SOFTWARE IS A CONSTANT BATTLE AGAINST COMPLEXITY.

ERIC EVANS, *DOMAIN-DRIVEN DESIGN*

I CAN'T UNDERSTAND WHY PEOPLE ARE FRIGHTENED OF NEW IDEAS.
I'M FRIGHTENED OF THE OLD ONES.

JOHN CAGE

YES.

ARTHUR WHITNEY

STEPHEN TAYLOR

POST ATOMIC

LAMBENT TECHNOLOGY

PUBLISHED BY LAMBENT TECHNOLOGY

`http://lambenttechnology.com`

First printing, March 2023

Contents

Dedicated to Kenneth E. Iverson, and

laconic Norwegians everywhere.

Part I

4u?

This book is for you if you have already learned some of the q programming language and would like to become fluent.

Improving your skill with a language means engaging with other speakers. This is as true of programming languages as it is of natural languages.

Among people who speak French, my French (such as it is) improves steadily. This is how you and I learned our native languages. Learning this way comes naturally to us, with little effort.

My French is also improved by watching French films and TV. I hear and remember useful ways to say things.

Finally, my French improves *really* fast when I write – provided someone corrects my work.

All this applies to you in learning q.

If you are lucky enough to work with others writing q you will gradually learn what they know. You will learn to read and write q much as they do. This will be little effort, and you will all read each other's code easily.

But there is a catch.

You will learn no more than they know. And it might appear to be all that there is to know.

I would be in the same situation if I spoke French only with other English people who learned it at school. We would all speak school French to each other.

Happily there is great French TV drama on streaming services, newspapers and magazines to read online, and I can get as many novels in French as I can read. I'm all set. I can read and hear a wide range of French.

With you and q things are more limited.

First, not many people write q. And most of us are bunched together in capital-markets technology. If you are not already working among us, you're unlikely to bump into us.

Second, (whisper it) most of us write school-level q. [1]

[1] The historical explanation for this is in Appendix A.

The point here is that very few people write fluent q, and most of the rest of us find their code so surprising (at best) or baffling (at worst) that we long ago dubbed them "the q gods". And so the first q textbook got its consoling title *Q for Mortals*.

I am not deprecating Jeff Borror's excellent textbook, nor Nick Psaris' excellent *Q Tips*. In fact, if you have read neither, you should probably put this book down and read one, because I am going to assume you know at least that much.

This book is not an introduction to q.

Prometheus is a hero from Greek myth. He brought down fire from heaven for humans to use.[2] This book is a Promethean project. Fluent q is not reserved to a race of divine beings. Like any language, it yields to study and practice. Go for it.

[2] Prometheus got punished for it. I'm hoping to avoid that.

LASTLY, YOU MIGHT HAVE COME HERE WITH ANOTHER AGENDA. If you are curious about programming languages and want to know what is special about q, the next chapter is a high-level summary.

yq?

Besides access to kdb's performance, is there any reason to prefer q to more widely-used programming languages?

Some simple examples may help you to decide whether you would like to code in q.

Hello world

The classic intro program[3] in C prints *hello, world* in the command shell.

[3] *The C Programming Language,* Brian M. Kernighan & Dennis M. Ritchie, 1978

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

In q:

```
q)"hello, world""hello, world"
```

The default behaviour of the q REPL[4] is to evaluate the line and display the result.

[4] read-evaluate-print loop

Pascal's Triangle

A Python program to print seven rows of Pascal's Triangle.

```
# Set the number of rows for Pascal's triangle
num_rows = 7
# Initialize the first row
row = [1]
# Print the first row
print(row)
# Generate the remaining rws
for i in range(1, num_rows):
    # Initialize the row with the first element
    new_row = [1]
    # Calculate te values for the rest of the row
    for j in range(1, i): new_row.append(row[j-1] + row[j])
    # Add the last element to the row
    new_row.append(1)
    # Print the row
    print(new_row)
    # Set the current row as the previous row for the next iteration
    row = new_row
```

In q: `6{(+)prior x,0}\1`

```
q)6{(+)prior x,0}\1
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

The lambda appends a zero to its argument (`x,0`) then adds each item of it to the preceding item (`(+)prior`). It does this six more times (`6{...}\`), returning the result of each iteration.

Fibonacci Series

A C program to print the first ten numbers of the Fibonacci Series.

```
#include <stdio.h>

int main() {
    int i, n = 10, t1 = 0, t2 = 1, nextTerm;
    printf("Fibonacci Series: ");
    for (i = 1; i <= n; ++i) {
        printf("%d, ", t1);
        nextTerm = t1 + t2;
        t1 = t2;        t2 = nextTerm;
    }
    return 0;
}
```

In q: `8{x,sum -2#x}/0 1`

```
q)8{x,sum -2#x}/0 1
0 1 1 2 3 5 8 13 21 34
```

Above, the lambda appends to its argument (`x,`) the sum of its last two items (`sum -2#x`). The Over iterator (`8{...}/`) applies that to the seed `0 1` eight times and returns the final result.

Fahrenheit-Celsius

Convert degrees Fahrenheit to Celsius.[5]

[5] from Kernighan & Ritchie, *op. cit.*

```
#include <stdio.h>

#define   LOWER   0      /* lower limit of table */
#define   UPPER   300    /* upper limit */
#define   STEP    20     /* step siz */

/* print Fahrenheit-Celsius table */main()
{
    int fahr;
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
```

```
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

In q:

```
q)flip 1{{x*`long$y%x}[.1](5%9)*x-32}\20*til 16
0   -17.8
20  -6.7
40  4.4
60  15.6
80  26.7
100 37.8
120 48.9
140 60f
160 71.1
180 82.2
200 93.3
220 104.4
240 115.6
260 126.7
280 137.8
300 148.9
```

Analysing the q:

```
q)20*til 16
0 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300
q){(5%9)*x-32}20*til 16  / F to C
-17.77778 -6.666667 4.444444 15.55556 26.66667 37.77778 48.88889 60 71.11111 ..
```

Above, the lambda `{(5%9)*x-32}` concisely expresses the conversion from Fahrenheit to Centigrade. (Iteration through the items of `x` is implicit.)

Round to one decimal place: divide `y` by the rounding interval `x` (0.1), cast to long, multiply by `x`: `{x*`long$y%x}[.1]`.

The Scan iterator `1{...}\` returns the results of applying the lambda zero and one times[6]; that is, the lambda's argument and its result, a list of two vectors; i.e. a two-row matrix. The `flip` keyword flips it into a two-column table.

[6] A form sometimes called 'the Zen monks':
— *How many Zen monks does it take to change a lightbulb?*
— *Two: one to change it, one not to change it.*

Code golfers and typists with sore fingers will relish the brevity. But the point of these comparisons is not that qbists do less typing.

The semantics of the q code is all about the algorithm, uncluttered by loop counters. And its brevity dramatically expands the range and complexity of code that can be typed and explored in the REPL, outside the edit-load-compile-run cycle.

As Arthur Whitney, designer of k and q, put it:[7]

[7] Keynote address to the British APL Association conference at the Royal Society, London, 2004

> It is theoretically impossible for a k program to outperform hand-coded C, because k compiles into C. For every k program there is an equivalent C program that runs exactly as fast. Yet k programs routinely outperform hand-coded C. How is this possible? Because it is easier to find your errors in four lines of k than in four hundred lines of C.

A whirlwind tour of q

THIS BOOK is for qbies who want to raise their understanding of q and skills with it – to become qbists.

But perhaps you picked up this book just to satisfy your curiosity about the language. You've heard something about it. What makes it so special?

So this is a whirlwind tour of the language. It is not intended as a tutorial, though if you already know several quite different programming languages, it might be all you need.

Q is an interpreted language, like Python and JavaScript. It presents a REPL[8] and a prompt `q)`. It evaluates the expression you type and displays the result.

[8] Read-evaluate-print loop

```
q)2+2       / who knew?
4
q)til 5     / gimme five
0 1 2 3 4
q)5#2 3     / take five
2 3 2 3 2
```

Data types

Q is designed for speed with very large data sets. Loose datatyping would compromise efficiency, as primitives silently cast from one type to another. Automatic type conversion is minimised by fine-grained data types.

An atom is the smallest unit of data. An atom cannot be indexed. There are 18 types of data atoms.

```
boolean   0b
guid
byte      0x00
short     0h
int       0i
long      0j, 0
real      0e
float     0.0, 0f
char      " "
symbol    `
timestamp 2023.03.16D11:51:26.923000000
month     2000.01m
date      2000.01.01
datetime  2023.03.16T11:52:07.320
timespan  00:00:00.000000000
minute    00:00
second    00:00:00
time      00:00:00.000
```

There is no String datatype. What qbists call a *string* is a vector of characters. (See Data Structures below.)

A symbol atom is a backtick ` followed by zero or more characters. It is used to enumerate repeated strings such as stock codes.

```
`goog   / Google
`ibm    / IBM
`msft   / Microsoft
```

A symbol atom displays as text, but the underlying data is an integer index into a master list of strings called the *symlist*.

The type keyword returns the type of its argument as a short. A negative sign indicates an atom.

```
q)type each (3;3.14159;"q";`q;2023.03m)
```

```
-7 -9 -10 -11 -13h
```

The Cast operator casts data between types. Its left argument is the target datatype, as a short, char or symbol from the table above.

```
q)9h$999
999f
q)"j"$3.14159
3
q)`byte$"q"
0x71
```

The `string` keyword returns a string representation of an atom.

```
q)"c"$3.14159
"\003"
q)string 3.14159
"3.14159"
q)string `ibm
"ibm"
q)string 2000.01.01
"2000.01.01"
```

Temporal data

Dot notation is a convenience for temporal data.

```
q)show now:.z.d+.z.t
2023.03.16D12:15:19.562000000
q)(now.minute;now.second;now.month)
12:15
12:15:19
2023.03m
```

Functions are first-class objects

Functions are first-class objects in q, so they too are atoms.

Functions include operators, keywords, projections, compositions, and lambdas. Each has its own datatype.

```
q)type each (+;2*;type;+/;{x*y+z})
102 104 101 107 100h
q)/operator;projection;keyword;derived function;lambda
```

Data structures

Atoms can be arranged in data structures. There are two types, lists and dictionaries.

Arguably, dictionaries are a more general form of list in which you control the index.

General lists

A list is zero or more items separated by semicolons and embraced by parentheses. A list item is an atom, list or dictionary.

Some lists:

```
(1;2;3;4)
(`ibm;`msft;`aapl)
("Kenneth";"Iverson";1920.12.17;`Canada;(`Harvard;`IBM;`IPSA))
(+;-)
(sum;avg)
()        / general empty list
```

List notation does not describe a list with one item.

```
q)type(3)  / atom
-7h
```

You can make a one-item vector with the Take operator.

```
q)type 1#3
7h
```

You can make a one-item list by appending the item to the general empty list.

```
q)type (),`ibm
```

```
11h
```

Or you can use the enlist keyword.

```
q)type enlist[`ibm]
11h
```

Vector literals

A *vector* is a list in which all the items are data atoms of the same type. Vectors and the efficiency of vector operations sit at the heart of q.

Q syntax permits vectors to be written as literals. For example:

```
100011110001b                           / boolean
0x03040506                              / byte
3 4 5 6h                                / short
3 4 5 6i                                / int
3 4 5 6                                 / long
3 4 5 6e                                / real
3 4 5 6f                                / float
3 4 5. 6                                / float
"3456"                                  / char
`three`four`five`six                    / symbol
2000.03 2000.04 2000.05 2000.06m        / month
00:00:03 00:00:04 00:00:05 00:00:06 / seconds
```

A vector has the datatype of its items. The type keyword returns this as a positive short.

```
q)type 3 4 5 6
7h
q)type `quick`brown`fox
11h
```

The type of a general list is 0h. Only data atoms form vectors.

```
q)type ("Kenneth";"Iverson";1920.12.17;`Canada;(`Harvard;`IBM;`IPSA))
0h
q)type (+;-)
```

```
0h
```

Dictionaries

A dictionary is a pair of same-length lists: a list of keys and a list of values.

Key lists are often symbol vectors, but the structure is perfectly general: a dictionary can be made of any two lists of the same length, using the Dict `!` operator.

```
q)show d:`ATW`KEI`SJT!("Whitney";"Iverson";"Taylor")
ATW| "Whitney"
KEI| "Iverson"
SJT| "Taylor"

q)type d  / a dictionary has type 99
99h
```

Tables

A table is a list of named same-length lists.

```
q)species:`cow`duck`python`snail
q)genus:`mammal`bird`reptile`mollusc
q)feet:4 2 0 1

q)show a:([]species;genus;feet)
species genus   feet
--------------------
cow     mammal  4
duck    bird    2
python  reptile 0
snail   mollusc 1
```

It is *also* a list of like[9] dictionaries.

[9] *Like dictionaries* have identical keys. (Order matters.)

```
q)c:`species`genus`feet!(`cow;`mammal;4)
q)d:`species`genus`feet!(`duck;`bird;2)
```

```
q)p:`species`genus`feet!(`python;`reptile;0)
q)s:`species`genus`feet!(`snail;`mollusc;1)

q)show b:(c;d;p;s)
species genus   feet
--------------------
cow     mammal  4
duck    bird    2
python  reptile 0
snail   mollusc 1

q)a~b  / does a match b?
1b
q)type a
98h
```

These two views of a dictionary are equally valid.

Tables are first-class objects in q.

Indexing

The indices of a list are origin-zero ordinals.

```
q)`cow`duck`python`snail [3 1]
`snail`duck
```

Indexing is *right atomic*: the result of indexing a list mirrors the index expression.

```
q)" #"[(1111b;1001b;1111b;1001b;1001b)]
"####"
"#  #"
"####"
"#  #"
"#  #"
```

The indices of a dictionary are its key.

```
q)d [`SJT`ATW]
"Taylor"
```

```
"Whitney"
```

A matrix is a list of same-length lists of uniform type.

```
q)show m:3 5#15?100
12 10 1  90 73
90 43 90 84 63
93 54 38 97 88
```

Its items are rows.

```
q)m[2 1]
93 54 38 97 88
90 43 90 84 63
```

A second index indexes the columns

```
q)m[2 1;2]
38 90
```

Omitting an index selects all values.

```
q)m[2;]
93 54 38 97 88
q)m[;]
12 10 1  90 73
90 43 90 84 63
93 54 38 97 88
```

Indexing is *functional*; the brackets above are syntactic sugar. See Apply/Index below for more on this.

Applying a function

A function is an operator, keyword, composition, projection or lambda. It may have from zero to eight arguments; the number of arguments it takes is known in q as its *rank*.

Many q tokens are overloaded by rank; that is, they denote different operators according to the number of arguments passed. Such functions are called *variadic*.

Bracket syntax

You can always apply a function to an *argument list*.

Like a general list, an argument list is zero or more items separated by semicolons, embraced not by parentheses but by brackets.

```
q)*[2;3]
6
q),[`ibm;`goog`msft]
`ibm`goog`msft
q)count["quick"]
5
q)ssr["quick brown fox";"br?wn";"brave"]
"quick brave fox"
```

Infix syntax

Binary operators and keywords can also be applied infix. This is good q style.

```
q)2*3
6
q)`ibm,`goog`msft
`ibm`goog`msft
```

Republic of functions

The right argument of a function applied infix is the result of evaluating everything to its right.

There is no precedence hierarchy among functions.

```
q)2*3+100
206
```

Every programming language follows this rule[10] when assigning a value to a name; the value assigned is everything to the right of the

[10] I am grateful to Arthur Whitney for this observation.

assignment token. Like other Iversonian languages (APL, J, k) q simply follows this practice consistently.

You can override this with parentheses; but good q style prefers recasting an expression to avoid parentheses.

```
q)(2*3)+100
106
q)100+2*3
106
```

Apply and Apply At

Function application has a functional form: the Apply operator. (Brackets are syntactic sugar.) The Apply operator, written as a dot, applies a function to a list of its arguments.

```
q).[*;2 3]
6
q).[ssr;("quick brown fox";"br?wn";"brave")]
"quick brave fox"
q).[count;enlist["quick"]]
5
```

Apply is a binary operator, and can be applied infix.

```
q)(*) . 2 3
6
q)ssr . ("quick brown fox";"br?wn";"brave")
"quick brave fox"
q)count . enlist["quick"]
5
```

Enlisting the one argument of a unary keyword such as count is tedious, so the Apply At operator is more syntactic sugar: it applies a unary function to its argument.

```
q)count @ "quick"
5
```

Prefix syntax

Prefix syntax lets us elide the infix use of Apply At.

```
q)count "quick"
5
```

This is a considerable blessing. Many important transformations can be expressed as a train of functions, each operating on the result of the next. In common mathematical notation one writes the result of applying function f to the result of function g applied to the result of function h applied to x simply as $fgh(x)$. In most programming languages you can write this as `f[g[h[x]]]`. In q you can write it as `f g h x`.

Good q style

Every programming language lets you assign a name to a value so you can refer to it without computing it again. The value assigned to the name is everything to the right of the assignment token.

Most languages perpetuate the unfortunate choice of = as the assignment token, overriding its familiar meaning.

Q extends this convention systematically. In infix and prefix syntax the right argument is the value of everything to the right.

You can of course interrupt with parentheses and brackets, grouping expressions for evaluation. But reading is interrupted as well, contrary to the principle of notation as a tool of thought.[11]

[11] See Kenneth E. Iverson, "Notation As A Tool Of Thought", ACM Turing Award lecture, 1979

In the best q style evaluation of an expression begins on the right and proceeds smoothly to a result or an assignment on the left.

Index and Index At

A foundational insight of q is the isomorphism of arrays and functions. An array is a function of its indices.

```
q)sqrs:0 1 4 9 16 25 36 49 64 81  / array
```

```
q)sqr:{x*x}                       / function

q)sqrs 5 2 9                      / array
25 4 81
q)sqr 5 2 9                       / function
25 4 81
```

This isomorphism is reflected in q syntax.

```
q)sqrs[5 2 9]                     / array
25 4 81
q)sqr[5 2 9]                      / function
25 4 81

q)sqrs@5 2 9                      / array
25 4 81
q)sqr@5 2 9                       / function
25 4 81
```

Above we see that the syntax of Apply At and Index At are the same. And they are both denoted by `@`.

Similarly for Apply and Index, both denoted by a dot.

```
q)show M:til[3]*/:\:til 5
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8

q)M . 2 3                          / Index
6
q)(*) . 2 3                        / Apply
6
```

Implicit iteration

Map iteration is implicit in most operators. The operator applies between corresponding elements of two lists.

```
q)2 3 4+5 6 7
7 9 11
```

An atom maps to a list of any length: *scalar extension* pairs the atom with each element of the list.

```
q)4+5 6 7
9 10 11
```

For many operators the implicit iteration is *atomic*: it recurses to the level of atoms.

```
q)(2 3;(4;5 6);(7 8;9)) + (1000;(4000 3000;4000);5000)
1002      1003
4004 3004 4005 4006
5007 5008 5009
```

And there are useful variations.

```
q)("brown";"brawn";"brave")like"br?wn"
110b
q)4 5 6 7 within 5 6
0110b
q)"abc"in"quick brown fox"
011b
q)("brown";"dog")in("quick";"brown";"fox")
10b
```

Iteration operators

When implicit iteration is not enough, you can specify iteration with a group of operators known as *iterators*.[12] The iterators are unary operators with postfix syntax.

[12] In k, and previously in q, these are known as *adverbs*.

An iterator takes a single argument on its left, and returns a derived function.

There are two groups of iterators, corresponding (very) roughly to Map and Reduce. They are known as the map and the accumulator iterators.

Map iteration is inherently parallel: a computation is performed for each item of a list; the computations are independent of each other.

Accumulator iteration is progressive: a sequence in which the result of one computation passes to the next.

Map iterators

The map iterators are Each `'` and its variants. A map iterator derives a function that iterates through the items of a list.

```
q)count ("quick";"brown";"fox")
3
q)ce: count'  / derived function: count each
q)ce ("quick";"brown";"fox")
5 5 3
```

It is not necessary to name the derived function; you can apply it directly.

```
q)max'[("quick";"brown";"fox")]
"uwx"
```

A derived function has infix syntax.

```
q)("quick";"brown";"fox")?'"uio"  / find "u" in "quick", etc
1 5 1
q)("quick";"brown";"fox")?'"o"    / find "o" in each string
5 2 1
```

In the last example above, scalar extension pairs the right-argument atom with each item of the left argument.

Iterators Each Left `\:` and Each Right `/:` map non-atomic arguments similarly.

```
q)1 2 */:10 100 1000  / Each Right
10   20
100  200
1000 2000
q)1 2 *\:10 100 1000  / Each Left
10 100 1000
```

```
20 200 2000

q)"ab",/:"cde"        / Each Right
"abc"
"abd"
"abe"
q)"ab",\:"cde"        / Each Left
"acde"
"bcde"
```

Each Prior `':` maps each item to the previous item.

```
q)+':[1 1]
1 2
q)+':[1 2 1]
1 3 3
q)+':[1 3 3 1]
1 4 6 4
```

Parallel Each `':` applies a unary function to each item of a list, distributing the computation between worker processes.

Accumulators

The accumulators are Scan `\` and Over `/`. The latter can be considered an edge case of Scan.

Scan is an iterator, so it takes a single argument on its left, from which it derives a new function.

The first argument of the derived function is the seed, or initial condition. The result of the first iteration becomes the first argument of the next iteration. And so on.

```
q)0 |\ 3 -40 7 2 4
3 3 7 7 7
```

Above, the derived function `|\` (Greater Scan) has infix syntax and a left (first) argument of zero. The result is calculated as

```
0|3    => 3
```

```
3|-40  => 3
3|7    => 7
7|2    => 7
7|4    => 7
```

With a ternary (rank-3) argument, Scan derives a function that iterates successive first arguments through both right-argument lists.

```
q)ssr\["quick";"cir";"rac"]  / string search-replace
"quirk"
"quark"
"quack"
```

With a unary, Scan derives a function that has only an initial state; there is no list through which to iterate. How, then, is iteration terminated?

Converge If no termination is specified, iteration continues until the result of an iteration matches either the previous result or the initial argument.

```
q)neg\[1]
1 -1
q){x*x}\[.01]
0.01 0.0001 1e-08 1e-16 1e-32 1e-64 1e-128 1e-256 0
```

Do All derived functions have infix syntax. An integer left argument is the number of iterations after which to terminate.

```
q)5{+':[x,0]}\1
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

While If the derived function's left argument is a unary test to be applied to the result of each iteration, iteration will continue until the test returns zero.

```
q){128>sum x}{+':[x,0]}\1
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
```

Over The accumulator derives a function that performs exactly the same computation as Scan but returns only the result of the last iteration.

```
q)ssr/["quick";"cir";"rac"]
"quack"
q){128>sum x}{+':[x,0]}/1
1 7 21 35 35 21 7 1
```

That is, for applicable value `f`

```
f/[x]  <==>  last f\\[x]
```

Apply/Index dualism means data structures can also be iterated.

Let `itinerary` represent the stages of a journey.

```
q)itinerary:`London`Berlin`Rome`Paris!`Paris`London`Berlin`Rome
q)show itinerary:`London`Berlin`Rome`Paris!`Paris`London`Berlin`Rome
London| Paris
Berlin| London
Rome  | Berlin
Paris | Rome

q)itinerary\[`London]                    / start and end in London
`London`Paris`Rome`Berlin

q)3 itinerary\`Rome                      / 3 stages from Rome
`Rome`Berlin`London`Paris

q)(`Rome<>) itinerary\`London            / London to Rome
`London`Paris`Rome
```

```
q)continue:`London`Berlin`Rome`Paris!1101b  / where not to stay
q)continue itinerary\`London
`London`Paris`Rome
```

Projection

A function of rank *N* requires *N* values as its arguments.

Applying it to fewer values projects the function onto them.

The rank of the projection is the number of omitted arguments.

```
q)triple: 3*
q)halve: %[;2]
q)hyphenate: ssr[;" ";"-"]

q)triple 15
45
q)halve 49
24.5
q)hyphenate "quick brown fox"
"quick-brown-fox"
```

Lambdas

A *lambda* in q is a user-defined function, written as an optional argument list followed by a list of expressions separated by semi-colons, all embraced by curly brackets. The result of evaluating the last expression is returned as the result of the function.

The argument list is a list of names separated by semicolons and embraced by brackets; it names the arguments of the function and determines its rank. The argument list may be omitted and default argument names `x`, `y` and `z` used.

```
{[a;b] a2:a*a; b2:b*b; a2+b2-2*a*b}
{(x*x)+(y*y)-2*x*y}
```

In a script, a lambda can be written across multiple lines provided that the continuation lines are indented. (Two spaces are conventional.)

```
foo:{[a;b] a2:a*a;
  b2:b*b;
  a2+b2-2*a*b}
```

Composition

Applicable values (functions, lists, dictionaries, projections) can be composed by juxtaposition and a general null as suffix. All the values except the last must have rank 1. The composition inherits the rank of the last value.

```
q}ds:2* (+)::            / double sum
q)ds[2;3]
10

q)dssr: {x,x} ssr ::     / duplicate ssr
q)dssr["quick";"i";"a"]
"quackquack"
```

qqq

In which we summarise what distinguishes q from other programming languages

This chapter is for you if you *didn't* open the book to improve your q skills, but just want to know something about the language. What's the fuss about?

The chapter is called *qqq* because this is what some authors call a "level three" description. We assume you are familiar with several programming languages already and are looking for points of comparison.

Q is an undocumented proprietary blend of APL and Lisp that supports the kdb+ column-store timeseries database. For three decades kdb+ has been the fastest timeseries database on Wall St.

This has been achieved by a few very experienced people implementing an abstraction with powerful 'mechanical sympathy'.

K code is extremely terse. For example, the lambda `{x~|x}` tests whether a string – or in fact any type of list – is a palindrome.

Q is its query and programming language. (Both q and k are spelled lower case. Sorry.)

It is a DSL embedded in k that

- supports SQL-like queries
- replaces unary overloads of binary operators with English keywords; for example, the q version of the lambda above is `{x~reverse x}`

Getting started with k was easiest for coders already familiar with Iverson Notation as either APL or J. Wrapping q around it made it easier to get people started and productive with q.

Distinctive features of q:

- Functions are first-class objects and can be included in data structures, which can thus form part of the flow of control.
- Dictionaries (a key list paired with a value list) are first-class objects.
- Tables are first-class objects, conceptually *both* a dictionary of column names (key) and a list of columns (value) *and* a list of same-key dictionaries.
- Unlike SQL sets, q tables are *ordered*, which simplifies queries on timeseries.
- A list can be considered a degenerate form of dictionary, in which the key is its indices.
- Because a list (or dictionary) can be considered a function of its key, the same syntax applies a function and indexes a data structure. This syntax is *functional*; brackets for function arguments and array indexes are both syntactic sugar.
- Operators and other functions have no precedence hierarchy. As in its ancestor language APL, the argument of a unary function (or the right argument of a binary) is the result of evaluating the entire expression to its right.
- Prefix notation (also inherited from APL) allows `f[x]` to be written `f x` and thus `f[g[2+h[x]]]` as `f g 2+h x` with significant gain in clarity.

- Most primitives iterate implicitly over lists, dictionaries and tables.
- Most other common forms of iteration (Each, Do, While, Converge) are provided by iteration operators; loops are rarely written.
- The q parser converts q expressions into *parse trees* for evaluation. Parse trees are q lists. Both the parser and the evaluator are q keywords.
- Query syntax is the same for tables in memory and on file.

The above is about the q language only. There is also:

- Interprocess communication baked into primitives: q processes talk by HTTP.
- Persisted tables use the filesystem simply: a table `t` with columns `a`, `b`, and `c` appears in the filesystem as directory `t` with files `a`, `b`, and `c`.
- Data serialisation is the fastest available: none.

If this has raised your interest in q, consult the learning resources in Appendix B.

q and kdb+

In which we consider what we will and won’t learn here

Coders writing q applications have four bodies of knowledge to master.

1. **Language** How to write q that works, exploits its speed, and minimises maintenance costs.
2. **Database** You need to understand the implications of persisting tables to the filesystem[13] and arrange your databases accordingly. You might have maintenance procedures to consider.
3. **Development** How to handle errors, secure communication between processes, use the debugging tools.
4. **Architecture** How to design applications as co-operating kdb+ processes.

[13] Roughly speaking, kdb+ is what happens when q tables are persisted to the filesystem. —Jeffry A. Borror, *Q for Mortals*

Of course, you might not be doing all this. You might, for example, need only to write queries or analytics that read existing data structures.

Either way, this book is about only the firsttopic: the q language.

how

In which we discuss pedagogy: how shall we proceed?

We shall ground our learning in the formalities of syntax to get a thorough understanding of the code we write.

You didn't need this to get started with q. We need it now to understand what q makes possible, especially beyond the bounds of whatever we have learned already.

We shall exploit a key skill you were born with – *play*.

Do not underestimate play. Play is how we explore new capabilities. The fox cubs tumbling in my garden are using play to explore and master their hunting skills. (It also looks fun.)

Matrix mathematics was dicovered or devised in academic play well before anyone found a use for it.

Cultivate your curiosity about what it is possible to express, whether or not you see an immediate use for it.

Forget your readers! Forget self expression! Poetry is research into the unfathomable.
— Aase Berg

Get into the habit of thinking about alternative ways to code something. Most times you will find the clearest efficient expression. And, occasionally, a door will open and you will see a better algorithm altogether.

I knew that

You should encounter much you recognise. This book is written for people already using q.

But some of it will have implications that are new to you. (That's why this book is written for people already using q.)

Thinking in vectors

In particular, we shall be working to shift your thinking.

Most programming languages train your brain to break a solution into small pieces and loop through them. Joel Kaplan on the Array Cast called this "one potato, two potato" thinking.

arraycast.com

We're aiming to retrain your brain to *also* see vector solutions. Vector solutions in q are usually significantly faster than loops. They benefit from q's ability to exploit vector instruction sets in the hardware.

Vector expressions: faster, shorter, clearer to write.

But your brain may have been trained not to see them. We might have a certain amount of *unlearning* to do.

Each chapter ends with some questions, which are answered and discussed in Part III. The separation is to encourage you to work on the questions rather than skimming through.

Busy you can pick up a lot by skimming through the answers, but it does very little to retrain your brain away from *one potato, two potato*.

The answers given are usually shown assembled in steps, because however interesting the solutions might be, the thought process that discovers them is more interesting. This is what you miss if you can't collaborate with experienced qbists; this is what you would miss by skimming; this is what you came here for.

Study and play with the examples and questions, however simple they may seem.

On the beach

Read this book with a q session open. Type stuff. Type the examples. Try some other stuff. Try some variations to see if the examples really work the way you think they do. Get playful. And curious.

Go barefoot. Maybe you're used to working in an IDE or a Jupyter notebook. Instead, consider running q in the command shell. Just you and the interpreter. Barefoot on the beach: sunshine and fresh air.

See code.kx.com/q for instructions.

Does this sound like a dumb way to work? It's how I've seen Arthur Whitney, q's original implementer, work: a few command-shell and text-editor windows; nothing else.

Back to work

After the play we shall get down to work. Part IV puts the lessons into practice with two applications as case studies.

They are both small problems, but non-trivial. Neither is drawn from the world of capital markets in which q grew up. They are both examples of q used for general-purpose programming; small but substantial enough to be useful.

In the first we use q to analyse a webserver's access-request logs. Analytics is the most common use of q so far, so this is likely to feel like familiar ground.

In the second we use q as an editor's tool to scan a Markdown text corpus for problems. Textual analysis is more usually prime territory for regular expressions but here we need to derive structure, and apply rules according to whether we're looking at a heading, body text, an embedded code string, or a code block.

Part II

also

To JOIN what must also be kept separate we need a separator. In q that is the humble semicolon `;`.

Four syllables seems a lot for one character. Perhaps we should simply call it *Also*.

Many characters are overloaded in q and mean different things in different contexts. Not so with Also, which is always and only a separator.

It separates the items of a list.

```
  (42;`life`death`meaningoftheuniverse;"Deep Thought")
```

It separates the arguments of a function.

```
  *[5 6;7]
  ssr[read0`h2g2.txt;"Douglas Adams";"Jeffry Borror"]
```

It separates the indices of a list.

A table is a list of dictionaries.

```
  M:M*/:\:M:til 10
  M[5 6;7]
```

It separates the expressions in a lambda.:

```
  {x2:x*x;y2:y*y;(x2-y2)+2*x*y}
```

And its arguments.

```
{[a;b](a*a)+(2*a*b)-b*b}
```

It separates the expressions in an expression list:

```
$[0<type x;enlist x;x]
```

Cond returns a result but is a control structure, not a function. A functional way to return an atom as a singleton list is to join it to an empty list.

Below, what is the initial value of `i` – and why?

```
while[10>i+:1;-1" ",string i]
```

Questions

1. When should a space follow a semicolon?
2. When should a semicolon follow an expression?
3. What is the last expression in an expression list?
4. Is `;` really a binary operator that returns a list? How would you know?
5. Why does the parse tree for `(2;3)` have `enlist` as its first item?

Lists

Q has a range of data *types* but only three data *structures*: atom, list and dictionary.

An **atom** is well named: it is indivisible. A single integer, byte, bit, symbol, character, timestamp… whatever. The point is: while you may be able to cast it to another type or derive other values from it (e.g. seconds or years from a timestamp) there's only one of it.

A **list** has multiple items and so can be **indexed**. This is true even if the number of items is zero. You can index an empty list; you cannot index an atom.

A **dictionary** is a pair of lists. One (key) is the index to the other (value), so it is a mapping from one list to another.

Provided the key values are unique, that is. Duplicates in the key make results unpredictable.

Tables are first-class objects in q but they too are lists: a table is *a list of like dictionaries*; that is, each item of a table is a dictionary, and all the dictionaries have the same key.

```
q)(`a`b!1 2;`a`b!3 4)  /list of like dictionaries
a b
---
1 2
3 4
q)(`a`b!1 2;`b`a!4 3)  /list of unlike dictionaries
`a`b!1 2
`b`a!4 3
```

List notation

In the previous chapter we saw the parse tree for a very simple list.

```
q)parse"(2;3)"
enlist
2
3
```

First surprise: `(2;3)` is not just an identity for `enlist[2;3]`; list notation is actually syntactic sugar for `enlist`.

That has some implications.

First, we see `enlist` is variadic. It takes as many arguments as we want items in the result. If this seems unfamiliar, it's because we use it mostly as a unary.

We know a function is limited to eight arguments. We also know a list can have more than eight items. Does the eight-argument limit not apply to `enlist`?

```
q)enlist[1;2;3;4;5;6;7;8;9;10]
1 2 3 4 5 6 7 8 9 10
```

It does not.

But if you *apply* `enlist` the limit, er... applies.

```
q).[enlist] til 8  / Apply enlist - a no-op
0 1 2 3 4 5 6 7
q).[enlist] til 9  / a step too far
'rank
  [0]  .[enlist] til 9
       ^
```

If list notation is syntactic sugar for `enlist`, then a list with missing items should be a projection.

```
q)type (`a;;`c)
104h
```

It is. Which suggests you could use it as a projected function.

```
q)(`the;;`brown;) . `quick`fox        / list notation
`the`quick`brown`fox

q)`quick`slick`crafty(`the;;`brown;)'`fox`dog`badger
the quick  brown fox
the slick  brown dog
the crafty brown badger

q)5(`abc;)/`x`y`z
`abc
(`abc;(`abc;(`abc;(`abc;`x`y`z))))

q)(;`abc;)/[`v`w`x`y`z]
((`v`abc`w;`abc;`x);`abc;`y)
`abc
`z
```

LISTS ARE ENTIRELY GENERAL. A list item is an atom, a dictionary, or a list.

Following Borror's usage[14], in a *simple* list every item is an atom of the same datatype. Here we shall call them *vectors*.

[14] *Q for Mortals*

Vectors get stored and processed efficiently in q.

This is so valuable that general lists get converted silently to vectors when possible.

```
q)type a:(3;"a";4)       / general list
0h
q)type b:(3;"a";4) 0 2   / vector of longs
7h
```

This can surprise you. If a list you suppose general has been converted to vector you will not be able to amend an item of a different datatype.

```
q)b[1]:`new              / b is a vector of longs
'type
```

```
    [0]  b[1]:`new
             ^
```

Recall that q has no string datatype. We use the term *string* in a loose and popular sense to mean a char (character) vector. A 'list of strings' is nothing but a list of char vectors. The type of such a list is `0h`: a general list.

A *matrix* is a list of same-length vectors of the same datatype. Nothing constrains the rows to be of the same length.

```
  q)show M:4 cut til 12  / 3x4 matrix
  0 1 2  3
  4 5 6  7
  8 9 10 11

  q)@[M;1;,;]99          / not a matrix
  0 1 2 3
  4 5 6 7 99
  8 9 10 11
```

A table is a collection of *named* same-length vectors.

Column names conform to the same rules function and variable names do.

Watch Out Giving a table column the name of a q keyword breaks qSQL queries.

```
  q)show t:flip`count`next`prev!4 cut til 12  / NO NO NO
  count next prev
  ---------------
  0     4    8
  1     5    9
  2     6    10
  3     7    11

  q)select count*2,prev+3, next-1 from t
  'type
    [0]  select count*2,prev+3, next-1 from t
                     ^
```

Vector literals of some datatypes have their own notation. Use it, because using general list notation prompts your reader to expect something other than a vector.

Indexing lists and dictionaries

A list is a list; a dictionary is a pair of same-length lists,

A dictionary is a mapping from its key list to its value list. In mathematical terms, it is a function: its domain is its key; its range, its value.

A list is also a mapping, but the key is implicit: the indices of its items.

```
q)v:`the`quick`brown`fox            / vector
q)d:0 1 2 3!`the`quick`brown`fox    / dictionary

q)v 2 0 3
`brown`the`fox
q)d 2 0 3
`brown`the`fox
```

Above, `d` mimics a list by setting as its key the `til count` of its value.

Indexing out of the domain

From this we see that a dictionary trivially gives us a sparse array.

```
q)show sl:(5?10000000)!5?`3  / sparse list
4277716| jec
1697727| kfm
7977181| lkk
2912950| kfi
2740134| fgl

q)sl 1697725+til 6          / values following 1697725
```

~~~
```kfm```
~~~

This works because if the argument to Index At is not a key, it returns a null of the value's datatype.

```
q)`a`b`c@5
`
q)10b@5
0b
```

This holds even when the list is empty.

```
q)(0#`)@5
`
q)type(0#`)@5  / result is an atom
-11h
q)(0#100)@5
0N
q)(0#"abc")@5 6 7
"   "
q)(0#100)@(5 6 7;8;9 10 11)
0N 0N 0N
0N
0N 0N 0N
```

Above we observe that Index At is *right atomic*: its result corresponds atom-for-atom to its right argument.

That is straightforward enough for a vector. But a dictionary value can be any kind of list.

What is returned from a general list?

```
q)(`a;"b";3)@5
`
q)("a";`b;3)@5
" "
q)(0;`b;"c")@5
0N
```

The pattern is clear. We get a null of the same datatype as the first item.

Watch Out: There is an exception here to the right-atomic property of Index At. The null value returned from an empty general list is – an empty general list.

```
q)()~()@5
1b

q)()@(5 6 7;8;9 10 11)
(();();())
()
(();();())
```

Questions

1. Find at least three other expressions that return

```
`the`quick`brown`fox
```

2. WIBNI[15] Dict accepted a value list one item longer than the key, the extra item defining the 'default' to be returned instead of null? Show how you might achieve the same effect, returning a symbol xxx from sl instead of a null.

[15] WIBNI: this handy term was coined by the late John Scholes, Dyalog APL implementor.

3. Construct a sparse matrix and show how you would index it.

4. Index At is right atomic. Exploit this to draw a crude ASCII 'heat map' for the temperatures in matrix temp, shading ranges (0–3; 4–6; 7–8).

```
temp:2({x,reverse x}flip::)/{x+/:\:x}til 5
```

Parse trees

The q REPL

- reads the string you type in response to its prompt
- parses it into a parse tree
- evaluates the parse tree and displays the result
- prompts you again

All this is open to you in q. The keyword `parse` returns a parse tree from a string. The keyword `eval` evaluates a parse tree.

So what's a parse tree?

A parse tree is a list. It represents an evaluation ready to be done.

A parse tree with one or zero items represents a *value*. It has 'noun syntax'.

A noun, a thing. Something you can use as a function argument or return as a result.

We could say that, as a noun, an object lacks *agency*. Scholar Simone Weil wrote that *The Iliad* tells of force turning people into things: slaves or corpses. Parentheses do as much in q, cutting an object off from application or indexing. (Unlike Homer, q has Apply and Index to return them to life.)

A value can be any q object. With it just as a noun, nothing happens.

```
q)parse "+"         / Add - but add what to what?
'type
  [0]  parse "+"
```

```
       ^
q))\
q)parse"(+)"          / Add as a noun
+
q)
```

An expression for evaluation has a parse tree with multiple items. The first is a function; the rest are its arguments.

```
q)parse"2+2"
+
2
2
```

Or the first is a list or dictionary, and the rest are indexes. It's all the same syntax.

```
q)parse"`a`b`c 2 0"
,`a`b`c
2 0
```

Above the symbol vector is indexed at `2 0`, because juxtaposition (prefix notation) is syntactic sugar for Apply At/Index At.

Specifying Index At doesn't change the result, but the parser notices.

```
q)parse"`a`b`c@2 0"
@
,`a`b`c
2 0
```

Above, Index At is to be evaluated against arguments `` `a`b`c `` and `2 0`.

The items of a parse tree can themselves be parse trees.

```
q)parse"2*3 4+5"
*
2
(+;3 4;5)
```

And a q expression does not have to get very complicated before its parse tree becomes hard to read. Which is why we write q expressions rather than parse trees.

But we *can* write parse trees. There is nothing special about the lists returned by parse. We can write them too.

```
q)eval (*;2;(+;3 4;5))
16 18
```

Which opens up all kinds of interesting possibilities for control flow.

Oh, k...

Some q keywords get parsed as the unary k operators they represent.

```
q)parse"count where 101b"
#:
(&:;101b)
```

We know # as Take and & as Lesser, but what are #: and &:?

They are the k unary operators we know in q as count and where.

Recall that q is a DSL embedded in k. These forms lurk just below the surface of q. They are called *exposed infrastructure*.

Domain-specific language

The k unary operators are covered by the q unary keywords to make it easier to get started in q. They also mask the variadic nature of operators that have both unary and binary forms, a potential source of confusion for beginners.

You are no longer a beginner. Should you use the k unary operators now? Are count and where just 'trainer wheels' you should discard in order to write with the more arcane-looking unary operators?

You should not. The keywords in q make it more accessible to beginners and it is good style to use them. The professional in you will also remember that however unlikely the k operators are to ever change, the k language is undocumented and unsupported.

Notice it, learn from it – and leave it alone.

Dot

Everything begins with a dot.
— Vassily Kandinsky

Arthur Whitney sometimes says the shortest program in the world is a dot – in the right language.

Let's see how much work q gets out of a dot.

The dot glyph denotes both Apply and Index, which have the same syntax.

You might resist the idea that applying a function and indexing a list are the same, but in q they have the same syntax, so you can write expressions which are agnostic about whether a function is being applied or a list being indexed.

It is not so much an overload; more like two ways of thinking about the same thing.

Whitney attributes this identity to his smarter older brother. (Sherlock Holmes had one too.)

Let's review.

```
q)N10:til 10
q)A:N10+/:\:N10          / addition table
q)M:N10*/:\:N10          / multiplication table
q)A[2;3]                 / 2+3
5
q)M[2;3]                 / 2*3
6
```

For `2 3` (and any other items of `N10`) the following identities hold.

```
A[2;3]  <=>  .[A;2 3]  <=>  .[+;2 3]
M[2;3]  <=>  .[M;2 3]  <=>  .[*;2 3]
```

Put another way, (for the domain `N10`) `.[A;] <=> .[+;]` and `.[M;] <=> .[*;]`; i.e. `A` Index is the same as Apply Add, and `M` Index the same as Apply Multiply.

Apply lets you apply a function to a list of its arguments.

Index lets you retrieve part of a deeply nested data structure.

```
q)DNDS:...                        / deeply nested data structure
q)DNDS[i;j;k] ~ DNDS . (i;j;k)
1b
```

Above, the right arguments of Apply and Index are vectors.

The right arguments can be any list, but beyond vectors the identities above no longer hold.

```
q).[+;(2 3;7 4)]             / 2 3+7 4
9 6
q).[A;(2 3;7 4)]             / A[2 3;7 4]
9  6
10 7
```

We see above the limits of the identity. The arguments of Apply Add (`.[+]`) and Index `A` (`.[A]`) both have a count of 2, but only the items of the former must conform.

An atom conforms with an atom or a list. Two lists conform if they have the same length and the pairs of corresponding items all conform.

Projections and selections

The right arguments of Apply Add and Index `A` need not have length 2.

For Apply, `.[+;y]` for a 1-item `y` returns the projection `+[y 0;]`. For Index, `.[A;y]` for a 1-item `y` returns the selection `A[y;]`.

```
q).[+;1#2]                 / 2+, a projection
+[2]
q).[A;1#2]                 / A[2;], a selection
"klmno"
```

Recall that where nothing follows the final semicolon in a list, the last item is a general null.

```
q).[A;2,(::)]              / A[2;]
"klmno"
q).[A;(::),2]              / A[;2]
"chmrw"
```

Rectangular selections

The right arguments to Apply and Index need not be vectors.

For Apply, that might be obvious.

```
q)ssr . ("file name with spaces";" ";"-")
"file-name-with-spaces"
```

For Index, not so much.

```
q)A . (2 3;7 4 5)          / A[2 3;7 4 5]
9  6 7
10 7 8

q)A . (2 3;::)             / A[2 3;]
2 3 4 5 6 7 8 9  10 11
3 4 5 6 7 8 9 10 11 12

q)A . enlist 2 3           / A[2 3;]
2 3 4 5 6 7 8 9  10 11
3 4 5 6 7 8 9 10 11 12
```

Vector arguments to Index get a cell of the left argument – or a higher-rank result if the index vector has FIXME

Scattered indexing

For a data structure d of rank N, an index vector of length N gets a single cell of d.

This gives us scattered indexing: we can select a list of atoms from anywhere in a data structure, however deeply nested.

Example: Make B a 10×10×10 int array of the first thousand natural numbers. Write an expression that mentions B only once and returns a vector of the four numbers at B[7;8;2], B[1;4;2], B[8;0;5] and B[8;5;2].

```
q)B:10 10 10#1+til 1000           / first 1000 natural numbers
q)B ./:(7 8 2;1 4 2;8 0 5;8 5 2)  / scattered indexing
783 143 806 853
```

Apply and Index

In some ideal universe, Apply and Index would have the same name.

They are denoted by the same symbol, the dot; they have the same syntax; and, with dualities such as Multiply and M above, return the same result. Same, same, same.

But English doesn't provide a verb that captures the duality. It would not help to speak of 'applying' a list or 'indexing' a function. So we risk the confusion. We say Apply with functions and Index with lists. But we write them the same, just the same.

Rank

Apply and Index are general, for functions and lists of any rank.

Rank? You might be more familiar with function rank as *arity*, the number of arguments a function takes. A unary function such as `count` has rank 1; a binary function such as `+` has rank 2; and so on.

Qbists say *argument* rather than *parameter*. We honour our origins in mathematics.

Lists have rank? They do. You might be more familiar with a dictionary or list's rank as its *dimensionality*, the number of dimensions or indices you can use to index it. A vector has rank 1; a matrix has rank 2; and so on.

We use *rank* for both arity and dimensionality because there are things to say about both in q, and they are the same things. By treating rank as a property of functions, dictionaries and lists, we can say the same thing once. (Always a goal of coders.)

The concept of *rank* is inherited from q's ancestor languages APL and J.

Sadly, it has nothing to do with the q keyword `rank`, which is about sorting. Unfortunate potential confusion. Sorry about that.

While we're thinking about terminology, *function* includes operators, keywords and lambdas.

Apply and Apply At

Using Apply to apply a unary function `f` entails passing it a list of the arguments to `f`. But a unary function `f` has only one argument, so the list has only one item. The only way to make a 1-item list from anything but an atom is to use `enlist`. (For an atom `1#` works fine.)

```
q)count . enlist `a`b`c
3
```

What a pain.

And such a nice terse language too. A colleague so much hates writing `enlist` he hacks `enl:enlist` into his development environment.

For unary functions (and lists and dictionaries) q provides Apply At and Index At `@` as syntactic sugar – so you can elide `enlist`.

```
q)count @ `a`b`c
3
```

But that's all `@` is: syntactic sugar. Apply At isn't doing anything that Apply doesn't do.

For Apply At this identity holds:

```
f @ y  <=>  f . enlist y
```

And of course q's prefix syntax lets you elide Apply At as well.

```
q)count `a`b`c
3
```

But Index At *does* do something that Index doesn't do. The identity above for function `f` does not hold for data structure `d` and Index At.

Index At is *right atomic*. It iterates recursively through the right argument and returns a result of the same structure.

```
q)b: (0011b;(101b;010b);000101b)
q)count @ b                        / Apply At
3

q)".#"  @ b                        / Index At
"..##"
("#.#";".#.")
"...#.#"
```

With keyword `count`, Apply At passes the entire list `b` for the function to evaluate.

With list `".#"`, Index At maps each atom of `b` to it.

With Index, in a multidimensional array of rank N you can think of an index vector of length N defining a path to a point in the N-space, much like a filepath in a hierarchical filesystem.

With Index At, the same index vector selects items only from the list's primary dimension.

Index At uses a vector to select top-level items. Index uses it to dive deep.

Trap

Apply and Apply At have an overload to protect evaluation. It specifies what to do if evaluating a function signals an error.

The overload has rank 3. As in the binaries, the first argument is applied to the second.

```
q)@[where;101b]
0 2
q).[+;2 3]
5
```

In the ternary form, if evaluation signals an error, the third argument is evaluated. If the third argument is a function, it is applied to the text of the signal.

```
q)@[where;`t`f`t]
'type
  [0]  @[where;`t`f`t]
       ^
q)@[where;`t`f`t;`oops]         / return `oops
`oops
q)@[where;`t`f`t;'`oops]        / signal oops
'oops
  [0]  @[where;`t`f`t;'`oops]

q).[+;`2`3]
'type
  [0]  .[+;`2`3]
       ^
q).[+;`2`3;`oops]
`oops
q).[+;`2`3;{'"Wrong ",x}]
'Wrong type
  [0]  .[+;`2`3;{'"Wrong ",x}]
       ^
```

Note the scope of the trap. In `.[x;y;z]` only the application of `x` to `y` is protected. If `y` is the result of an expression, evaluation of the expression is not protected by the trap.

```
q).[+;1 2+`1;'`oops]
'type
  [0]  .[+;1 2+`1;'`oops]
              ^
```

Above, the evaluation of `1 2+`1` was unprotected, so the type error was signalled.

Similarly, the trap does not protect evaluation of the third argument.

This is more consequential, because in a production environment the third argument is likely to invoke error-logging code, which therefore needs its own protections.

Applicable values

How to refer to the left arguments of Apply/At and Index/At? They can be functions, lists or dictionaries, according to the insight that "arrays are functions".

Just as English has no verb that can be used as a metaphor for applying a function or indexing a list, it has no noun corresponding to functions, lists and dictionaries.

So we coin the neologism *applicable value*: a value that can be applied. It is horrible, but we have found nothing better.

Amend and Amend At

There is still more work to be squeezed out of the dot.

Index is a functional way for *getting* the value of any atom in a data structure `d`.

Dot has ternary (rank-3) and quaternary (rank-4) overloads for *setting* the value. These overloads are the Amend operator.

The Amend and Amend At operators allow you to change the value of selected items of a list without first naming it. In other words, instead of assigning the result of some expression to (say) `L` then setting the values of `L` at selected indices, you can make the amendments 'in flight'.

That is instead of writing something like

```
L:f2 f1 something
L[1 3 7]&:100
f3 L
```

You can write `f3@[;1 3 7;100&]f2 f1 something`.

Or, if your list is already named `L`, you can write ``@[`L;1 3 7;100&]`` and `L` gets amended 'in place' at positions 1, 3 and 7. And *that* works whether your list is in memory or on disk.

Amend and Amend At are very powerful. Let's look at how that works.

In the ternary form of Amend, the third argument is a unary applicable value. In `.[x;y;z]` Amend applies `z` to the items of `x` at indices `y`.

```
q)show A:til[5]+/:\:til 10
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 3 4 5 6 7 8  9  10 11
3 4 5 6 7 8 9  10 11 12
4 5 6 7 8 9 10 11 12 13

q).[A;(2 3;4 7 1);7&]               / projection
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 3 4 5 6 7 8  7  10 11
3 4 5 6 7 8 9  7  11 12
4 5 6 7 8 9 10 11 12 13

q).[A;(2 3;4 7 1);{x*x}]            / lambda
0 1  2 3 4  5 6  7   8  9
1 2  3 4 5  6 7  8   9  10
```

```
2 9  4 5 36 7 8  81  10 11
3 16 5 6 49 8 9  100 11 12
4 5  6 7 8  9 10 11  12 13

q)show S:til[15]*til 15             / squares
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196

q).[A;(2 3;4 7 1);S]                / vector
0 1  2 3 4  5 6  7   8  9
1 2  3 4 5  6 7  8   9  10
2 9  4 5 36 7 8  81  10 11
3 16 5 6 49 8 9  100 11 12
4 5  6 7 8  9 10 11  12 13

q)show R:20?1000000                 / vector of randoms
644841 853890 728999 102464 498865 951534 444777 52050 601329 796968 776408 9..

q).[A;(2 3;4 7 1);R]
0 1      2 3 4      5 6  7      8  9
1 2      3 4 5      6 7  8      9  10
2 102464 4 5 444777 7 8  796968 10 11
3 498865 5 6 52050  8 9  776408 11 12
4 5      6 7 8      9 10 11     12 13
```

In the quaternary form of Amend the third argument is a binary applicable value, and in the fourth position, its right argument.

```
q).[A;(2 3;4 7 1);&;5]
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 3 4 5 5 7 8  5  10 11
3 4 5 6 5 8 9  5  11 12
4 5 6 7 8 9 10 11 12 13
```

The fourth argument must conform to the selection specified by the second argument.

An atom, as above, always conforms.

```
q).[A;(2 3;4 7 1);+;2 3#1000*til 6]
0 1    2 3 4    5 6  7    8  9
1 2    3 4 5    6 7  8    9  10
2 2003 4 5 6    7 8  1009 10 11
3 5004 5 6 3007 8 9  4010 11 12
4 5    6 7 8    9 10 11   12 13
```

Note above how

```
q)2 3#1000*til 6
0    1000 2000
3000 4000 5000
```

was mapped to the items of `A`.

Lastly, when the Assign operator `:` is its third argument, the quaternary is known as Replace.

```
q).[A;(2 3;4 7 1);:;2 3#1000*til 6]
0 1    2 3 4    5 6  7    8  9
1 2    3 4 5    6 7  8    9  10
2 2000 4 5 0    7 8  1000 10 11
3 5000 5 6 3000 8 9  4000 11 12
4 5    6 7 8    9 10 11   12 13

q).[A;(2 3;4 7 1);:;999]
0 1   2 3 4   5 6  7   8  9
1 2   3 4 5   6 7  8   9  10
2 999 4 5 999 7 8  999 10 11
3 999 5 6 999 8 9  999 11 12
4 5   6 7 8   9 10 11  12 13
```

Amend At

Amend At, the ternary and quaternary overloads of `@`, follows the previous pattern of being syntactic sugar for Amend.

The common use case is to amend a vector.

```
q)@[s;where s=" ";:;"-"]
"the-quick-brown-fox"
```

Type promotion

Watch Out: Neither Amend nor Amend At allow type changes.

```
q).[A;(2 3;4 7 1);<;5]
'type
  [0]  .[A;(2 3;4 7 1);<;5]
       ^

q)@[til 10;3 5 7;0<]
'type
  [0]  @[til 10;3 5 7;0<]
       ^
```

Above, type errors are signalled as the operators refuse to replace integers with booleans.

Questions

1. Write a lambda that returns the leading diagonal from a square matrix.
2. What q objects can you *not* apply or index?
3. Write a lambda that returns the rank of a list or function.
4. Which of the following conform to `L`?

   ```
   L:(1 2 3;(4 5;(6;7 8));9;(10 11;12))

   0
   "wxyz"
   (`a`b`c;(`d`e;(`f;`g`h));`i;(`j`k;`l))
   (1 2 3;(4 5;(6 7;8));9;(10 11;12))
   ("a";("b";("cde";"f"))"ghi";("j";"klmnop"))
   `cow`sheep`cat`dog!5480 9473 6234 1492
   ([]animal:`cow`sheep`cat`dog;id:5480 9473 6234 1492)
   ```

5. For `A:til[5]+/:\:til 10`, revise the expression `.[A;(2 3;4 7 1);<;5]` so that it replaces the six selected numbers with ones and zeroes.

Projection

A function of rank N expects (is entitled to) N arguments. If you apply it to fewer, the result is a *projection*. A function of rank N projected onto M arguments has a rank of $M-N$.

```
q)hyfn8:ssr[;" ";"-"]
q)hyfn8 "quick brown fox"
"quick-brown-fox"
```

Above, ssr is projected onto its second and third arguments, so the projection hyfn8 has rank 1.

Projecting a function has the same syntax as applying it.

```
q)treble:3*
q)quarter:%[;4]

q)f:ssr["quick brown fox";" "]
q)g:ssr . ("quick brown fox";" ")
q)h:.[ssr;("quick brown fox";" ")]

q)f~'(g;h)         / scalar extension applies to function atoms
11b
q)(f;g;h)@'"-_/"
"quick-brown-fox"
"quick_brown_fox"
"quick/brown/fox"
```

THE APPLY/INDEX DUALISM continues to hold here. Consider a function `f` and a list `L`, both of rank N, and let `x` be a list of length N.

Apply: `f1:f . x` is a projection of rank $N-M$.

Index: `L1:L . x` is a list of rank $N-M$.

Let `y` be a list of length $M-N$:

`f1 . y` evaluates `f` over the arguments in `x` and `y`

`L1 . y` returns the item from `L` at the indices in `x` and `y`

PROJECTION is an efficient way to set constants in a function definition. Any computation required is done once when the projection is defined; the argument value is then bound to the projection.

For example, suppose your algorithm requires a long list as a seed.

```
alg0: {[foo;bar]
  seed:10+5*til 1000000;
  ..
  }
```

Above, `alg0` will evaluate `10+5*til 1000000` every time it is called. Instead, project the function onto the vector.

```
alg1: {[seed;foo;bar]
  ..
  }[10+5*til 1000000;;]
```

Above, the value of `seed` is computed once, when `alg1` is defined. This is better than reading a global variable – unless the value involved occupies a lot of memory. (If the value in question is not constant, pass it to the function as an argument in good functional-programming style.)

Questions

FIXME

Composition

The ideal style for a q expression is a sequence of unary functions, each applied to the result of the one that follows it. For example, where mathematical notation would write $fgh(x)$ your q expression would be

```
f g h x
```

Ideally your reader begins reading on the right, considering the evaluations successively to the left end of the line where the result is assigned a name or returned to the console or a calling function.

According to Whitney, Iverson considered infix expressions a sequence of unary projections, e.g. `2*3+100` as `(2*)(3+)100`.

Where you need to iterate explicitly over the items of `x`, you have some choices.

```
f'[g'[h'[x]]]            / A
(f')(g')(h')x            / B
f each g each h each x   / C
{f g h x}each x          / D
{f g h x}'[x]            / E
```

Above, messy expressions (A–C) are equivalent: the derived functions `f'`, `g'`, and `h'` are applied in turn. Expressions (D–E) instead apply the functions `f`, `g`, and `h` to each item of `x`. The result of all four is the same; the distinction might or might not have consequences for performance.

There is an alternative to conjuring a lambda just to chain functions together. The Compose operator glues two functions together and returns a *composition*; its rank is the rank of its second argument.

```
q)hyfn8:ssr[;" ";"-"]
q)foo:('[hyfn8;ssr])                   / Compose hyfn8 and ssr
q)foo["quick brown fox";"f?x";"dog"]
"quick-brown-dog"

q)bar:.[';(hyfn8;ssr)]                 / Apply Compose to hyn8 and ssr
q)bar["quick brown fox";"f?x";"dog"]
"quick-brown-dog"

q)bar1:(') . (hyfn8;ssr)               / Apply Compose to hyn8 and ssr
q)bar1["quick brown fox";"f?x";"dog"]
"quick-brown-dog"
```

We can use Compose Over to compose a list of functions.

Here we write Compose as a projection on *none* of its arguments '`[;]` to distinguish it from other overloads of the quote glyph.

```
q)fubar: ('[;]) over (count;hyfn8;ssr)
q)fubar["quick brown fox";"f?x";"dog"]
15
```

But q syntax offers us something better.

```
q)fb:count hyfn8 ssr ::
q)fb["quick brown fox";"f?x";"dog"]
15
```

Simply suffixing a sequence of functions with the general null `::` suffices to compose them. This yields our sixth (and best) way to iterate `f`, `g`, and `h` over `x`.

```
(f g h::)each x
```

Above, there is a tiny performance advantage in avoiding the lambda, but the difference is unlikely to trouble you. The best argument for preferring compositions is that writing them encourages you to express algorithms as sequences of unary transformations.

Code in this form is easier to parallelise or refactor as microservices.

Questions

1. Is `{..}each x` better q style than `{..}'[x]`?

2. Write a lambda `ind` that returns the indices of a list or dictionary; that is

   ```
   q)ind `a`b`c!1 2 3
   `a`b`c
   q)ind"abc"
   0 1 2
   ```

   Write the lambda as a single expression that is a sequence of unaries, and do not use Cond.

Implicit iteration

One of the most common errors you are likely to make as a qbie is to specify iteration unnecessarily.

Many of us have had our brains trained by other programming languages to loop through data structures. It's close to a reflex.

One potato, two potato.

And this is a problem? Not at all. Exactly what you should do in C-style programming languages. You can probably write For-loops in your sleep. You can probably write For-loops in your sleep.

We need you to wake up.

Zen master Zuigan called out to himself each day.
— *Master.*
— Yes, sir.
— *Become sober.*
— Yes, sir.
— *Do not be deceived by others.*
— Yes, sir; yes, sir.

Binary operators: atomic power

First off, most binary operators iterate implicitly. No doubt you are familiar with examples such as

```
q)10 20 30 + 3 4 5
13 24 35
q)1000 + 3 4 5
1003 1004 1005
```

In the first, Add is applied between corresponding items of the two vectors. Two same-length vectors conform, and all is well.

In the second, atom 1000 is added to each item of the vector `3 4 5`.

The latter is sometimes described as 'scalar extension': a scalar (atom) gets paired with every item of a list. So the following are identities:

```
3 4 5+1000
3 4 5+1000 1000 1000
```

That is true as far as it goes, but there is a good deal more going on here.

In *atomic iteration* scalar extension is applied recursively until atoms are reached.

```
q)L:(1 2 3;(4 5;(6;7 8));9;(10 11;12))
q)L+1000
1001 1002 1003
(1004 1005;(1006;1007 1008))
1009
(1010 1011;1012)
```

Since 1000 is an atom it conforms to any list, so winds up added to every atom in `L`.

You *could* think of atomic iteration here as an atomiser, spraying a cloud of atoms that settle onto every leaf of the `L` tree but the oversimplification obscures how atomic iteration actually works.

```
q)L+1000 2000 3000 4000
1001 1002 1003
(2004 2005;(2006;2007 2008))
3009
(4010 4011;4012)
```

Above we see the four items of `L` paired with the four ints.

- 1000 adds to each of `1 2 3`
- 2000 adds to each of `4 5`; to `6`; to each of `7 8`
- 3000 adds to `9`

- 4000 adds to each of `10 11` and to `12`

```
q)L*(10;(10b);100b;1000)
10 20 30
(4 5;(0;0 0))
9 0 0
(10000 11000;12000)
```

Above

- 10 multiplies `1 2 3`
- `1b` multiplies `4 5`; `0b` multiplies `6` then `7 8`
- `100b` multiplies `9`
- 1000 multiplies each of `10 11` then `12`

Notice that in atomic iteration if one argument is an atom or vector the result mirrors the structure of the other argument.

But in the general case, as immediately above, you can count only on the top-level structure being replicated. That is, the result will have as many items as the argument list/s.

It will also conform to both.

Which operators and keywords iterate atomically?

- Arithmetic operators `+`, `-`, `*`, `%`
- Comparison operators `=`, `<>`, `<`, `<=`, `>=`, `>`
- Logical operators `|`, `&`, `or` and `and`
- Math keywords: `abs`, `acos`, `asin`, `atan`, `ceiling`, `cos`, `div`, `exp`, `floor`, `log`, `mod`, `neg`, `rand`, `reciprocal`, `signum`, `sin`, `sqrt`, `tan`, `xexp`, `xlog`
- `null` and Fill `^`

- Index At `@`
- Cast and Tok `$`, `string`
- `upper`, `lower`

Some of these will be less familiar than others.

Math keywords

A few examples will serve for all.

```
q)ceiling L*1.5
2 3 5
(6 8;(9;11 12))
14
(15 17;18)

q)acos neg (11b;(1b;(111b;11b);11b);1b)
3.141593 3.141593
(3.141593;(3.141593 3.141593 3.141593;3.141593 3.141593);3.141593 3.141593)
3.141593
```

`null` *and Fill*

`null` and Fill are both right-atomic.

```
q)null "string with spaces"
000000100001000000b

q)/Index At returns nulls for arguments outside the right domain
q)null "abcdefghi"@L
000b
(00b;(0b;00b))
1b
(11b;1b)
```

The most common use case for Fill is to replace every null value
with some other value, such as zero.

```
q)0^1 2 3 0N 5 6 0N 8 9  / zero for null
1 2 3 0 5 6 0 8 9
q)"-"^"string with spaces"
"string-with-spaces"
```

But Fill iterates

```
q)"one slick brown dog"^"the quick       fox"
"the quick brown fox"
```

Atomically.

```
q)N:(1 0N 3;(4 5;(6;0N 8));9;(10 0N;12))
q)1000^N
1 1000 3
(4 5;(6;1000 8))
9
(10 1000;12)

q)1000 2000 3000 4000^N
1 1000 3
(4 5;(6;2000 8))
9
(10 4000;12)
```

Index At

Index At is right-atomic: see the questions in Lists.

Cast

```
q)"hij"$101b
1h
0i
1

q)"bhij"$L
111b
(4 5h;(6h;7 8h))
9i
```

```
(10 11;12)
```

Tok

Tok offers a variation on atomic iteration in which the recursion stops at either atoms or strings.

Call this *right string-atomic*.

```
q)"BHIJ"$("123";"45";("6";"78");"999")
0b
45h
6 78i
999
```

Nothing like `like`

The `like` keyword has its own kind of implicit iteration. Its left domain is either a string or a list of strings, according to which it returns a boolean atom or vector.

```
q)"brown"like"br?wn"
1b
q)("brown";"brawn";"frown";"brain")like"br?wn"
1100b
```

We would call `like` *left string-atomic* if it recursed through deeper structures – but it doesn't.

```
q)string[(`brown;`brawn`frown;`brain)]like"br?wn"
'type
  [0]  string[(`brown;`brawn`frown;`brain)]like"br?wn"
                                           ^
```

Nothing iterates like `like`.

'upper' and 'lower'

Both `upper` and `lower` have atomic iteration.

```
q)s
("The";"quick")
(("Brown";"Fox");"JuMpS")
"over"
(("the";"LAZY");"dog")

q)lower s
("the";"quick")
(("brown";"fox");"jumps")
"over"
(("the";"lazy");"dog")
```

Aggregators

Aggregators iterate through the items of a list.

```
q)sum 3 5 8 13                        /items of a vector
29
q)sum (3 5 8 13;2 4 6 8;1 3 5 7)  /rows of a matrix
6 12 19 28
```

The `sum` keyword covers the derived function `+/` Add Over, which is the map-reduce pattern.

A matrix is a list of same-length vectors. In applying Add between list items (rows) the rules of conformity hold, so `sum` can be applied to a nested list of conforming items.

```
q)sum (3 5 8 13;3;1 3 5 7)
7 11 16 23

q)sum (3 4;(5;6 7);(8 9;10))
16 17
20 21
```

The last example proceeds by adding `3 4` to `(5;6 7)` to get `(8;10 11)`then adds `(8 9;10)` to get `(16 17;20 21)`.

From a rectangular array of rank N an aggregator returns a rectangular array of rank $N-1$. Put another way, it reduces (removes) the first dimension of its argument.

Questions

With a little care you can write utility functions that also iterate implicitly. Write a binary function `rng` that returns the inclusive range between its integer arguments, e.g.

```
q)rng[3;7]
3 4 5 6 7
```

Now get `rng` to work with conformable arguments; e.g.

```
q)rng[3;5 8]
3 4 5
3 4 5 6 7 8

q)rng[3 4;5 8]
3 4 5
4 5 6 7 8
```

And so on.

Iterator syntax

Implicit iteration is almost always the best iteration you can get in q. For everything else, there are iterators.

ITERATORS are... amazing. To use them fluently, you need a firm grip on their syntax.

Iterators are known in k, and were previously known in q, as adverbs.

Iterators are unary operators with postfix syntax. That is to say, an iterator has a single argument, which can be written on its left. It returns (or *derives*) a function, the *derived function*.

A simple example:

```
q)("quick";"brown";"fox")?'"ioo"
2 2 1
```

Above, the derived function `?'` finds `"i"` in `"quick"`, `"o"` in `"brown"`, and so on.

The Each iterator takes Find as its argument, to derive the function `?'`. Find Each is a thing. It has type 106 and you can assign it a name.

```
q)fe: ?'          / Find Each
q)type(fe)
106h
q)fe[("quick";"brown";"fox");"ioo"]
2 2 1
```

Postfix syntax is conventional and also good q style. However, the syntax you already know still applies.

```
q)fe ~ ('[?])     / bracket syntax
1b
q)fe ~ @[';?]     / Apply At operator
1b
```

The best way to write Find Each is ?', but if you are applying an iterator to a function that is the result of evaluating an expression, Apply At might be what you need.

Iterators apply to overloads

As the q Reference has it, glyphs such as ? are overloaded. The ? glyph denotes several operators, such as Find, Roll, Select and Vector Conditional. So which of them does `?'` denote?

All of them.

The derived function is as variadic as the argument function.

Many binary operators are secretly variadic

In k many binary operators, such as + and &, have unary overloads, which in q are wrapped as keywords. For example, the unary forms of + and & are, respectively, `flip` and `where`.

These unary overloads are *exposed infrastructure* from k, undocumented and unsupported. But there all the same.

As a consequence, a derived function such as `#'`, which *looks* like it should be unambiguously binary, is actually variadic. It has both binary and unary syntax.

And `count'`, which looks unambiguously unary, has variadic syntax too.

All derived functions have infix syntax

That's right, all of them. Regardless of their actual ranks.

Some of them have ranks to match.

```
q)+/[1000;2 3 4]   / +/ as binary
1009
q)+/[2 3 4]        / +/ as unary
9
```

Some don't, e.g. `count'` and `#'`.

So how is q to know whether, say, `+/` is to be parsed as a binary or a unary?

You tell it.

You have three ways to do that.

- Apply it with bracket syntax, as above, which is unambiguous: e.g. `+/[x;y]` or `+/[x]`.
- Apply it with infix syntax, which is unambiguous, e.g. `x+/y`.
- Parenthesise it and apply it with prefix syntax.

To project a binary derived function, be sure to include the semicolon. That is, `+/[1000;]` is a projection, but `+/[2 3 4]` is not.

The third is almost a syntactic trick. Parenthesising the derived function, e.g. `(+/)` gives it noun syntax. Now, in the expression it appears in, it's a thing. It doesn't have agency. It doesn't act on anything else. Unless… you apply it.

```
q)(+/) . (1000;2 3 4)    / Apply (binary)
1009
q)(+/) @ 2 3 4           / Apply At (unary)
9
```

As usual, good q style prefers infix and prefix syntax.

```
q)1000+/2 3 4
```

```
1009
q)(+/) 2 3 4
9
```

Assigning a derived function removes its infix syntax

```
q)tot: +/
q)tot 2 3 4
9
q)tot[100;2 3 4]
109
```

You can evaluate `100+/2 3 4` but not `100 tot 2 3 4`, because `tot` inherits the variadic property from `+/` but not inherit its infix syntax.

However, because `tot` does not have infix syntax, you can apply it as a unary without ambiguity. That is, you can write `tot 2 3 4`; you do not have to use parentheses, Apply At, or bracket syntax.

Now you understand why derived functions are parenthesised for unary application.

You can write a lot of q without mastering this, because q provides keywords for the most commonly used derived functions, and also for the iterators themselves.

Keywords for unary application of derived functions

```
op   op/   op\
---------------
+    sum   sums
*    prd   prds
|    max   maxs
     any
&    min   mins
     all
```

These are not learning aids to be abandoned once you have mastered iterator syntax.

Good q style prefers the keywords.

Keywords for iterators

Similarly, q has keywords for the iterators themselves.

```
'    each
/    over
\    scan
':   prior
```

These keywords are not themselves iterators, but binary functions with infix syntax.

```
q)type(')       / iterator
103h
q)type(each)    / function
100h
```

The each keyword takes as its left argument a unary function: `x each y` is equivalent to `x'[y]` or `(x')y`.

As functions, these keywords do *not* return derived functions. But you can project them and apply the projection much as you would apply a derived function.

```
q)ced:count'              / derived function
q)cep:count each          / projection: each[count;]
q)ced ("quick";"brown")
5 5
q)cep ("quick";"brown")
5 5
```

Again, good q style prefers `count each x` to `count'[x]` or `(count')x`. But this does have its limits.

```
q)M:(("quick";"brown");("fox";"jumps"))
```

```
q)count''[M]
5 5
3 5
```

Above, the derived function `count'` is the argument of the second Each, deriving the function `count''`. The two iterators amount to one loop nested inside another.

This is not made clearer by use of the keyword.

```
q)(count each)each M
5 5
3 5
q)each[count each]M
5 5
3 5
```

While you can and should use the keyword for simple expressions, to write more complex expressions, master iterator syntax.

Do the same for all the iterator keywords.

Applicable values

The argument of an iterator is an *applicable value*.

That includes functions, but also lists, dictionaries, and communication handles.

```
q)show fsm:-10?10               /finite-state machine
4 3 9 1 5 6 8 0 7 2
q)fsm\[7]
7 0 4 5 6 8

q)show rps:`rock`paper`scissors!`scissors`rock`paper
rock    | scissors
paper   | rock
scissors| paper
q)1 rps\`rock`paper
rock     paper
scissors rock
```

```
q)(-1')("quick";"brown";"fox");
quick
brown
fox
```

Map iterators

The map iterators are all variations on Each.

The Each iterator is the Map in the Map-Reduce paradigm.

It derives a function that iterates across the items of its argument/s.

```
q)/unary
q)ce:count'
q)count("quick";"brown";"fox")
3
q)ce("quick";"brown";"fox")
5 5 3

q)/binary
q)("quick";"brown";"fox")?'("uc";"o";"o")
1 3
2
1

q)/ternary
q)ssr'[("quick";"brown";"fox");("qu";"own";"f?x");("sl";"ave";"dog")]
"slick"
"brave"
"dog"
```

The derived function `count'` has rank 1. We would like to apply count Each with prefix syntax (`count' x`) but recall that *all derived functions have infix syntax* regardless of their ranks, so we can't.

Avoiding bracket syntax and the Apply At operator leaves us the syntactic trick of applying the parenthesised derived function with prefix syntax: `(count')x`, to which q adds (and prefers as good style) the binary keyword `each`. This has infix syntax, so we can write `count each x`.

Use of the `each` keyword has led some authors to call the Each iterator *each-both*, but this term obscures the fact that the iterator is agnostic about rank, so the term is deprecated.

Scalar extension means that in `x f'y` an atom left argument is used as the left argument for

```
q)"*",'("quick";"brown";"fox")
"*quick"
"*brown"
"*fox"
```

We often want a similar effect with an argument that is not an atom.

```
q)("* ";"* ";"* "),'("quick";"brown";"fox")
"* quick"
"* brown"
"* fox"
```

But we are all about removing tedious repetitions. Above, we could project Join onto `"* "` to get a unary.

```
q)("* ",)each("quick";"brown";"fox")
"* quick"
"* brown"
"* fox"
```

Or – a handy piece of syntactic sugar – q provides the Each Left and Each Right iterators.

```
q)"*  ",/:("quick";"brown";"fox")  / Each Right
"*  quick"
"*  brown"
"*  fox"
q)"*  ",\:"quick brown fox"        / Each Left
```

```
"*quick brown fox"
" quick brown fox"
" quick brown fox"
```

Another handy lump of syntactic sugar is Each Prior `':`, which pairs each item in a list with its predecessor.

With a unary, `':` is Each Parallel, which partitions the computation and delegates it to worker tasks.

```
q),':["abcde"]
"a "
"ba"
"cb"
"dc"
"ed"
q){x,'prev x}["abcde"]
"a "
"ba"
"cb"
"dc"
"ed"
```

Much as with `each`, the `prior` keyword provides infix syntax and is to be preferred.

```
q)(,) prior "abcde"
"a "
"ba"
"cb"
"dc"
"ed"
```

Questions

- Why do `count'` and `count each` have different types?
- Will `x f'y` evaluate if both arguments are atoms?

Accumulators

The q Reference shows two accumulators, Over and Scan, with a variety of forms according to the rank of the argument, and the rank with which the derived function is applied.

The reality is much simpler.

To start with, we can consider Over as syntactic sugar for Scan.

```
  f/[x]      <==>      last f\[x]
```

'Syntactic sugar' is not exactly right, because Over iterations discard interim results that Scan must keep and return, making Scan more hungry for memory. But for our purposes we can examine Scan and remember that Over performs the same series of computations, returning only the last result of the series.

Like the other iterators, Scan takes a single argument postfix, i.e. on its left, and derives a new function. Call the argument `f` and the derived function `f\`.

The derived function is applied to its first argument and the result becomes the first argument of the next iteration.

And so on until the termination condition is met; `f\` then returns the original argument followed by the results from all the iterations; `f/` returns only the last result.

If `f` is

binary or higher rank, `f\` iterates through the right argument/s then terminates

unary the termination condition is either specified (Do, While) or that the results have converged (Converge).

Using binary infix operators as a model, the first argument of a binary is also known as the left argument; for a binary f the second argument is the right argument; for functions of higher rank the second and subsequent arguments are the right arguments. This holds whether or not the function has infix syntax.

Rank 2

If `f` has rank 2, it iterates through the right argument.

Recall that derived functions have infix syntax.

For example:

```
q)"* ",\"abc"
"* a"
"* ab"
"* abc"
```

Iteration terminates after traversing the right argument.

Rank 3+

If `f` has rank 3 or higher, the right arguments must conform.

The infix syntax of the derived function is irrelevant: it must be applied with brackets or with Apply.

```
q){x,y,z}\["* ";"abc";"X"]  / "abc" and "X" conform
"* aX"
"* aXbX"
"* aXbXcX"
```

Rank 1

For a unary `f` the termination condition must be specified – or left to converge.

Recall that **derived functions are variadic**.

To *specify* the termination condition, apply `f\` as a binary, with the termination condition as the left argument.

To leave the termination condition *unspecified*, apply `f\` as a unary.

Do and While

Specify either a number of repetitions, or a test that must be passed.

```
q)6{(+)prior x,0}\1             / Do (integer)
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

The expression `1 f\` is a handy point-free alternative to `{(x;f x)}`.

Notice the use here of a composition as the While test.

```
q)(7>count::){(+)prior x,0}\1  / While (test)
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

Converge

The (unspecified) termination condition is that the result matches the result from the prior iteration – or the original argument.

```
q){x*x}\[.01]
0.01 0.0001 1e-08 1e-16 1e-32 1e-64 1e-128 1e-256 0
q)show fsm:-20?20  / finite-state machine
12 9 0 4 14 3 16 18 7 5 1 17 8 10 13 11 6 2 19 15
```

```
q)fsm\[0]
0 12 8 7 18 19 15 11 17 2
q)neg\[1]
1 -1
```

Questions

1. Write a While expression in which the test is a list.
2. For which of the above forms can you use the `scan` and `over` lambdas?
3. Function es takes an argument pair:

   - a list of primes
   - a bitmask (boolean vector)

   and returns a pair: the primes with the next prime appended and the bitmask with that prime and its multiples set false.

   Write es and use it to implement Eratosthenes' Sieve to find the primes up to 10000.

Part III

Answers

Answers to Also

When should a space follow a semicolon?

Trick question.

Because Also is always and only a separator it need *never* be followed by a space.

There are occasions to do so, but this is a question of style, not syntax.

In English typography, a semicolon is conventionally followed by a space. Your brain is already trained to recognise `; ` as a separator. So it makes sense to exploit this and suffix a space.

```
$[test this; then do this; else do that]
```

But q is a lot more terse than English. Consistently suffixed spaces quickly get distracting, maybe annoying.

```
M[i; j; k]
$[x>0; x+3; x*2]
```

A better rule is to omit the spaces *except where they improve clarity*. That‘s an aesthetic judgement. Don’t be afraid to make it.

```
M[i;j;k]
$[x>0;x+3;x*2]
```

When should a semicolon follow an expression?

Not a trick question.

It must *always* follow an expression in an expression list or lambda unless it is the last expression.

A lambda returns as its result the result of evaluating the last expression in its expression list. (See the chapter on lambdas for more detail.)

What is the last expression in an expression list?

The last expression is the one that follows the final semicolon.

An empty expression evaluates to the generic null `::`.

There is *always* a last expression; a lambda *always* has a result.

A 'lambda with no result' is just a loose and popular way of saying a lambda that returns the generic null.

```
q)q:{show x*6;}7
42
q)~[q;::]
1b
```

Is Also actually a binary operator that returns a list?

Cute idea. And some authors have written as much.

But it has erroneous implications, so let's clear it up.

Now the following certainly evaluate to the same:

```
2 3
2,3
(2,3)
```

```
(2;3)
```

But if ; *were* a binary operator returning a list then

```
q)2 3 ~ 2;3
3
```

would return `1b` not 3. The parser provides further clues.

```
q)parse"2,3"
,
2
3
q)parse"2;3"
";"
2
3
q)parse"(2;3)"
enlist
2
3
```

At first glance, the parse trees for `2,3` and `2;3` resemble each other, but a closer look shows that in the second case the first item is a char, not a function. The parser treats semicolon as a special case. (Try parsing `"2+3"`or `"2*3"`for comparison.)

Why does the parse tree for `(2;3)` *have* `enlist` *as its first item?*

Good question.

And it brings us to the next chapter.

Answers to Lists

Browning the fox

```
q)``quick``fox^`the``brown`           / Fill
`the`quick`brown`fox
q){`the,x,`brown,y}[`quick;`fox]      / Join
`the`quick`brown`fox
q)@/[`the``brown`;1 3;:;]`quick`fox   / Amend At
`the`quick`brown`fox
```

Dictionary with a specified default

The Fill operator replaces nulls in its right argument with items from its left argument.

```
q)sl 2912950 + 0 1
`kfi`

q)sld:`xxx^ sl ::    / sl with xxx default
q)sld 2912950 + 0 1
`kfi`xxx
```

Above, `sld` is defined as a composition. It could as easily have been defined as ``{`xxx^sl x}``. But it is a good habit to compose the unaries instead, even if the overhead of the lambda is tiny.

Construct a sparse matrix

A matrix has rank 2; that is to say, two indexes.

We could emulate the syntax of a matrix by adapting the sparse array sa, making its value a list of sparse arrays.

```
q)show sm:(4?10000000)!(4 3#12?1000000)!'4 3#12?`3
9099330| 444777 52050 601329!`bgh`ifn`foh
6471333| 796968 776408 974541!`kdj`eeg`nce
6680492| 584094 946833 766842!`jog`cih`hkp
2200212| 981627 987244 323212!`aea`blm`ooe

q)sm[6471333;776408]
`eeg

q)sm[6471333;776408+0 1]  / nulls for other indices
`eeg`
```

But the emulation breaks down if we omit an index.

```
q)sm[;776408+0 1]
9099330|
6471333| eeg
6680492|
2200212|
```

What were we expecting there? A ten-million item vector?

Let's relax our expectation of an exact emulation of matrix syntax. How many rows in our sparse matrix will have more than one column filled? A better scheme makes the key a list of index pairs.

```
q)show sm:cut[2;8?10000000]!4?`3
5450872 9275078| ojp
2328058 5594916| ckn
3782668 5541596| ohe
4190253 5791571| nem

q)sm@2328058 5594916
`ckn
```

```
q)sm 2328058 5594916
`ckn

q)sm (4190253 5791571;0 0;2328058 5594916)
`nem``ckn
```

Draw an ASCII heat map

Here are the values we want to map.

The composition `{x,reverse x} flip ::` is applied twice to produce four rotations (0°, 90°, 180°, 270°) of the 5×5 addition table.

```
q)show temp:2({x,reverse x}flip::)/ {x+/:\:x}til 5
0 1 2 3 4 4 3 2 1 0
1 2 3 4 5 5 4 3 2 1
2 3 4 5 6 6 5 4 3 2
3 4 5 6 7 7 6 5 4 3
4 5 6 7 8 8 7 6 5 4
4 5 6 7 8 8 7 6 5 4
3 4 5 6 7 7 6 5 4 3
2 3 4 5 6 6 5 4 3 2
1 2 3 4 5 5 4 3 2 1
0 1 2 3 4 4 3 2 1 0
```

The tops of the first two ranges are 3 and 6. Count how many range-tops each item exceeds.

```
q)sum 3 6<\:temp
0 0 0 0 1 1 0 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 1 1 1 1 1 1 0 0
0 1 1 1 2 2 1 1 1 0
1 1 1 2 2 2 2 1 1 1
1 1 1 2 2 2 2 1 1 1
0 1 1 1 2 2 1 1 1 0
0 0 1 1 1 1 1 1 0 0
0 0 0 1 1 1 1 0 0 0
0 0 0 0 1 1 0 0 0 0
```

That’s almost our heat map right there. We just map to characters.

```
q)"./#"sum 3 6<\:temp
"....//...."
```

```
"...////..."
"..//////.."
".///##///."
"///####///"
"///####///"
".///##///."
"..//////.."
"...////..."
"....//...."
```

Parse trees

FIXME

Answers to Dot

Leading diagonal from a square matrix

```
q)show A:5 5#.Q.a             /square char matrix
"abcde"
"fghij"
"klmno"
"pqrst"
"uvwxy"

q)A . 0 0                     /A[0;0]
"a"
q)til count A                 /row indices
0 1 2 3 4
q)2#'til count A              /row:col index pairs
0 0
1 1
2 2
3 3
4 4
q)A ./:2#'til 5               /leading diagonal
"agmsy"
q)ld:{x ./:2#'til count x}   /leading diagonal
```

What q objects can you not apply or index?

This question reveals that *every* object in q can be applied or indexed *except* atoms.

```
q)3 . 0
'type
  [0]  3 . 0
         ^
```

And even here there are exceptions: handles.

The integers 0, 1 and 2 and their negatives are, respectively, handles for the console, stdout and stderr.

```
q)-1 . enlist "quick brown fox";
quick brown fox
```

The same applies to integers assigned as handles to files or sockets.

So the answer is: atoms that are not assigned as communication handles.

A lambda that returns the rank of a list or function

Start with list rank. This is a slightly slippery concept.

The Iversonian ancestors of q, APL and J, know only rectangular arrays. A rank-2 APL array `M` is a matrix; its rows are all the same length; in `M[i;j]`, `j` has the same range for all values of `i`.

In contrast, q inherits from Lisp. A matrix is a list of vectors of uniform type and length; if the rows do not happen to have uniform type and length, it is not a matrix – but it is still a list.

Modern APLs support such lists but distinguish them more easily from matrices.

So how shall we measure list rank in q? Consider a matrix `M`:

```
q)show M:4 5#20?100
12 10 1  90 73
```

```
90 43 90 84 63
93 54 38 97 88
58 68 45 2  39
```

The list's type doesn't distinguish it froma general list.

```
q)type M
0h

q)first\[M]                          / dig for atoms
(12 10 1 90 73;90 43 90 84 63;93 54 38 97 88;58 68 45 2 39)
12 10 1 90 73
12

q)-1_ count each first scan M         / shape
4 5
q)shape:-1_ count each first scan::
q)rnk: count shape::
```

That was all right for actual matrix `M`. Now consider dictionary svg:

```
q)show svg:`line`triangle`square!((0 0;4 5);(1 3;4 7;3 2);(1 1;1 5;5 5;5 1))
line    | (0 0;4 5)
triangle| (1 3;4 7;3 2)
square  | (1 1;1 5;5 5;5 1)

q)svg[;0]
line    | 0 0
triangle| 1 3
square  | 1 1

q)svg[;0;1]
line    | 0
triangle| 3
square  | 1
```

Clearly svg can have three indices.

```
q)svg[`triangle;2;0]
3
q)svg . `triangle,2 0
3
```

But it is not rectangular, and if we apply shape – which assumes it is and examines only the first value – the result suggests the range of the second index is only (0,1).

```
q)shape svg
3 2 2
```

If our criterion for rectangularity in M is that in M[i;j], j has the same range for all values of i, then svg has only rank 1. If we needed exact rectangularity as the criterion for rank, we could write a (much slower) version of rnk that would explore that. For this exercise, we shall allow that svg has rank 3.

A quick check confirms its results for a vector and an atom.

```
q)rnk 1 2 3
1
q)rnk 3
0
```

Yes, an atom has rank 0 – no dimensions at all. That seems intuitive, and corresponds to our inability to index it.

RANK OF A LAMBDA is immediately apparent from inspection. If the lambda has a signature (list of arguments) its rank is their count.

```
q){1+sum";"=(x?"]")#x} string({[a;b;c;d]a+b*c<d})
4
```

Otherwise we look for x, y and z.

```
q){1+last where(1#'"xyz")in" "vs@[x;where not x in .Q.a;:;" "]}string({x+y*z})
3
```

Putting it together:

```
rnkl:{ /lambda rank
  $[first[x]="["; 1+sum";"=(x?"]")#x;
    1+last where(1#'"xyz")in" "vs@[;where not x in .Q.a;:;" "] x]
  } trim 1_-1_ string ::
```

It remains only to examine the type of the argument:

```
rnkg:{  /general rank
  t: type x;
  $[t<0; 0;
    t in 0 98 99; rnk x;
    t<77: 1;
    t=100; rnkl x;
    t=101; 1;
    t=102; 2
    t=103; 1;
    0N ] }
```

Which we could rewrite as a composition

```
rnkg:({0};rnk;{1};rnk;rnkl;{1};{2};{1};{0N})
  sum -1 0 1 78 98 100 101 102 103> type ::
```

Conforming lists

All of them.

In turn:

- `0` is an atom. An atom conforms to any list.
- `L` has four items. Each of the four atoms of "`wxyz`" conforms to any list.
- The nested symbol list exactly mirrors the structure of `L`: sublist to sublist; atom to atom.
- The nested int list is a slight variation from `L`. Where `L` has `(6;7 8)`, it has `(6 7;8)`. But that still conforms, because atom `6` conforms to list `6 7`, and atom `8` conforms to list `7 8`.
- Although the strings in the nested char list are of different lengths than the int sublists of `L`, in each case a char atom pairs to an int sublist; or a string to an int atom.
- The dictionary has four entries; the value of each is an atom.

- The table has four rows; both columns are vectors.

Promoting type

Since Amend will not promote the type from boolean to integer implicitly, we must do it explicitly.

```
q).[A;(2 3;4 7 1);{"j"$x<y};5]      / lambda
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 1 4 5 0 7 8  0  10 11
3 1 5 6 0 8 9  0  11 12
4 5 6 7 8 9 10 11 12 13

q).[A;(2 3;4 7 1);('["j"$;<]);5]    / composition
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 1 4 5 0 7 8  0  10 11
3 1 5 6 0 8 9  0  11 12
4 5 6 7 8 9 10 11 12 13
```

Since the fourth argument is an atom, the ternary form (with a unary as the third argument) will do just as well.

```
q).[A;(2 3;4 7 1);{"j"$x<5}]        / lambda
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 1 4 5 0 7 8  0  10 11
3 1 5 6 0 8 9  0  11 12
4 5 6 7 8 9 10 11 12 13

q).[A;(2 3;4 7 1);('["j"$;5>])]     / composition
0 1 2 3 4 5 6  7  8  9
1 2 3 4 5 6 7  8  9  10
2 1 4 5 0 7 8  0  10 11
3 1 5 6 0 8 9  0  11 12
4 5 6 7 8 9 10 11 12 13
```

Answers to Projection

FIXME

Answers to Composition

Better q style

Using the keyword `each` is better q style than bracket syntax because it facilitates the ideal style of a sequence of unary transformations: instead of `x` on the right one could as easily have a long expression to be evaluated.

In this case the advantage is marginal. If the right argument is simply `x` then `each x` is only slightly clearer than `'[x]`. Consistency is probably decisive here. Since `each` is generally better style for unaries, best to use it consistently and let your reader reserve cognitive power for your algorithm.

Indices of a list or dictionary

This question relates to the idea that a list is a dictionary with an implicit key.

Clearly the first issue is type.

```
q)type d: `a`b`c!1 2 3
99h
```

For a dictionary we apply `key`; for a list we apply `til count`.

```
q)(til count::;key) 99h= type d
!:
```

That's key in its k form as the unary overload of `!`.

Just need to apply it to `d`.

```
q)@[;d] (til count::;key) 99h= type d
`a`b`c
```

And there's our lambda.

```
q)ind: {@[;x](til count::;key)99h=type x}
q)ind d
`a`b`c
q)ind "abc"
0 1 2
```

Answers to Implicit iteration

The basics of rng are simple enough. Here are two ways to express it.

```
q){x _ til y+1}[3;7]    / (I)  _ til
3 4 5 6 7
q){x+til y-x-1}[3;7]    / (II) + til
3 4 5 6 7
```

The difference in their performance characteristics will show up on long vectors. If x and y are both large then (I) will use a lot of memory to generate a short vector. If x is small and y is large then (II) will perform a lot of additions instead of dropping a few items. We can imagine using Cond to select between (I) and (II) on this basis.

Here we'll set those considerations aside and focus on getting rng to iterate implicitly. For that we'll use (II), because Add iterates implicitly, and Drop does not.

```
q)rng0:{x+til y-x-1}
```

Iterating over vectors

We can see that

- x+ is all right for vector x

- and y-x-1 is all right if x and y conform
- but til takes only an atom argument

But an each might work here.

```
q)rng1:{x+til each y-x-1}
q)rng1[3;7]
3 4 5 6 7
q)rng1[3 4;7]
3 4 5 6 7
4 5 6 7
q)rng1[3;7 5]
3 4 5 6 7
3 4 5
q)rng1[3 4;7 5]
3 4 5 6 7
4 5
```

This looks like what we want. But wait. Have we broken our first test case?

rng1[3 4;7] and the others returned 2-item lists. Did rng1[3;7] return a 1-item list, i.e. enlist 3 4 5 6 7?

```
q)count rng1[3;7]
5
```

It did not. It returned a vector. How did that work so nicely?

The answer is that Each over an atom is a no-op.

```
q)til'[3] ~ til[3]
1b
```

Effectively the each in til' is free: it's there if it's needed.

If you find the switch above between each and Each confusing, the following chapter on map iterators should clear it up.

The last example suggests we should think about the domains of rng and the edge cases.

```
q)rng1[3;3]
,3
q)rng1[3;2]
```

```
`long$()
q)rng1[3;1]
'domain
  [1]  rng1:{x+til each y-x-1}
                  ^
```

The first result looks intuitively right. The second looks plausible: the range from 3 to 2 is empty.

We could signal a domain error if x>y. Or we could return an empty list.

The latter is better q style. The language is designed to prefer null results to signalling errors.

```
q)rng2:{x+til each 0|y-x-1}
q)rng2[3 4 ;7 8]
3 4 5 6 7
4 5 6 7 8
q)rng2[3 4 ;7]
3 4 5 6 7
4 5 6 7
q)rng2[3 ;7 8]
3 4 5 6 7
3 4 5 6 7 8
q)rng2[3;3]
,3
q)rng2[3;1]
`long$()
```

Recursive iteration

We have an algorithm now that works for atoms or vectors in both arguments. But we have seen the power of atomic iteration. Can we do anything similar?

Could we write rng such that

```
q)to[(3 4;5);7 8]
(3 4 5 6 7;4 5 6 7)
```

```
  5 6 7 8
```

We can do this by recursing until we find vectors or atoms, checking conformity every time we do so.

First, let's collect our test cases so far.

```
  q)show c:([]x:(3;3;3;3 4;3;3 4);y:(7;3;-5;7;7 8;7 8))
  x   y
  -------
  3   7
  3   3
  3   -5
  3 4 7
  3   7 8
  3 4 7 8
  q)update r:x rng2'y from `c
  `c
  q)c  / test cases
  x   y   r
  ------------------------------
  3   7   3 4 5 6 7
  3   3   ,3
  3   -5  `long$()
  3 4 7   (3 4 5 6 7;4 5 6 7)
  3   7 8 (3 4 5 6 7;3 4 5 6 7 8)
  3 4 7 8 (3 4 5 6 7;4 5 6 7 8)
```

Our next iteration towards `rng` will first see if both arguments are either atom or vector. If they are, it returns the result of `rng2`. If not, at least one argument is a general list (type 0) through which we iterate.

```
  rng3: {$[all type each(x;y); x+til each 0|y-x-1; x .z.s'y]}
```

Above,

- `all type each` will catch any combination of atom and vector arguments;
- `.z.s` refers to the function currently under evaluation; i.e. the way a function can refer to itself whether or not it was assigned a

name.

Add one more test and run them all.

```
q)c,:`x`y`r!((3 4;5);7 8;((3 4 5 6 7;4 5 6 7);5 6 7 8)) / mixed
q)c[`r] ~ c[`x] rng3' c`y
1b
```

So `rng3` works.

We are still Eaching through the cases, though. Is that necessary?

Not at all:

```
q)c[`r] ~ rng3[c`x;c`y]
1b
```

Or, for that matter,

```
q)c[`r] ~ rng3 . c`x`y
1b
```

Beyond Cond

But let's use array thinking here instead of if/then/else.

Using Cond leans towards *one potato, two potato*, because its first argument is an atom. Cond answers one question at a time. Vector thinking prefers lists of answers.

Suppose, unlike the toy utility we have been considering, our normal use case was a *long* list of argument pairs, and that the test `all type each` is comparably expensive to the algorithm itself (as it is here), and that q's vector sympathies make the tests much cheaper in bulk.

Let's test in bulk.

Start small – `all type each` returns a flag: whether we can apply the core function without iterating. It's an atom, but we have an alternative to Cond.

```
rng4:{.[;(x;y)] (.z.s';{x+til each 0|y-x-1})@all type each(x;y)}
```

Above we use the flag to pick one of two functions, which passes to `.[;(x;y)]` to be applied to the arguments. This has more potential. Where Cond works only on a single flag, indexing works with vectors.

Vectors! Vector Conditional is also an alternative to Cond.

```
rng5:{.[;(x;y)] ?[;{x+til each 0|y-x-1};.z.s'] all type each(x;y)}
```

Notice with Vector Conditional above how

- all the arguments are atoms
- the second and third arguments are functions
- the first argument is also an atom, but if it were a vector the result would also be a vector

Notice also how it is projected on its constant second and third arguments so that the evaluated first argument appears on the right. The same is true for Apply. Good q style finds clarity in a sequence of unaries applied with prefix syntax.

Go large

We're warmed up. Now let's consider a list of argument pairs.

We can classify the arguments as atom, mixed, or vector by the signum of their types: respectively `-1 0 1`.

```
q)(type'')flip c`x`y
-7 -7
-7 -7
-7 -7
7  -7
```

```
-7 7
7  7
0  7
q)signum(type'')flip c`x`y
-1 -1
-1 -1
-1 -1
1  -1
-1 1
1  1
0  1
```

And we can envisage a truth table: whether we need to recurse through Each.

If you have learned to iterate a unary such as type using the each keyword, you might wonder why above you see type''. More on this in the next chapter.

```
   | -1  0  1
---+---------
-1 |  0  1  0
 0 |  1  1  1
 1 |  0  1  0
```

We can use the signum of the types to index into the truth table, telling us for each argument pair, whether to recurse through Each.

```
q)(3 3#010111b) ./: 1+signum(type'')flip c`x`y
0000001b
```

Above, we use the expression for scattered indexing that we encountered in the chapter on Dot.

This gives us the functions we need to apply to each argument pair.

```
q){({x+til each 0|y-x-1}.;.z.s)(3 3#010111b)./:1+signum type''[x]}flip c`x`y
.[{x+til each 0|y-x-1}]
.[{x+til each 0|y-x-1}]
.[{x+til each 0|y-x-1}]
.[{x+til each 0|y-x-1}]
.[{x+til each 0|y-x-1}]
.[{x+til each 0|y-x-1}]
{({x+til each 0|y-x-1}.;.z.s)(3 3#010111b)./:1+signum type''[x]}
```

Notice that the lambda is unary, which means `.z.s` is unary, which means that the core lambda itself

```
{x+til each 0|y-x-1}
```

has to be projected by Apply. Now that we have a list of functions to apply to the argument pairs it remains only to Apply At Each to the list of pairs.

Putting it together:

```
rng6:{[args]
  e:(3 3#010111b)./:1+signum(type'')args;  /Each?
  f:({x+til each 0|y-x-1}.;.z.s)@e;        /fns
  f'args }

q)rng6 flip c`x`y
3 4 5 6 7
,3
`long$()
(3 4 5 6 7;4 5 6 7)
(3 4 5 6 7;3 4 5 6 7 8)
(3 4 5 6 7;4 5 6 7 8)
((3 4 5;4 5);7 8)
```

Why put effort into making a function iterate implicitly in this way?

For the same reason q does. When your lower-level functions handle iteration predictably, you can focus your thoughts on more rewarding problems.

Writing For-loops and if/then/else constructs is a habit that takes practice to break.

Don't simply swap For-loops for Each iterators. Consider first whether q *already* performs your iteration without being told to do so. Look first to see if you need to specify anything at all.

Familiarise yourself with how the q primitives iterate implicitly. At first, this effort will distract you from the 'larger' problems you want to solve. Later, having internalised implicit iteration, your mind will focus on the larger problems more clearly.

Answers to Map iterators

Types of `count` *Each and* `count each`

```
q)type (count')      / Each-derived function
106h
q)type (count each)  / projection
104h
```

The two types are composition (`104h`) and an Each-derived function (`106h`).

The key distinction here is that Each (`'`) is an iterator and `each` is a lambda.

```
q)type (')     / iterator
103h
q)type (each)  / lambda
100h
```

So `count'` is a derived function, and `count each` is a projection of `each`, but they are both unary and interchangeable.

Each on atoms

```
q)2 3 4#'"abc"      / list; list
"aa"
"bbb"
```

```
"cccc"
q)2 3 4#'"a"         / list; atom
"aa"
"aaa"
"aaaa"
q)4#'"abc"           / atom; list
"aaaa"
"bbbb"
"cccc"
q)4#'"a"             / atom; atom
"aaaa"
q)(4#'"a") ~ 4#"a"  / same
1b
```

So, under scalar extension, for atom arguments, `(x f'y)~x f y`.

Answers to Accumulators

A list as a test

FIXME

Using the keywords

FIXME

Eratosthenes' Sieve

Okay, es needs to have Apply projected on it to make a unary so that we can use Converge.

We still prefer *writing* es as a binary though, so we don't have to decompose the argument into primes and bitmask.

```
es:{i:y?1b;(x,i+1;y&count[y]#(i#1b),0b)}.  / next step
```

The left argument lists primes identified; the right flags numbers that are not their multiples. The Apply operator turns it into a unary that takes the two arguments as a 2-list.

```
q)es(2;0010101010101010101010b)
2 3
```

```
  000010100010100010100b
```

The termination condition is that the last prime found exceeds the square root of x.

```
  sqrt[x]> last first ::
```

Writing the test as a composition handily has the square root of x computed just once, when the test is specified.

```
  q)(sqrt[20]>last first::)es\(2;001010101010101010101010b)
  2     001010101010101010101010b
  2 3   000010100010100010100b
  2 3 5 000000100010100010100b
```

The bits flag the primes left in the sieve.

```
  q)@[;1;1+where::](sqrt[20]>last first::)es/(2;001010101010101010101010b)
  2 3 5
  7 11 13 17 19
```

Putting it together...

```
  q){raze@[;1;1+where::](sqrt[x]>last first::)es/(2;0b,x#01b)}10000
  2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 ..
```