takes the square root, so we can do `{1+x}%2`. But what about this?

```
{1+x} %2
```

Seems correct, right? K seems to think otherwise:

```
'type
  {1+x}%2
    ^
```

The problem, which isn't obvious through the type error, is that you are not giving K two nouns one after the other.

```
{1+x} %2
|---| ||
|   |+-noun
|   +---verb
+-----noun
```

So it is computing `{1+x}` divided by `2`. It's important to remember that all of K's verbs have overloads for different numbers and types of arguments. Basically, we have written it in a way that makes K use division instead of square root. To fix this, we can wrap the argument in parentheses, making sure K sees a noun.

```

{1+x} (%2)
|---| |--|
|      |
|      +---noun
+-----noun

```

As you may have noticed, `{1+x}%2` is simpler and shorter. Implicit indexing is a nice convenience in many other cases, so don't be afraid to use it!

#### 4.0.2 M-Expressions

The second form of application is called the M-expression, which may be familiar to people coming from the Lisp language. The M-expression is the most general form of application, and it assumes the form of `noun[args]`.

for a function call, you can do `f[a;b;c]`. Remember that function calls include semicolons.

this means that `+ [1;2]` is the same as `1+2`, and you can effectively translate most K programs to use M-expressions only.

For an array, each argument indexes into a different dimension:

```

:a:3 3#1 2 3 4 5 6 7 8 9
(1 2 3
 4 5 6
 7 8 9)

```

The `:` at the beginning of the line is used to force a return value, so `a` gets printed. Now, with the help of M-expressions, you can do `a[1;2]` to get 6, and `a[0;0]` to get 1.

You can also omit a dimension in reshape using `ON` (a null value), like so:

```

3 ON#1 2 3 4 5 6 7 8 9
(1 2 3
 4 5 6
 7 8 9)

```

This will automatically split the array to fit in the given dimension constraint.

Like with `@`, you can also use arrays to index into any dimension: `a[0 1;2]` will give you `3 6`. Indexing always covers a rectangular area regardless of number of dimensions.

The final form of application is dot (`.`).

#### 4.0.3 f.x Dot

**Symbol:** `.`

**Args:** `noun . array`

**Description:** Applies a noun **f** to the arguments given in array **x**.

Dot application takes a noun on the left, and an array on the right. Each array element is taken as an argument.

(+) . 1 2 is the same as 1 + 2. Since + is a primitive, you must put it in parentheses to make it a noun before giving it to ..

**Note:** This behaviour where parenthesizing a verb creates a noun is called *nominalization*. It is mainly useful when a verb takes a function as an argument.

from the previous example, `a[1;2]` can be rewritten as `a . 1 2`, and `a[0;0]` is `a . 0 0`.

## 4.1 Projections

K lets you omit arguments in application.

For functions, omitting arguments implies *partial application*:

`-[1;]` is a function that does `1-x` given an `x`.

`-[;1]` is a function that does `x-1` given an `x`. Basically, Projections will fill in values that you have left empty.

Projections are often used to pass in outside data to a function. This is a common method of omitting global variables, and passing data to inner blocks in K.

For example, this snippet should return a function that adds `a` to its argument.

```
f:{a:x;{a+x}}[1]
f[1]
'value
{a+x}
~
f[1]
~
```

here, `{a+x}` can't see the value of `a`, since `a` is outside of its block, and `a` is not global.

To fix this, we should instead pass `a` via projection:

```
f:{a:x;{x+y}[a]}[1]
f[3]
4
```

as compared to assigning to `a` globally (`f:{a::x;{a+y}}`), this keeps data within the function, and makes the function side-effect free. This is not only useful in K, but helps for any programming task in general.

We can also express this using the shorthand for partial application:

```
f:{a:x;+[a;]}[1]
f[3]
4
```

Now, if you remember from the beginning of this chapter, functions, arrays and dictionaries are all treated the same way in K. So projections can *also* apply to arrays and dictionaries!

For an array, a projection will take all elements from the nth dimension of an array:

```
3 3#!9
(0 1 2
 3 4 5
 6 7 8)
a:3 3#!9
a[1;]
3 4 5
a[;1]
1 4 7
```

For dictionaries, it will index into each nested dictionary within, and pull out the respective values:

```
(1 2 3!(3#,1 2 3!4 5 6))[;2]
1 2 3!5 5 5
```

## 4.2 Vocabulary from this lesson

- Array: A list of elements mapped by indices.
- Dictionary: A list of elements mapped by K values.
- Function: A range of elements mapped by a K expression.
- Application: Getting a corresponding value from a given noun. Can be indexing or calling a function.
- Projection: A partially applied value, obtained by incomplete indexing.

## 4.3 Exercises

1. Get the third column of the array `4 4#!16` using a projection. Then do it without a projection.
2. What happens when you project outside the domain of a noun? What are the differences in arrays, functions and dictionaries?
3. Write a function to add the first two elements of an array, but do it without indexing.

## 5 Adverbs

Any respectable programming language will have its own forms of looping. You have already seen one form of looping in K's recursive functions (o, function self reference). But even outside of recursion, you have seen K's looping through K's conforming operations, since they move through each element of the arrays given to them.

Even with the power of recursion and branching, you will often find yourself in predicaments where these just do not fit the task. K's general design lends itself strongly towards some aspects of functional programming, and many of its adverbs help omit loops and branching to solve tasks concisely and quickly.

If you are acquainted with functional programming already, ' is map, ' (dyadic) is zip, / is foldl, and \ is scanl.

Hence, you will most likely never need recursion and branching in most situations.

Each (') is an adverb which takes a function or verb and applies the function on each element of an array. Yes, the symbol for each is a single quote.

### 5.0.1 f' x Each

**Symbol:** '

**Args:** function' array

**Description:** Apply function to each element of array.

For example, to reverse an array, we can use the | verb:

```
|'(1 2 3; 4 5 6)
(3 2 1
 6 5 4)
```

### 5.0.2 | x Reverse

**Symbol:** |

**Args:** | array

**Description:** reverse the elements of an array.

and to reverse each row of a 2D array, we can do |':

```
a:3 3#!9
(0 1 2
 3 4 5
 6 7 8)
|'a
(2 1 0
 5 4 3)
```

8 7 6)

Each can also be used dyadically, and this form is called eachboth:

### 5.0.3 `x f' y` Each Dyad

**Symbol:** `'`

**Args:** array/atom function' array/atom

**Description:** Apply dyadic function to each element of both arrays.

Each dyad zips two arrays together with a dyadic function, creating an array of the results. Both `x` and `y` must have the same length, unless either of them are atoms. When an atom is given, it is repeated to the length of the other argument (`5 f' 1 2 3` is the same as `5 5 5 f' 1 2 3`).

Each dyad and each are useful with functions that don't fully support conforming operations. For example, `!` (modulus) only penetrates to the atoms in its right argument, meaning `2 ! 3 4 5` will work, but `1 2 3 ! 3 4 5` will not. This is where `!'` comes in handy:

```
1 2 3 ! 3 4 5
1 2 3!3 4 5
1 2 3 !' 3 4 5
0 0 2
1 2 3 !' 5
0 1 2
```

Each dyad will repeat atoms to the length of the other argument, but sometimes you need an array to behave like an atom. Say, something like this:

```
1 2 3 !' (5 6; 5 6; 5 6) / not very ideal
(0 0
1 0
2 0)
```

One way to solve it is using a projection:

```
![:,5 6]' 1 2 3
(0 0
1 0
2 0)
```

But the best way to do it is with `eachleft(\:)`:

```
1 2 3 !\: 5 6
(0 0
1 0
2 0)
```

Eachleft applies a dyadic function to each element in the left array and the entire right argument.

#### 5.0.4 `x f\:` **Each Left**

**Symbol:** `\:`

**Args:** array/atom function\: array/atom

**Description:** Apply dyadic function to each element of `x` and the whole of `y`.

For example, in `(a; b; c) f\:` `d`, we will get `(f[a;d]; f[b;d]; f[c;d])`.

The same applies for each right, which you can probably guess the symbol for (`/:`).

### 5.1 Folds and Scans

Folds and scans are used to combine values from an array to produce one or many results.

#### 5.1.1 `x f/` **y Over (Fold from left)**

**Symbol:** `/`

**Args:** any dyad/ array

**Description:** Fold array `y` from left with a dyadic function. `x` is optional.

There are many, many useful applications of folds, some of which are:

Fold	Verb	What it Does	Example
<code>+/</code>	Add	Sum	<code>+/1 2 3 -&gt; 6</code>
<code>*/</code>	Multiply	Product	<code>*/3 4 5 -&gt; 60</code>
<code> /</code>	Maximum	Maximum	<code> /3 5 1 -&gt; 5,  /1 0 0 -&gt; 1</code>
<code>&amp;/</code>	Minimum	Minimum	<code>&amp;/3 5 1 -&gt; 1, &amp;/1 0 1 1 -&gt; 0</code>
<code>,/</code>	Concatenate	Join all elements together (Raze)	<code>,/(1 2 3; 4 5 6) -&gt; 1 2 3 4 5 6</code>

These folds can also be used with an initial element, providing a fallback value:

```
0+/1 2 3
6
0+/( )
0
1*/7 8 9
504
1*/( )
1
```

With an initial element, these are fine. But what should happen *without* one?

```

+/( )
0
*/( )
1
|/( )
-9223372036854775807
&/( )
9223372036854775807
,/( )
( )

```

Sum and Product seem OK, but `|/` and `&/` have two meanings, since max and min mean different things for integers and booleans: the highest integer is what K returns by default, but the max and min booleans are 1 and 0. Hence, adding an initial element does the trick on booleans:

```

0|/( )
0
1&/( )
1

```

### 5.1.2 `x f\ y` Scan (Scan from left)

**Symbol:** `\`

**Args:** any dyad\ array

**Description:** Fold array `y` from left with a dyadic function, producing intermediate results in an array. `x` is optional.

Scan is the same as fold, *except* it will also give you the intermediate values of the fold.

```

+\ 1 2 3 4 5      = 1 3 6 10 15
  1                = 1
  1+2              = 3
    3+3            = 6
      6+4          = 10
        10+5       = 15

```

They are useful all by themselves, but they are also *very* useful for debugging fold functions and finding out what's going wrong midway.

```

*/7#1000
3875820019684212736

```

```

*\7#1000
1000 1000000 10000000000 10000000000000 10000000000000000 10000000000000000000 3875820019684212736

```



In the above example, we can see that integer overflow resulted in an "incorrect" result.

Scans are often useful with boolean functions (i.e. `<`, `=`, `>`, `&`, `|`). For example, to "turn off" all 1s that appear after the first 0, we can use:

```
&\1 1 1 0 0 1 1 1 0 1 0 1
1 1 1 0 0 0 0 0 0 0 0 0
```

Scans are also one of the main ways that K lets you access intermediate values from loops, making it widely useful. For example, to count or number runs of 1s in some boolean input:

```
{y+x*y}\1 1 1 0 0 1 1 1 0 1 0 1
1 2 3 0 0 1 2 3 0 1 0 1
```

*Fold* and *scan* both use the first argument as initial value, and the rest of the arguments as input arrays. Note that scans of empty inputs return the (empty) input unchanged.

```
4+\1 2 3
5 7 10
0+\()
()
2*\7 8 9
14 112 1008
1*\()
()
```

## 5.2 Iterated Fold and Scan

Fold and Scan both have overloads for more than two arguments. We will be looking at examples of iterated scan only, since fold returns the last result of scan anyway.

### 5.2.1 `f\[x;...y]` Iterated Scan

**Symbol:** `\`

**Args:** `n-ad\[int;...arrays]`

**Description:** The length of `arrays` and the argument count of `n-ad` must be the same.

- Do the following steps `x` times:
- Take the last `n` elements of `arrays`, call this `last`
- Add the result of `f.last` to the end of `arrays`
- Return the first `x+1` elements of `arrays`.

A simple use of this is to construct the fibonacci sequence:

```
+\[10;0;1]
0 1 1 2 3 5 8 13 21 34 55
```

There is another overload for this scan, where `x` is a list. In that case, `x` is treated as an initial value of sorts, and the `arrays` are iterated through. The first value is `f.(,x),arrays[;0]`, then `x` is set to that value. The next iteration we use `arrays[;1]`, and so on. You can see the difference with this example:

```
{x+y*z}\[3; 1 2 3; 4 5 6;7 8 9]
(1 2 3
 4 5 6
 7 8 9
29 42 57)
{x+y*z}\[1 2 3; 4 5 6;7 8 9]
(29 30 31
 69 70 71
123 124 125)
```

Similar to ordinary fold/scan, the iterated versions simplify stateful computation. However, these overloads tend to be more niche in their use.

### 5.3 Vocabulary from this lesson

- Adverb: A syntactic form which acts like a higher order function on both nouns and verbs.
- Each: Zap each element with function
- EachBoth: Zip elements of two arrays with a function
- Eachleft and Eachright: Zip each element in one argument with the whole other argument with a function.
- Fold: Condense an array into a single value using a function.
- Scan: fold but outputs intermediate values as an array.

### 5.4 Exercises

1. `|` when used with a single argument, reverses an array. Instead, reverse an array with the usage of a single fold.
2. Write a function to sum each row of an array.
3. Convert an array of base-2 digits to a base-10 integer using a fold. `f/1 0 1 -> 5`.
4. Write a function to get the prefixes of an array using an adverb: `f 1 2 3 -> (,1;1 2;1 2 3)`

## 6 Numbers and Logic

K does not like branching, but it likes logic very much. What this means is that you will be doing a lot of interesting things with integers in order to avoid a branch.

The basic logical verbs of K are:

- > greater than
- < lesser than
- = equal to
- ~ exact match
- ~: not
- ^ is null?
- & minimum / and
- | maximum / or

The other comparison related verbs outside of these are Grade Up (<) and Grade Down (>). These functions are the sorting functions of K.

Grade Up and Down both return the indices of the values of the array in their final locations. Some would say that they generate a *permutation vector*. The function `{x@<x}` sorts an array in ascending order, for example. Effectively, < and > allows you to sort an array using any other array as "weights", making it much more flexible.

All of the simple logical verbs, with the exception of *exact match* support conforming operations, and return 1 for true, and 0 for false. Since booleans are also numbers in K, this lets us do some interesting things with them.

For example, let's say we want to increment all multiples of 2 in an array. Usually, one would check each number one at a time, and double it if it were even.

To check if a number is even, we can use `mod (atom ! array)`:

```
{2!x}2 3 4
0 1 0
```

We can then negate this to put 1s at the spots where even numbers are:

```
{~2!x}2 3 4
1 0 1
```

And then, add it to the array:

```
{x+~2!x}2 3 4
3 3 5
```

An important function that works on boolean arrays is *where* (&). Where takes a list of integers and repeats their indices by their values:

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

What makes & interesting is that it can also repeat an index 0 times, thereby discarding it.

& can be used to filter elements based on a condition, like `{y@&x'y}`, which keep elements in y that satisfy predicate x. +/- If you want the index of the first

element that holds true under a condition, you can use `*&`. But a simpler and more efficient alternative is `?` (`find`).

### 6.0.1 `x?y` Find

**Symbol:** `?`

**Args:** `array ? array`

**Description:** Index of each element of `x` in `y`. If not found, return `0N`.

`0N` is a null value, as described in *Arrays, Functions and Dictionaries*. A null value is generally used to signify the absence of a value, and generally denote a special value in a given type. The most usual place where null values are found are in indexing, where an out of bounds index returns a null value, and in `find`.

#### Null Values for each data type

Type	Null
Integer	<code>0N</code>
Float	<code>0n</code>
Character	<code>" "</code> (Space)
Symbol	<code>`</code> (no characters after backtick)
Function	<code>(::)</code> (identity function)

`::` (monadic `:`) has been used in earlier chapters. It is used to force returns, and it always returns its argument without modification.

Sometimes, there is confusion between `0N` and `0n`, so you can use `@` (type) to distinguish between them:

```
@'(0N; 0n; " "; `; ::)
`i`d`c`s`u
```

`i` for integer, `d` for double, `c` for character, `s` for symbol, `f` for function and so on. If given an array that is of a single type, these are capitalized. More info on ngn/k's types can be read about on the K Wiki.

Since these null values are not very easy to keep track of in an array, there are two primitives that are made specifically for dealing with them:

- `Null?` (`^`) which tells you which atoms in an array are null values (1 for null, 0 for not null).

```
^(1;0N;" ")
0 1 1
```

- `Fill` (`^`) which substitutes an element in places where there are null values.

```
"A"^(1;0N;" ")
(1
 "A"
 "A")
```

## 6.1 Vocabulary from this lesson

- Grade Up/Down: A special type of sorting function that provides indices of the final sorted values in the array.
- Null Value: In K, every type has its own null value. Null values are placeholders, and all types of nulls can be detected with `^`.

## 6.2 Exercises

1. Given an array `x` and a 'mask' array `y` consisting of zeroes and ones only, get the elements in `x` that are at the same positions as the ones in `y`. `f [2 7 89 92 -123;0 1 0 1 1] -> 7 92 -123`.
2. Create a dictionary from an array where the keys are the types of the elements in the array, and the values are the elements of the array.
3. Create a function with two arguments `x` and `y`, which takes the sum of `x[y]` without nulls.

## 7 More Adverbs

Most of the time, you can manage to write good K programs with neat little `eaches` and `folds` and `scans`. But *some* algorithms choose not to use arrays. Some algorithms insist on themselves, and that's why K has its own version of explicit loops from other languages:

When given a number `n` and a function on the left, `/` acts as a *for* loop:

```
3 {1+x}/0
3
```

A `for` loop will apply a monadic function `n` times to a given value.

### 7.0.1 `x f/ y` For

**Symbol:** `/`

**Args:** `int monad/ any`

**Description:** Apply function `f` `x` times to value `y`.

The same overload for `\` will produce the intermediate values of the *for*, like a `scan`:

### 7.0.2 `x f\ y` For (Scan)

**Symbol:** `\`

**Args:** `int monad\ any`

**Description:** Apply function `f` `x` times to value `y`. Collect intermediate values of `y` in an array.

```
3 {1+x}\0
0 1 2 3
```

### 7.0.3 `f g/ z` While

**Symbol:** `/`

**Args:** `monad monad/ any`

**Description:** Apply function `g` to `z` until `f[z]` is falsy.

When given two monadic functions on the left, `/` acts as a while:

```
{x<5}{1+x}/0 # While x is less than 5, add 1 to x.
5
```

The truthy and falsy values for while are the same as it is for `If($)`. You can reshuffle on what they are by checking Part 3.

While-scan is quite self-explanatory (It provides the intermediate results of a while loop):

```
{x<5}{1+x}\0
0 1 2 3 4 5
```

## 7.1 Other looping adverbs

The following are some of the more specialized looping adverbs in K which don't exactly fit in the previous two categories.

Let's start with *converge*:

```
{1_x}/1 2 3 4 / Drop the first element till the array no longer changes.
!0
```

Converge applies a verb to a value till the value no longer changes, and it's also called *fixpoint* due to that reason.

### 7.1.1 `g/ z` Converge

**Symbol:** `/`

**Args:** `monad/ any`

**Description:** Apply function `g` to `z` until `g[z]` matches `z`.

### 7.1.2 `x _ y Drop`

**Symbol:** `_`

**Args:** integer `_` array

**Description:** Drop the first `x` elements of `y`. If `x` is negative, drop the last `x` elements.

*Converge-scan* is the `scan(\)` version of the same, and returns the intermediate results:

```

{1_x}\1 2 3 4
(1 2 3 4
 2 3 4
 3 4
 ,4
 !0)

```

*eachprior* works like *each*, but instead applies a dyadic function to each element and the one before it, useful for many pairwise operations.

Since the first element of an array doesn't have anything before it, *eachprior* allows an optional left argument.

```

45 ,':1 2 3 4
(1 45
 2 1
 3 2
 4 3)
-':1 2 3 4 / deltas
1 1 1 1
+':1 2 3 4
1 3 5 7

```

### 7.1.3 `x f': y Eachprior`

**Symbol:** `'`

**Args:** any dyad `'`: array

**Description:** for each element `b` in the array and the element `a` before it, calculate `f[b;a]`. If `x` is not provided, The first element is left unchanged.

Stencil is a more general version of what *eachprior* does:

```

3_:'':"ABcdE"
("abc"
 "bcd"
 "cde")

```

The main difference between `stencil` and `eachprior` is that `stencil` applies a function to each `n`-length chunk, instead of passing in elements as arguments.

#### 7.1.4 `x f': y` Stencil

**Symbol:** `'`

**Args:** `int monad'`: array

**Description:** apply `f` to each overlapping `x`-length chunk in array `y`.

To simply get the chunks of the array, we can use the identity function `::: num ::': array`, but there is a more convenient adverb for it: `x ': array`, which is called *window*.

#### 7.1.5 `x ': y` Window

**Symbol:** `'`

**Args:** `int '`: array

**Description:** get each overlapping `x`-length chunk in array `y`.

It may be quite surprising at first that `window` does not take a function, but be not afraid. `Window` behaves like any other function, and can be stored as well.

Why isn't *window* a verb, if it doesn't take a function, then? A K interpreter has to make the distinction because it's an overload of an adverb symbol.

## 7.2 More adverb overloads

Now that you have seen `window`, we can look at some more neat adverbs:

- *base encode* (`int(x) \ int(y)`) converts a base-10 integer to an array of digits specified by the bases on the left. If `y` is an array, every number in `y` is encoded into a column.
- *base decode* (`int(x) / int(y)`) is the inverse of *base encode*.
- *split* (`string(x) \ string(y)`) splits `y` on string `x`.
- *join* (`string(x) / array(y)`) joins array `y` on string `x`.
- *binary search* (`array(x) ' any(y)`) does a binary search for each element of `x` in `y`. If you do not know what a binary search is, I recommend reading this article.

These *can* be implemented using the help of K's primitives (and K's philosophy does support that), but having them as symbols is quite convenient, since they are very commonly used.

## 7.3 Not Really Adverbs

Two important functional programming primitives: `#(filter)` and `_(filter-out)` are not adverbs, since they don't quite fit in the overloads, but they are still quite



important in general code.

Generally, filtering in K is done with the help of `&`(where), which repeats the index of each element by the value of each element:

```
&1 0 1 0 1
0 2 4
```

### 7.3.1 `& x` Where

**Symbol:** `&`

**Args:** `& int_array`

**Description:** Repeat the index of each element in `x` by the value of each element in `x`.

this can be used in tandem with `@` to select values from an array:

```
("Raze"; "blah"; "ti"; "blah"; "me")@&1 0 1 0 1
("Raze"
 "ti"
 "me")
```

This can then be extended to use a condition on the array, keeping elements with length lesser than 5:

```
{x@&(5>#:) 'x}@("K"; "looong"; "is"; "blaah"; "cool")
("K"
 "is"
 "cool")
```

*Filter* and *filter-out* are shorthands for this type of filtering. They expect a monadic function (given as a noun), and apply it to the whole array.

Then, the rest is done the same way as it is with `&` and `@`. The previous example can be written as:

```
((5>#:)')#("K"; "looong"; "is"; "blaah"; "cool")
("K"
 "is"
 "cool")
```

Remember that *filter* and *filter-out* are verbs, and hence expect function values and not verbs. this means `~#x` will not work, but `(~:)#x` will.

## 7.4 Vocabulary from this chapter

- Window: a continuous run of elements in an array.
- Binary Search: A search that can find the position an element should be inserted after in a sorted array.

## 7.5 Exercises

1. *Converge* (*g/ z*) is a special case of *while*. Implement your own *converge* function using a *while*.
2. Implement a function that converts a number to a list of digits in a given base (use *while/fixpoint*).

## 8 Dictionaries, Tables, Strings and other Miscellaneous functions

Dictionaries and strings are not highly related, just that this chapter aims to show the rest of the verbs we have not covered in K, which happen to be mainly relating to dictionaries and strings. Think of it as a rapid-fire explanation of their quirks.

### 8.1 Dictionary Trivia

Dictionaries are an *ordered collections*. This means that many primitives which work on arrays work similarly on dictionaries. This includes:

- *concat* , (values in *y* overwrite values in *x*)
- *reverse* | (reverses key array and value array separately)
- *map* ' (applies on values)
- *first* \* (first value from key-value pair only)
- *not* ~
- *at* @ preserves the structure of the dictionary when given as the second argument.
- *filter* # and *filter-out* \_ (filter by value)
- *where* & (keys are repeated by values)
- *grade up* < and *grade down* > (sort keys by values)

Outside of these, the most complex of the dictionary functions is *group* (=), which group the indices of elements by their values.

For example, we can get the counts of the elements of an array:

```
:a:="aaaabbbbbaaaacccc"
"abc"!(0 1 2 3 9 10 11 12;4 5 6 7 8;13 14 15 16)
:b: #'a
"abc"!8 5 4
```

But the result is still a dictionary. Luckily, we can separate the keys with *!*, and the values with *..*

```
!b
"abc"
.b
8 5 4
```

### 8.1.1 Tables

Tables are flipped dictionaries. A dictionary can be flipped with + if:

- All keys are symbols
- All values are of the same length

Tables are dictionaries which allow indexing of columns with symbols, and rows using integers.

```
:a:+`a`b`c!3 3#!9
+`a`b`c!(0 1 2;3 4 5;6 7 8)
a[1]
`a`b`c!1 4 7
a[`a]
0 1 2
```

## 8.2 String functions

\$ will cast any atomic value to a string. It also conforms to arrays and dictionaries:

```
$(`symbol; 123; {x+y}; 1 2 3!4 5 6)
("symbol"
"123"
"{x+y}"
1 2 3!(,"4";,"5";,"6"))
```

Dyadic \$ casts between different types:

```
`i$"dsfdasfsd" / Character to Integer
100 115 102 100 97 115 102 115 100
```

Monadic \_ floors numbers and lowercases strings:

```
_ (1.1;2.2;"ABC123")
(1
2
"abc123")
```

Dyadic \$ is pad with spaces. Padding conforms to atoms on the left, and strings on the right. Positive integers pad on the right, and negative integers pad on the left.

```
(3;4;(5;6))$("xy";"za";("bc";"cd"))
("xy  "
"za   "
("bc   "
"cd    "))
```

Some adverbs have useful string related overloads:

x \ y is split. Splits y on x.

```
" \"string with spaces"
("string"
 "with"
 "spaces")
```

Naturally, `x / y` is join. Joins `y` on `x`.

```
"|"/("string";"without";"spaces")
"string|without|spaces"
```

Note that these adverbs *only* work on strings and nothing else.

### 8.2.1 The Rest

**8.2.1.1 Random number functions** Dyadic `?` when given two integers `x` and `y` generates `x` random integers from `!y`. If `x` is negative, this will be a random permutation.

If `y` is an array, then it will deal `x` elements from it. Both of these allow repetitions.

Negative `x` will not allow repetition, and if the absolute value of `x` is greater than the length of `y`, then `K` will throw a length error.

Monadic `?` when given a single integer `x` will generate an array of random floats between 0 and 1 of length `x`,

Finally, `ON?x` where `x` is an array will shuffle `x`. If `x` is a number, then `!x` is used.

**8.2.1.2 Base Decode and Encode (`\` and `/`)** `ngn/k`'s base conversion adverbs are generalized: this means that you can do mixed-radix conversion and other cool things. These adverbs will be familiar to APL programmers. Remember that these are adverbs, and not verbs. Some things you do with them may not work exactly as expected.

The simplest use, and the most often use for this, is conversion to and from base 2:

```
2 \ 5
5
'rank
~
```

```
2 / 5
2
```

Wait a moment: that isn't right. Both of these results are wrong, and one of them is an error!

**Print debugging and comments** The reason for this is one of K's limitations: \ and / are interpreted differently based on the whitespace before them, and hence, the space before them changes their meaning.

Hence \ 5 is hence interpreted as "pretty print 5 to stdout, and return its value", and / 5 is interpreted as a K comment.

In order to fix this, just remove the space.

```
2\5
1 0 1
2/1 0 1
5
```

For mixed radix, we can try some time conversion: hours, minutes and seconds to seconds:

```
24 60 60/2 46 40
10000
```

These adverbs also generalize to arrays, and can perform base conversion on multiple numbers:

```
2\5 6 7
(1 1 1
0 1 1
1 0 1)
2/(1 1 1;0 1 1;1 0 1)
5 6 7
```

**8.2.1.3 Miscellaneous** Monadic = when given a single integer  $x$  will produce an identity matrix of side length  $x$ .

```
=3
(1 0 0
0 1 0
0 0 1)
```

Monadic . when given a string is eval. This will evaluate any string as K code. Although eval is a convenient method to parse K-like strings, it is slow and dangerous. Use it sparingly.

```
."1+2"
3
```

Dyadic : is right. it returns its right argument. : has many special meanings, so it is required to call : with an M-Expression:

```
: [1 2; 3 4]
3 4
```

*no new vocabulary for this chapter.*

### 8.3 Exercises

1. Write a function to convert seconds into hours, minutes and seconds, using `\.`
2. Write a function that takes a list of numbers `x` and a number `y`. Group the numbers in `x` into a dictionary based on whether they are multiples of `y`. `f [1 3 5 9;3] -> (0;1)!(1 5;3 9)`

## 9 Special Forms

A special form is a kind of verb that can perform specialized operations like control flow and global modification. Special forms are generally polyadic (more than 2 arguments).

You have already seen one special form: `If ($[; ; ..])`. We will be covering the rest of these special forms in this chapter.

### 9.1 Amend (@)

#### 9.1.1 3 Arguments (@[x;y;z])

Given array `x`, indices `y` and monadic function `z`, Amend will apply `z` to the elements at indices `y`.

```
@["ABC";1 2;_:]  
"Abc"
```

This is useful when you don't want to use assignment to modify specific elements in an array.

#### 9.1.2 4 Arguments (@[x;y;z;u])

Same as 3 argument amend, but `z` is dyadic, and `u` is the second argument provided to `z`.

```
@[2 3;0;+;4]  
6 3
```

### 9.2 Drill / Deep Amend (.)

#### 9.2.1 3 Arguments (.[x;y;z])

Similar to amend, `x` is an array, but `y` is a single multidimensional index, and `z` is a monadic function.

#### 9.2.2 4 Arguments (.[x;y;z;u])

Similar to 3 arguments, but with `z` dyadic, and `u` as the second argument.

### 9.3 Try (. [x;y;z])

An error handling mechanism.

**x** is a function, **y** is an array of arguments to **x**, and **z** is the function that is applied to the error message (see ch 11), if **x.y** fails. Otherwise returns **x.y**. *Try* is most useful when you are in deeper levels of nesting in a function, and print debugging with `\` (trace) is not completely helpful.

```
. [+;1 2;"E:",]  
3  
. [+;1,`2;"E:",]  
"E:'limit\n"
```

### 9.4 Splice (?[x;y;z])

Splices an array **z** between the indices given in **y** in array **x**. A very useful string function.

```
?["abcd";1 3;"xyz"]  
"axyzd"
```

## 10 Input and Output

The main way of performing input and output in K is via its I/O verbs.

Every I/O verb has a different data format. However, the destination and source can be any of the following:

- the empty symbol ```, which represents STDIN and STDOUT
- An absolute path like `"/path/to/file"`
- A relative path Input, Output and System functions

The main method of I/O(input/output) in K is via its input verbs: an input verb is denoted by a single digit, followed by a `:` symbol.

The number of an input verb denotes its data format:

- `0:` takes in arrays of character arrays (lines).
- `1:` takes in arrays of bytes.

You have already seen `0:` used for output in the first chapter.

Every I/O verb reads input when used monadically, and writes output when used dyadically. The general form is as follows:

**destination** <Output Verb> **data**

and

<input verb> **source**

every I/O verb has a different data format. However, the destination and source can be any of the following:

- the empty symbol ```, which represents STDIN and STDOUT
- An absolute path like `"/path/to/file"`
- A relative path like `"/path/to/file"` (relative to the directory the K interpreter is run in)
- An IP path in the form `"host:port"` where `host` can be omitted to imply `127.0.0.1`
- A file descriptor number which can be obtained via the open verb `<`

The *open* verb(`<`) initiates a stream of input/output to and from a given location, which can be from the first 4 types given above.

Every opened stream must then be closed with `>` (close) before the end of your K program.

Now that we know what the input numbers take, we can finally understand what the basic K Hello world program does:

```
` 0:"Hello, World!"
| |
| v
| takes a string or list of strings
v
STDOUT
```

To prompt for input, we do `0:`` , which will read a single line of input from STDIN as a string.

A simple program which prompts and returns output would look like this:

```
input: -1_1:`
Razetime
`1:"Hello ",input,", I'm Dad!"
Hello Razetime, I'm Dad!
```

We have to use a `1_` here since `1:` will read a newline as part of the input. You can see this by just checking `1:`'s output:

```
1:`
hello
"hello\n"
```

Now, you should have all you need to write interactive programs in K!

## 10.1 Command line arguments and other console values

The other general way of getting input is through command line arguments.

Command line arguments (and all special reserved values) are represented by symbols. The name used for Command line arguments is ``argv`.



These are the command line arguments I get when running the ngn/k repl:

```
`argv[]  
("/home/razetime/Software/k/k"  
"/home/razetime/Software/k/repl.k")
```

The first argument will always be the K binary, and the second argument will be the path to your K program, so usually you would want to do `2_`argv[]` and then take the arguments you need.

Here are some of the other useful console values:

- ``t` gives the current time in microseconds.
- ``env` returns a dictionary containing the names of all environment variables mapped to their values.
- ``prng` is a pseudo-random number generator.

There are a few more special symbols available in the K repl (like ``json`) which you can find by typing `\`` in the REPL.

## 10.2 Vocabulary from this chapter

- I/O verbs: Special verbs which perform input and output with a specific data format.
- File Descriptor: An integer bound to an I/O stream. Can be used with I/O verbs.
- Special Symbols: The reserved words of K.

## 10.3 Exercises

1. Write a program that takes two numbers from STDIN and adds them.
2. Write a program that returns its odd numbered command line arguments (the first, then the third, and so on).

# 11 Debugging and Errors

*much of this chapter was populated with ngn's help.*

One thing that array languages used to be (and still are) notorious for are their terse error messages. This is often a hill to climb for most programmers. APL, J and K4 try to work around this problem with their debugging tools, but the common thing that they all share is that errors are not very descriptive. Array language expect you to use your knowledge of the symbols to guide the debugging process.

K does not have complex debugging tools available, and primarily favours print debugging. The types of errors that can happen are small in number, generally provide the position of the error, and errors take up very less space on screen.

Hence, K debugging is usually simpler than one expects. If you like long, detailed explanations of what you need to do, then K may not be for you.

This page will disambiguate all the possible errors that ngn/k can throw, and what can cause them.

A K error will generally have a) an error name and b) A message prefixed with a quote and b) a few lines pointing to the code where the error happened. Errors will be disambiguated first on error name, then the situation it may appear in.

### 11.0.1 'parse

A parse error means that the K interpreter has found malformed code. This is also called a syntax error.

#### Examples:

```
/ mismatched bracket:
({
/ unclosed string:
"fsdfsad
/ bad parameter lists:
{[x;1]1}
/ bad hex literals:
0xfgh
/ odd number of digits in hex:
0x123
/ nonexistent io verbs:
9:~
```

### 11.0.2 'rank

A rank error occurs when a primitive is called with arguments it does not have an overload for. This can be due to mismatched argument types, or a mismatched number of arguments.

#### Example:

```
/ + takes only 1 or 2 args:
+[1;2;3;4]
```

### 11.0.3 'type

When the type of a value is incompatible with a primitive, a type error is thrown.

#### Examples:

```
1+`a
!"a"
&`symbol
```

#### 11.0.4 'domain

A domain error occurs when the argument types for a primitive match, their lengths are correct, but the values of the arguments are not supported (they are not in the right domain).

##### Examples:

```
(0;1 2)!3      / arrays cannot be keys
&-1           / cannot repeat by negative amount
1?-1          / negative range does not work
1 2_()         / empty array cannot be dropped from
-1 2_(1 2;3 4) / negative drop values do not work on a 2d array
```

#### 11.0.5 'index

An index error triggers when you assign outside the bound of an array. This can be done by normal assignment, or with the help of *amend*.

##### Example:

```
a:1 2 3;a[-7]:0 / assigning to an index that does not exist
@[1 2 3;0 5;;8 9] / amending an index that does not exist
```

#### 11.0.6 'length

A length error occurs when the lengths of two arguments do not conform to each other in an *each* (') or a conforming operation.

**Example:** 1 2 3 + 1 2

#### 11.0.7 'value

A value error occurs on variable names that are not declared yet.

**Example:** a+5 will error if a is not defined.

#### 11.0.8 'nyi

NYI means not yet implemented. It triggers on overloads which are in ngn/k's design, but are not implemented yet. 'nyi errors change from time to time, and if you do get one, you have to find a workaround for the primitive you are trying to use.

#### 11.0.9 'stack

A stack overflow happens when a function recursively calls itself too much.

##### Example:

```

{o}[]
{o}
{o[]}[]
'stack
{o[]}
~
{o[]}
~
{o[]}
~
{o[]}
~
..

```

#### 11.0.10 'io

An IO error happens when there is a problem during the usage of an I/O verb. This can be:

- A file permissions problem
- file access problem
- problem with the path given to the verb

#### 11.0.11 'compile

ngn/k first compiles your code to bytecode. This indicates a problem in the bytecode compilation phase.

#### 11.0.12 'limit

All functions in ngn/k K have the following limits:

- 8 local variables (including parameters)
- max bytecode size per function - 256 bytes

If a function exceeds these limits, you will get this error.

#### Example:

```
{[a;b;c;d;e;f;g;h;i]a} / i is the 9th variable
```

#### 11.0.13 'eoleof

Every K program must end with a trailing empty line. If you get this error often, you may want to ask your editor to insert one. Most popular editors have a way to do it.

#### 11.0.14 'oom

Out of memory. Your program seems to be gobbling up large amounts of memory. This often happens in conditional iteration where values tend to get really big.

oom is a non recoverable error i.e. K will not point to the line where everything went wrong, so you'll need to check for the location yourself.

### 11.1 Finding and throwing errors

Print debugging is the main method of debugging in K. The general method of printing ( ``0:` ) is quite clunky in this regard, so instead you should use `\` (trace, note the leading space), which autoformats and prints the K value given to it, and then returns it. Trace is an adverb, and hence will be executed like an adverb, right to left.

You can also throw custom errors using ``err`, which will throw an error with the name of the string given, along with the line the error was thrown on.

```
`err@"Fsd fsdfA"
'Fsd fsdfA
`err@"Fsd fsdfA"
^
```

### 11.2 Inspecting functions internal bytecode

All user defined functions in K are compiled to a bytecode. It can be inspected using `!`:

```
!{n:42; x+y+n+1}
("{n:42; x+y+n+1}"           / function code itself
 0x0268424a67694a214921502100 / bytecode
 0xff00020202000b0c090a070800 / bytecode metadata (which local variable to use, reference c
`x`y`n                        / list of local variables
::                             / constants used
42
1)
```

### 11.3 Vocabulary from this chapter

none.

### 11.4 Exercises

- Try getting a 'io error on your machine.
- Now that you know how to debug your K functions, you may want to check out the previous exercises and figure out some of the questions you were not able to finish.

## 12 Thinking in an array language

*You can view the full source code for this chapter at [GitHub](#).*

Since you are now properly acquainted with K, let's do some programming. Most K programming happens through the REPL, because it is very useful to iterate upon previous code. ngn/k with rlfe has history with the up/down arrow keys, and that should be more than enough to begin developing bigger programs in K. Functions are tested in the REPL, and then moved to actual code. Note that ngn/k's prettyprinting always returns valid k data, and you can precompute some things beforehand to speed up your program.

A K script is always executed like it was typed in the repl, that is: Each line is executed, and its return value is printed *unless* it ends with a semicolon. A script also allows multiline definitions, which are convenient for readability. Oftentimes, you may save your work in a script, and want to use it in a repl. In order to use your stored data and functions, just do `\l file.k` in the repl, and your file will be executed, and its data will be loaded. You can load a file into the REPL more than once, overwriting older data. The repl help accessed with `\` lists more useful commands as well.

K programming (and array programming in general), is a continuous process of simplifying your patterns. A big, unwieldy pattern has one or more ways to condense to a smaller, more declarative, easy to read pattern. This is discussed in a lot of detail in Patterns and Anti-patterns in APL: Escaping the Beginner's Plateau - Aaron Hsu - Dyalog '17, if you'd like to understand it better.

A common problem most people have in K is the need to translate a common, well known algorithm to K, usually taken from a programming website like geeksforgeeks, or a Wikipedia article. Let us take an example: Matrix Multiplication.

From this wikipedia article, the iterative algorithm for matrix multiplication is as follows:

```
Input: matrices A and B
Let C be a new matrix of the appropriate size
For i from 1 to n:
  For j from 1 to p:
    Let sum = 0
    For k from 1 to m:
      Set sum ← sum + Aik × Bkj
    Set Cij ← sum
Return C
```

If you want, you can try translating this to K. A direct translation would be:

```
matmul: {
  A::x
  B::y
```

```

n::#A
m::#*A
p::#*B
C::(n;p)#0
i::0
j::0
k::0
sum::0
{
  i::x
  {
    j::x
    sum::0
    {
      k::x
      sum::sum+A[i;k]*B[k;j]
    }'!m
    C[i;j]::sum
  }'!p
}'!n
C}

```

This is the worst K code I've ever written, because we are trying to write K like an imperative language, and K doesn't work well with that design. The main problems are:

- Many, many globals are assigned
- multiple nested loops
- lots of modification

Luckily, there are a lot of things we can simplify here, and we can address these problems one by one.

Let us begin at the innermost loop:

```

sum::0
{
  k::x
  sum::sum+A[i;k]*B[k;j]
}'!m
C[i;j]::sum

```

The first and simplest fix we can make is summing using a fold (/).

```

C[i;j]::+/{
  k::x
  A[i;k]*B[k;j]
}'!m

```

One global down, 9 more to go.

The next global we can remove is `C`. Since `'` (each) returns an array, `C` doesn't need to be modified. We can simply return the value of the nested loop.

```
{
  i::x
  {
    j::x
    +/{
      k::x
      A[i;k]*B[k;j] }'!m }'!p }'!n}
```

Now, we have three loops with no modification, which makes our job much easier. The main variables to look at now are `i`, `j`, and `k`.

- `i` indexes each row of `A`.
- `j` indexes each column of `B`.
- `k` indexes each column of `A` and row of `B`.

Basically, `k` is responsible for pairing each row of `A` with each column of `B`, which are then multiplied. Hence, we can eliminate the middle man here, and directly match them without `k`. This also eliminates one loop, and removes the need for `m`.

```
{
  j::x
  +/A[i]*B[:,j] }'!p }'!n}
```

Next, to remove `j`, we need to take each column of `B` and pair it with `A[i]`. To do this, we transpose `B` and pair each element with eachright (`/:`).

```
matmul: {
  A::x
  B::y
  n::#A
  i::0
  {
    i::x
    A[i]{+/x*y}/:+B}'!n }
```

In order to remove `i`, we do a similar thing: Use `eachleft` to pair each row of `A` with each column of `B`.

```
matmul: {
  A::x
  B::y
  A{+/x*y}/:\:+B }
```

We need no more globals!

```
matmul: {x{+/x*y}/:\:+y}
```



Now *that* is matrix multiplication in K. This is the most direct algorithmic conversion of matrix multiplication. Now we will look at ways to shorten it, and remove more loops.

`+` (transpose) is costly, and we can remove it. What we are currently doing is naive. Instead of multiplying each row of `x` with each column of `y`, we can conform each row of `B` to the whole of `A`, doing the same thing implicitly.

```
matmul: {x{+/x*y}\:y}
```

Now, we have a function which can be easily made tacit. With the rules from Chapter 3, we get our final result:

```
matmul: (+/*)\:
```

A matrix multiplication function you can be proud of. This process may seem like it has a lot of steps, but condensing code will become much easier and intuitive as you practice your skill in K.

Matrix multiplication is a simple procedure which works well with K's array support. We will be seeing more algorithms that don't play well with K, and how to handle them in future chapters.

## 13 Writing a Prelude

One of K's most important design constraints is the QWERTY keyboard: K keeps all the symbols, and we get to have all the letters. Since we have all the letters, and K itself has few symbols to work with, many people build their own K preludes, which contain things that they use regularly.

Apart from words, `ngn/k` allows unicode verbs, if you are into that:

The unicode symbols must always be put in parentheses, and you can omit either definition.

Unicode verbs occupy the same class as primitive verbs i.e. they can be used infix in a K expression once they are defined. However, the QWERTY keyboard is where we are working, and the QWERTY keyboard is where we will stay.

Now, we can begin on our Prelude. Go to your `ngn/k` folder (most people install at `$HOME/k`), open up a new K file `prepl.k`, and add the following line:

```
\l full/path/to/prelude.k (bash expansions like ~ do not work.)  
\l full/path/to/repl.k
```

`\l` loads a K file, running the entire thing and importing its data. You have to start the REPL *after* you create all your prelude functions, hence this is the order.

Then, we can create `prelude.k`. This prelude will cover some simple operations that work in a general manner, taking inspiration from APL and BQN. Then,

you can alter these functions, add and remove things from it as per your taste. I recommend having a stab at the functions presented here before you look at the possible approaches.

## 13.1 Math

Absolute value makes a number positive. The easiest way is to make a conditional:

```
abs:{$[x>0;x;-x]}
```

We can remove the conditional with an array:

```
abs:((-x;x)@x>0)
```

This can be simplified further with a maximum:

```
abs:{x|-x}
```

Now, for a simple sign function. Luckily, we already have absolute value:

```
sgn:{x%abs x}
```

Ceiling is the negation of negated floor:

```
ceil:-_ -:
```

What we see here is that the name `ceil` is bigger than its definition. You can simply delete this, and remember `-_-` as `ceil`, cause that removes the need for a word.

In K, most of the time, it is better to a) reuse a sequence of symbols that are shorter than a name b) Use short names for widely reused functions. Some people like to keep things within 1-2 characters max. You don't have to, but it's often useful to understand terse code when working with K, because you will be seeing a lot of it.

**Note:** Many sets of symbols are smaller than their names, but they can be difficult to remember. You can store these as a list of strings, and print them with a help function, say:

```
hstr: ("ceil 1+_"
      "not equal ~="
      "flatten ,//")
help: {`0:hstr}
```

You can then refer to your own personal snippet collection with `help[]`. A new website called `ngnkcarr` has been made for searching up common snippets, so it's easier to find and add the ones that you need.

Not equal to is in the same category as `ceil`. `neq:(~=)` is quite redundant, so it's better to not have it.

We can use the Euclidean GCD algorithm, and LCM can be derived from GCD, so one of them comes for free.

```
gcd:{$[y>0;o[y;y!x];x]}
lcm:{(x*y)%gcd[x;y]}
```

Every recursive algorithm can be converted to an algorithm using iteration. let's try that. A few things can be noted:

1. Iterate as long as y is not zero.
2. maintain y and y!x every iteration.

A simple way to get this with while would be:

```
gcd:{*({y>0}.){y,y!x}.)/x,y}
```

Trigonometric functions can all be derived from ``sin`, which is a special symbol in ngn/k (see the help topic at `\``). If ``sin 0` is giving you `0n`, then make sure that you reinstall ngn/k with `make k-libc`.

Using trigonometric identities, we can derive the following:

```
PI: 3.141592654
sin: `sin
cos: `sin (PI%2)- / sin(90-theta)
tan: {(sin x)%cos x}
```

Finally, the power function is quite simple:

```
pow:{* / y#x}
```

But this function only handles power for positive integer y. In order to get a generalized power function, we need to use some math:

```
pow:{{`exp y*`ln x}}
```

## 13.2 Strings

One thing that K and other array languages tend to struggle with are strings. A few commonly used string functions are given here.

To upper case can be implemented by subtracting 32 from any lowercase alphabet. So really, the first thing we need to be doing is finding out if a character is a lowercase alphabet or not. K allows character comparisons, so this is simple:

```
islower: {(x>"a"-1)&x<"z"+1}
```

what's better is that this works directly on strings, since all its operations conform:

```
islower "ABCDabcd"
0 0 0 0 1 1 1 1
```

If you find a function that needs to be applied to each element of an array, always check if you can make it conform without the `each`. Now, the `toupper` function becomes much easier: just add 32 to each place with a 1.

```
toupr:{x-32*islr x}
```

but this yields a list of numbers:

```
toupr "UPPERcase123"  
85 80 80 69 82 67 65 83 69 49 50 51
```

to remedy this, we need to just cast them back to characters:

```
toupr:{`c$x-32*islr x}
```

Basic string replacement can work using a simple split and join:

```
replace:{z/y\x}
```

### 13.3 Array primitives

There's a lot of things you can do with arrays, and not all of them are convenient to type over and over, so this is a section dedicated to primitives from other array languages, and some general ones.

K only has grades, so having some simple and convenient sorts makes more sense:

```
asc:{x@<x}  
desc:{x@>x}
```

Index of is enough to find out if an element is in an array, but having a direct `in` function is more convenient.

```
in:{~^y?x}
```

Array rotation is a pretty common primitive that was even present in K3, where it used the `!` symbol. Here we define a version with `cut`, but you can also find a version without splitting. Here, negative `x` rotates right, and positive rotates left.

```
rot:{, /! [#y;0,x]_y}
```

We can add our matrix multiplication from the previous chapter:

```
mmul: (+/*)\:
```

Now that we've made a nice prelude for ourselves, we can finally use it!

```
rlwrap path/to/k prepl.k
```

### 13.4 Exercises

1. What primitives would you remove from K, and which ones would you add in their place? A fun thing to note about K is that even after 9 iterations, the official answer to this question hasn't been decided!
2. Make the `sec`, `cosec` and `cot` functions using your Prelude so far.
3. Create an `islower` function and a `tolower` function for your own.

4. Write a string replacement function using splice (?).

## 14 A Sudoku program

Sudoku is great array task for K since it deals with small 2D arrays and it can be solved very nicely with array based methods. In this chapter, we will make a fully featured Sudoku game in K. This chapter will feature longer K snippets, and it will test your ability to read K.

If you do not know what Sudoku is, I suggest reading these articles.

### 14.1 Generating a random board

One way to generate a puzzle is placing random digits in valid locations. The array-oriented method which we will be using, is generating a puzzle from a solved board. Let's begin with a simple definition:

```
sg: (1 2 3 4 5 6 7 8 9 / a pre-solved grid.  
    4 5 6 7 8 9 1 2 3  
    7 8 9 1 2 3 4 5 6  
    2 3 1 5 6 4 8 9 7  
    5 6 4 8 9 7 2 3 1  
    8 9 7 2 3 1 5 6 4  
    3 1 2 6 4 5 9 7 8  
    6 4 5 9 7 8 3 1 2  
    9 7 8 3 1 2 6 4 5)
```

You can use any grid you want as long as it's a valid, completely solved sudoku grid. The algorithm for the board is as follows:

1. Switch the numbers in the grid around. Example: 1 becomes 9, 9 becomes 3, so on.
2. Shuffle the rows in each 3x9 block, and the columns in each 9x3 block.
3. Shuffle the 3x9 row blocks and the 9x3 column blocks.

Let us name this function `gen`.

Before doing any random number generation, we must seed the randomizer with ``prng:`

```
`prng@0
```

This initializes it with the current time.

For the first step, we generate a random permutation of the numbers 1 to 9, and index into them (see chapter 8 for the random function ?):

```
s1:(0,1+0N?9)@sg
```

Since we have to exclude 0, we prepend it to make sure it is never used.

A possible value of `s1` would be:

```
4 7 2 3 8 1 5 9 6
3 8 1 5 9 6 4 7 2
5 9 6 4 7 2 3 8 1
7 2 4 8 1 3 9 6 5
8 1 3 9 6 5 7 2 4
9 6 5 7 2 4 8 1 3
2 4 7 1 3 8 6 5 9
1 3 8 6 5 9 2 4 7
6 5 9 2 4 7 1 3 8
```

For the second step, we have to shuffle individual 3-row and 3-column blocks.

```
4 7 2 | 3 8 1 | 5 9 6
3 8 1 | 5 9 6 | 4 7 2
5 9 6 | 4 7 2 | 3 8 1
7 2 4 | 8 1 3 | 9 6 5
8 1 3 | 9 6 5 | 7 2 4
9 6 5 | 7 2 4 | 8 1 3
2 4 7 | 1 3 8 | 6 5 9
1 3 8 | 6 5 9 | 2 4 7
6 5 9 | 2 4 7 | 1 3 8
```

```
4 7 2 3 8 1 5 9 6
3 8 1 5 9 6 4 7 2
5 9 6 4 7 2 3 8 1
-----
7 2 4 8 1 3 9 6 5
8 1 3 9 6 5 7 2 4
9 6 5 7 2 4 8 1 3
-----
2 4 7 1 3 8 6 5 9
1 3 8 6 5 9 2 4 7
6 5 9 2 4 7 1 3 8
```

The following line does both the shuffles at once, transposing the array twice.

```
s2: 2{+x@,/0N?'3 3#!9}/s1
```

You can try deriving the third step from the above snippet. If you're having trouble, you can reveal the spoiler here:

```
2{+x@,/(!3)+/:3*-3?3}/s2
```

The final function looks like this:

```
gen: { / generate a random sudoku board.
      / https://stackoverflow.com/a/61442050/4568534
      s1: (0,1+-9?9)@sg / shuffle the numbers around from the grid.
      s2: 2{+x@,/0 3 6+-3?'3#3}/s1 / shuffle rows in each 3x9 block, cols in each 9x3 block
```

```
2{+x@,/(!3)+/:3*-3?3}/s2}      / shuffle 3x9 row blocks, 9x3 col blocks
```

## 14.2 Displaying a Sudoku puzzle

A sudoku puzzle has lines akin to a tic tac toe puzzle. The default K prettyprint is nice, but having the lines like in the newspaper is nicer.

There are many different approaches you can use to make this. Here, I will be modifying a character array to get the desired result. First, we add the horizontal lines using the splice form ?:

```
s:(" "/"$x)?[;;,17#" -"]/6 3
```

This may be confusing, but it is simpler if you realize that this is just a fold with a 2-element projection. In `x f/y`,

```
x: " "/"$x
f: ?[;;,17#" -"]
y: 6 3
```

Then, we add some vertical lines (and their intersections) with the help of a column projection:

```
s[;5 11]:2#'11#"|||+"
```

neat and concise, and a good place to use modification.

## 14.3 Solving a Sudoku puzzle

A beginner programming course would usually use a recursive algorithm to solve a sudoku puzzle. Even though there are multiple ways to recursively solve sudoku in K, we will be looking at an iterative method.

The solver we will write now is a translation of Roger Hui's Sudoku solver, originally written in J, then translated to K. The basic algorithm behind Roger Hui's sudoku solver is to

1. Take each board given.
2. Take an empty spot on the board and place all possible valid numbers.
3. If there are no more moves possible, remove it. If it is solved or has possible moves, keep it.

There are some test puzzles in `code/sudoku.k` that you can use to understand the result of the steps below.

First, we define a few simple helpers:

```
j:      (,/0 27 54+\":0 3 6)+\":,/0 9 18+\":0 1 2 / helper for c and r
r:      9#',: '(9*!9)+\":!9 / row for each cell.
c:      ((!9)+\":9*!9)@9#,!9 / col for each cell.
b:      j@(/9# '!9)@j / block for each cell.
I:      r, 'c, 'b / all cells to check for each cell.
```

```
R:      j,((9*!9)+\:!9),(!9)+\:9*!9      / row, col, block indices.
in:      {~^y?x}                          / is x in y?
```

You can try peeking at their values to understand what they mean.

### 14.3.1 Finding valid moves

First, we need a function to find valid moves for each cell of a board. We will call this `free`.

```
free:    {(0=x)>(1+!9)in/:/:(,/x)@I}
```

What `free` does is take a board and mark which numbers can be placed at which positions.

`(1+!9)in/:/:(,/x)@I` checks if each of 1-9 are present in the row, column, block for each cell.

The check `(0=x)>` makes sure that numbers can never be placed at non-empty positions. What it finally returns is a 9x9x9 boolean array where the result at `[i;j;k]` is 1 if `1+k` can be placed at `[i;j]`. If we do `1+&:''` on the resultant array, we can find out which numbers we can place at each location.

### 14.3.2 Validating a board

To validate a board, we need to check each region and make sure that all is getting the regions is simple: we just use the helper `R`:

```
regions:{(/x)@R}
```

This returns a list of 9-element lists which contain each region. Next we can check each region with a cool idiom: `x?x`. What `x?x` helps with is checking for duplicates. If all elements in `x` are unique, `(x?x)~!#x`. Hence, we can check each row with `!#x` (which is `!9` in this case) to find out which elements are out of place.

For each row we now do

```
{(!9)=x?x}'
```

This expression however will mark empty spots as wrong, so we need to rectify that:

```
{(0=x)|{(!9)=x?x}'x}
```

Then we need to make sure that all numbers are in 0-9 and we have a function that checks individual cells:

```
chk: {(x in !10)&(0=x)|{(!9)=x?x}'x}
```

Finally, we can use it to validate an entire board:

```
ok: {&/,/chk regions x}
```



### 14.3.3 Step 1: Filling in the right places

The simplest case in sudoku is when a single cell has a single candidate number. The other case is multiple candidates. We will handle these separately. To handle these cases, we don't need the entire grid, but the numbers that can be placed in its cells, the result of applying **free** to the grid.

**14.3.3.1 For a single candidate number** First, we find free cells which sum to 1:

 $\{1=+/' 'x\}$ 

Then we mark each cell by its respective number:

(+/' 'x\*\:\:1+!9)

+/' here can also be replaced with |/. What matters is that we are only handling single numbers, so summing cells with multiple possibilities will be zeroed out in the next step, where we multiply the two:

$$s1x: \{ (+/' 'x* \backslash : \backslash : 1+!9)*1=+/' 'x \}$$

**14.3.3.2 For multiple candidate numbers** Here, what we will look at is whether a candidate number occurs exactly once in a row or column, and assign that number to its respective cell.

First, among all groups, we find the ones that have one element free. We can do this as follows:

$$m: 1 = + / ' ( , / x ) @ R ; \quad j : \& | / ' m$$

Once we have the groups, we can then find the groups that have only one free value (indices).

j:&amp;|/'m

This can then be immediately used to locate the first 1 in those respective groups.

k:m[j]?\:1b

Using these locations, we can now find them in the actual grid:

$$i:,/(R@j)@'\&: '(+:(, /x)@R@j)@'k$$

Finally, we fill these new elements into the grid and return it.

9 9#@[&81;i;;;1+k]

```

s1u: {m:1=+/'r:regions@x      / (27x9 array) boolean array of values which
                                /   are only free in one element of a group per group
                                /
                                /   indices of groups which have at least one value which is
j:&|/'m                        /   for such groups, the first value which is free only once
k:m[j]?\:1b                   /   find the positions of the first element of each group
i:/ (R@j)@'&:'(+:'r@j)@'k     /   which is only free once in that group

```

```
9 9#@[&81;i;;;1+k]} / make a matrix of these filled in
```

#### 14.3.4 Step 2: Filling all possible candidates

The previous step tries to input "forced" moves, where there's only one possibility to fill a single cell. Now we have to fill other cells with all possible outcomes.

First, if there are no zeroes in the grid, we can just return it as is.

```
$[~in[0;,/x];:,x;::]
```

Here, we use `::` because `$` always requires two branches. `::` is a null and equivalent to an empty statement, so you can use it to mark empty statements.

Now, operating on the free list of the puzzle, we can find a single cell with a minimum number of possibilities.

```
b:free x; i:t?&/t:/ (10,1_!9)@+/'b
```

Then, each possibility for that minimum cell is taken and inserted into the grid.

```
d:1+&(,/b)@i; x+/:d*\:i=9 9#!81
```

Here's the filling function. Since we have multiple possible grids that can approach the answer, `s2` is defined below.

```
s2a:{$[~in[0;,/x];:,x;:]; b:free x; i:t?&/t:/ (10,1_!9)@+/'b
      d:1+&(,/b)@i; x+/:d*\:i=9 9#!81}
s2: {,/s2a'x}
```

#### 14.3.5 Combining all the steps

Our final step is to apply all the above steps till we reach a solution. This is easy with `fixpoint (/)`. The solution is simple: Apply step 1, filter out invalids, apply step 2.

```
sudoku:{{s2 (ok')#s1 x}/,x}
```

### 14.4 Creating a valid puzzle

A sudoku puzzle needs to satisfy the following requirements:

1. A puzzle may have only one valid solution.
2. A puzzle must have digits placed in valid squares.

We can add `n` empty spaces to a board like so:

```
nz:{b:/,x;9 9#@[b;y?&-0=b;:;0]}
```

To check whether a puzzle is valid, all we need to check is if `1=#sudoku x`.

To generate a random board and test this, you can use `gen[]`.

```
gp:{{~1=#sudoku x}{z;nz[x;y]}[x;y]/9 9#9}
```

Note that the least possible number of hints a sudoku puzzle can have is 17, and many sudoku puzzles are not reducible to 17 hints. Use `gp` very carefully.

## 14.5 Playing the game

Now that we have all the important parts done, we can play the game.

```
`1:"Sudoku Puzzle\n"
```

First, we ask the player for a number of empty spaces (for difficulty). If it is not numeric or not less than or equal to 64, we should prompt again.

```
zs:{~((10/x-"0")<65)&(&/x>"0"-1)& &/x<"9"+1){`1:"Enter number of empty spaces:":-1_1:~}/"65"
```

1: returns a trailing newline with its input, so we remove that. We use "65" to start the loop because while is entry controlled.

Finally, we can set up a game loop:

- The while condition checks if the game is won.
- The user can end the game at any time, and a flag is set to check for a win or a loss.
- Coordinates are validated using *outdexing* i.e. if an array is indexed out of bounds, it returns a null value.
- Finally, the puzzle is amended at the given coordinates using the special form for Deep Amend.

```
brd:gen[]
win:1
fin:{~x~brd}{
  `1:"Board:\n"
  `0:fmt x
  `1:"End Game? (y/n)"
  $["y\n"~1:0;[win::0;:brd];:]
  cd:{~0=x . y}[x]{`1:"Enter coordinates to place at: "; 10/'(" \-1_1:0)-"0"}/ON ON
  n:{^(1+!9)?x}{`1:"Enter number to place (1-9): ";10/(-1_1:0)-"0"}/10
  .[x;cd;:;n]}/gp[brd;zs];

`1:"Final Board:\n"
`0:fmt fin
`1:$[win;"You Win!\n";""]
`1:"Game Over\n"
```

Enjoy your sudoku game!

## 14.6 Exercises

- We can place numbers on the board just fine, but we can't modify the numbers we have previously placed. You can try:
  - An undo button

- Input that allows you to modify any square that was previously empty
- Currently, this solver uses some flattening (`,/`) and reshaping `a#x` in order to solve a puzzle. These idioms are convenient, but flat arrays are much more efficient. Try to optimize the program to account for flat arrays.

## 15 Tables and Libraries

ngn/k has a hidden, somewhat secret data structure: the table. Tables are a prominent presence in k7 and k9, the latest versions of K, so ngn/k supports them for that reason.

### 15.1 Tables

A table uses symbols to mark columns, and integer indices to mark rows. Making a table is simple. The first method is flipping a dictionary1:

```
:t:+`a`b!(1 2;3 4)
+![ `a `b
+(( 1; 3)
 ( 2; 4))]
```

The display is a bit strange because ngn/k as described earlier, always returns valid K data. However, as long as you have more than one column in your table, the columns will be aligned correctly.

The other way to make a table is with a list of dictionaries:

```
:t:(`a`b!1 2;`a`b!3 4)
+![ `a `b
+(( 1; 2)
 ( 3; 4))]
```

The key names of every element must be the same, and they must be in the same order. This form is good if you are constructing tables programmatically, so `(,`a`b!1 2),`a`b!3 4` would be a different way to come up with the same table.

Creating a K table requires rectangular data: all rows must be of the same length, and all columns must be of the same length.

Tables can be indexed into in the same ways as other datatypes, described in Chapter 4. However, tables are special in that you can index into them using column labels *and* row labels, without the need of a projection. Using `t` from the previous example:

```
t[1]
!/+((`a;3)
 (`b;4))
t[`b]
```

2 4

Here, retrieving a single row returns a dictionary, and a single column is a simple list. Retrieving multiple rows and columns results in a table.

ngn/k does not come with a query language for tables, but there are equivalent ways to access data. Usually a single SQL query breaks down into multiple K primitives, abstracting away all the dirty work. In the Kx Systems implementation, the Q language has such syntax if you'd like to try it out.

# (filter) filters rows, like an SQL `SELECT ... WHERE` query. The same goes for `-`.

```
/ SELECT * FROM t WHERE a < 2;
{x[`a]<2}#t
+`a`b!(,1;,3)
```

Updates to a table have to be done with indexing. An SQL group by query had to be done with `=`, which returns a dictionary that has to be reconstituted into a table.

```
t:+`a`b!(!10;10?10)
,/t@=2!t@`a
```

## 15.2 File Model

ngn/k's file I/O is based on the UNIX model of file management. It is best to read up on the mechanics of the model using the UNIX manual pages or other reliable document before using K for file I/O.

As you may remember from Chapter 10, we can use `0:` (line-oriented) and `1:` (byte-oriented) to read from and write to files. This is easy with a string denoting a file name, but you can also open a file descriptor to a file with `<`.

### 15.2.1 <x Open

**Symbol:** `<`

**Args:** `< symbol`

**Description:** Return a file descriptor that points to the file at location `x`.

### 15.2.2 >x Close

**Symbol:** `>`

**Args:** `> integer`

**Description:** Close the file descriptor with id `x`.

File descriptors can be used in place of file names for `0:`. You can also use file descriptors that were not opened by your program. A simple example is opening

stdin on linux, which is usually open at file descriptor 0. So this is another way to prompt for input:

```
0:0
123
,"123"
```

### 15.3 Libraries and Namespaces

A somewhat common approach in a language like K is to store some forms of processed data in a file for later analysis (similar to prolog/datalog, the data format is consistent and easy to read). In any K file or the REPL you can do `\l data.k` to reuse the data from a file, or simply use the file like a library.

Sometimes however, your code and your data files may have some naming conflicts, and this is where `\d` comes in. One easy way to use `\d` is as an easy way to make large namespaces. Once `\d ns` is used for example, all global variables defined after that are prefixed with `ns..`

```
\d ns
a:1
ns
'value
ns
^

ns.a
1
```

Note that `ns` is not a variable you can inspect, it is just a prefix for variables in your namespace. You can freely use any variables defined in `ns` as long as you don't navigate to a different one with `\d`, like `\d .`, which goes back to the global namespace.

```
\d .
a
'value
a
^

i
ns.a
1
```

A common pattern for importing libraries is hence:

```
\d l
\l lib.k
\d .
```

You can load as many libraries as you like into a namespace, allowing some useful grouping strategies.

Finally, to "index" into namespaces, you can use `..`. This builtin is mostly for convenience.

### 15.3.1 `..x` Get

**Symbol:** `..`

**Args:** `.. list_of_symbols`

**Description:** Index into the given namespaces. `..`a`b`c` will return the value of `a.b.c`. Invalid routes will throw an error.

## 15.4 Serialization

The other way of serializing K data, apart from the default prettyprinter ``k` is in the JSON format. K's data model generally maps well to JSON. You can convert any object to JSON with the help of ``j`.

```
`j@`a`b`cd!(1;"a";`sym)
"{\"a\":1,\"b\": \"a\", \"cd\": \"sym\"}"
```

These special functions like ``j` and ``k` are symbols that behave specially when called on data, and are special to the ngn/k implementation.

1: Internally this is not a full transposition of a dictionary. It just changes how the data is seen by the ngn/k interpreter.

## 16 The ngn/k C API

This chapter will require a working knowledge of C, and it will make use of C macros that are aimed at writing very terse code.

It is often the case that you will find a gap in one of you favourite languages that K fills (and vice versa). This has led to programmers like phantomics reimplementing APL for use as an embeddable language. Commercial K implementations also include a number of methods to do so.

ngn/k uses a unified C API that was defined in ktye's K implementation. The ngn/k specification of the API header file is here: `k.h`. ktye's API is well documented, and if you are experienced with C interfaces, it is recommended that you skip this chapter and use the time to simply read the docs. You can also see `onikuruma` for an example of a C binding to a popular library. This chapter simply describes in a more linear, friendly manner how to use the API.

## 16.1 Extending ngn/k with the C Foreign Function Interface

To use C functions in K, you need to do the following:

- Modify the ngn/k makefile and build `libk.so`
- Write glue code for your K program using `k.h`
- Compile the glue code into a dynamic linked library (in this case, a `.so` file)
- Import the functions you require from the library using the 2: I/O verb, and use them!
- Pray to your favourite deity that there are no segmentation faults (optional)

A simple example for this is given with the implementation at `x/extend`.

## 16.2 Building `libk.so`

First, run `make c` in your ngn/k installation to avoid any strange build errors. You need to modify the ngn/k makefile on line 4, changing the `L` variable to `L='-lm -ldl -Wl,-E'` (ignore the comment).

Then, run `CC=clang make k libk.so` (using clang is important). This should give you a `libk.so` file.

## 16.3 C API Datatypes

ngn/k data is represented by the type `K` in C (which is secretly a `long long`). All function arguments must be of type `K`.

Assuming you know what is contained in a variable of type `K`, you can convert it to C data of the appropriate form. `cK(K)` for example, converts a `K` value into a C `char`.

`sK(K)` does not exist however, so you have to use the K execution function `Kx(char*, args)` which lets you execute an arbitrary K function on the required arguments. `Kx("$", x)` will cast a symbol of type `K` to a character list of type `K`, which you can then read with the functions in the next section.

For converting a list, you have to first query:

- `NK(K)` to query length of the list
- `TK(K)` to get type of the list you need to allocate the required amount of memory using these values, and then use a conversion function, like `CK(char*, K)`, which reads a `K` value into a C character array.

A basic rule is that functions that start with a lowercase letter convert atoms, uppercase converts arrays.

Once you have processed the converted C data, you have to then return a `K` value from your function. This is done with functions like `Kc(char)`, `KC(char*)`,



`size_t`) and so on. They have a similar naming scheme. These functions return a value of type `K` and need no special preparation.

## 16.4 Using functions from libgmp

We are using `libgmp`, since it provides a utility that most `K` implementations do not. Make sure to get your installation files from <https://gmplib.org/#DOWNLOAD> and follow the instructions given with the download. Make sure you have it on your `C` include and library paths.

The base `C` file will just contain a couple of functions and include the two main headers: `gmp.h` and `k.h`. We will first write a makefile to compile it correctly:

**File: Makefile**

```
K_PATH=/home/razetime/code/k
run: libadd.so
    LD_LIBRARY_PATH=$(LD_LIBRARY_PATH):$(K_PATH) $(K_PATH)/k add.k
libadd.so: add.c
    $(CC) -I$(K_PATH) -L$(K_PATH) -shared -fPIC $^ -o $@ -lk -lgmp
```

make sure to change the `K_PATH` variable on line 1 to the correct location on your device.

In `C` we will take large integers as strings and add them together. We are using the macros from `k.h` here for the sake of consistency. First, we will start by converting `K` strings into `C` strings, and return the first argument for now.

**File: add.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <k.h>
K add(K a,K b){
    N al=NK(a),bl=NK(b);
    C* ac=calloc(al+1,sizeof(C));C* bc=calloc(bl+1,sizeof(C));
    CK(ac,a);CK(bc,b);
    printf("a: '%s'\nb: '%s'\n",ac,bc);
    return KC(ac,al);
}
```

In `K`, we will simply import the function, specifying number of args, and call it.

**File: add.k**

```
add:`"./libadd.so"2:(`add;2)
add["123";"345"]
```

Running `make run` now should show the correct values of `a` and `b`.

Using the extracted data in GMP is simple. We initialize our operands and result variables, and then use the Integer Functions as per the documentation. The result will be an `mpz_t`. This can be converted to a C string, then converted to a K string. So the last few lines of `add()` become:

```
mpz_t big_a, big_b, r;
mpz_init_set_str(big_a, ac, 10); mpz_init_set_str(big_b, bc, 10); mpz_init(r);
mpz_add(r, big_a, big_b);
return KC(mpz_get_str(NULL, 10, r), a1);
```

`make run` will now give you the correct sum: "468". Feel free to experiment with bigger numbers, and enjoy your new K powers!