K: Remarks on Style

May 1, 1995

Introduction

Most of us recognize "stylish" code when we see it, even though we are unable to say, in general, what good style is, or even whether there is one way to achieve it. Stylish programming shows the author's concern to write code that is transparent, consistent, direct, and readable. The care taken to make code stylish is one aspect of writing "good code": programs which are small, fast, and extensible.

This document -- a collection of remarks, observations, advice, rules, and models -- is intended to be the nucleus for a proper manual of K style. My hope is that K programmers will find the project sufficiently interesting to think through the consequences of their current practices and try to articulate the principles on which those practices are based.

I have not sought to group the material under fixed chapter headings: in my opinion, that would be premature. Instead, it is just one remark after another.

Kernighan and Plauger, in their classic work on the elements of programming style, organize issues of style under seven headings: expression, control structure, program structure, input/output, common blunders, efficiency, and documentation.

In the 1970's, addressing an audience of mainframe Fortran programmers, this way of dissecting the components of programming style had much to recommend it. For us, the emphasis is distributed differently. Some wholly new problems have arisen, and some, which once appeared central to the business of programming, now seem to be artifacts of the languages then used. As examples of the first class, consider: client-server interaction, GUI control, timing problems, and programming with arrays and lists. And as examples of the second, recall how much mental effort was expended on the proper way to process files, write loops, and branch within a program.

In the final part of this paper, I reprint (with commentary) the complete set of style rules from Kernighan and Plauger. In a later version, I will include a summary of whatever rules survive winnowing by the K programming community.

K maxims

Style improves with knowledge of the language. The more K you know, the less code you will need to write. K has been designed to simplify many of the tasks routinely required in writing distributed event-driven applications. Stay alert for code that feels "wrong". Expect K to provide a simple solution to what seems like a simple problem.

Program with, and not against, the grain of the language. For example, K provides two notations for constructing a list:

```
1 2.2 3                 / real vector
(1;2.2;3)               / list of integers and reals
```

Vector notation is used to create structures of atoms of the same type. List notation is more general, and can be used to create lists whose elements are arbitrary K data. Vector notation requires less typing. Vectors have the most efficient form of storage. And vectors compute faster than general lists. K nudges you in the direction of efficiency by making efficient structures easier to create than inefficient ones. Seek to apply this principle to your own designs.

Seek opportunities to throw away code. A fact to bear in mind is that all of K is contained in 1200 lines of C code. That figure encompasses the language as well as all the code for interprocess communications (IPC), windows (GUI), object-file management (database), and operating system interaction. K's freedom from bugs is not unrelated to the high degree of compression achieved in the source code.

Names

Names should be easy to type. That is, short, and containing no, or only a very few, shifted characters.

The Great Divide in K is between global and local variables. Only global variables can be on the screen. Only globals can have attributes such as assignment triggers and dependency definitions. Global variables contain the persistent data, programs, and attributes of your application.

Although every piece of named K data is a variable, I will reserve this term for variables whose datatype is not functional. Function-valued variables I will call "functions", leaving context to sort out whether I am referring to the name, to the variable, or to its data.

In K, we use dictionaries in two ways: as name-spaces for functions and variables, and as symbolically-indexed structured data. Official K calls global dictionaries "directories", but I want to reserve this term for dictionaries used as name-spaces. For example,

```
\d stat
avg:{(+/x)%#x}
var:{avg[_sqr x]-_sqr avg x}
std:{_sqrt var x}
```

Directory names should be as short as possible, in order to minimize the length of absolute paths.

I'll use the term "dictionary" for local dictionary-valued variables, and for global dictionaries when used as structured data. For example,

```
Bond:.+(`tick`coup`mat;(`xyz;9.2;19990815))
```

Attribute variables are always global. K will not even parse the following function:

```
f:{
  v..att:10
}
```

Since an attribute dictionary cannot be local, there is never any ambiguity about whether `a` is global in

```
x..y.a
```

Therefore, we will not require that it be named to look like a global variable. Typically, an attribute is replicated many times on the K tree, so naming it like a local will help reinforce the idea that attributes are "private" to their objects.

Conventions for names

The following list of naming conventions are designed to help you write intelligible, compact code.

Directories begin with a lower-case letter.

Global variables begin with an upper-case letter.

Global constants are all upper-case.

Global functions begin with a lower-case letter, and contain at least three characters.

Local variables begin with a lower-case letter, and contain one or two characters.

Local functions begin with a lower-case letter, and contain exactly two characters.

If an **attribute** begins with a lower-case letter, it must be at least three characters long (to avoid potential clashes with system attributes).

Variables and functions in an attribute dictionary follow the rules for locals.

The letters `x`, `y`, and `z` are used exclusively for the **first, second, and third arguments to a function**.

Sample names

Global variables:

```
Global:10 20 30
NUM:"0123456789"
```

A global function:

```
dlb:{(+/&\x=" ")_ x}
```

A directory:

```
\d .my.u
ss:{&0<#:'x _ss\:y}
```

A table with three fields:

```
Bonds[`Tick`Coup`Mat]:(`abc`def`ghi
                       9.4 10.2 6.5
                       19990815 19991201 20010101)
```

Local variables:

```
foo:{
  f:*x
  r:1_ x
  :
```

A local function:

```
goo:{
  db:_dv[;" "]
  :
```

Entries in an attribute dictionary:

```
x..att.v:`one`two
x..att.eq:{x=y}
```

Names_with_underscores

Use of the underscore character as a separator in multipart names should be avoided, since this will interfere with the commonly used K verb written "_". Phrases such as the following are difficult to read:

```
a_b _ c_d _ e_f_g
```

If the result is legible, avoid separators altogether:

```
a:ab _ cd _ efg
```

If you must distinguish the parts of a name, do so by shifting case:

```
aNameWithParts
```

The only context in which the underscore should occur as part of a name is where that name has its origin outside of K; for example, if you are reading a Sybase table into K, and that table contains names with underscores.

NamesWithIntermittentUpperCase

Long descriptive names are almost always less readable than short ones. Consider the following sequence of ever-shorter names:

```
deleteLeadingBlanks
deLeadBlns
dlb
```

The first variant is clearly the most meaningful: it conveys unambiguously the effect of the function. The last is completely cryptic: find the documentation, or read the comment for the line in which the function occurs.

Some programmers think that variant 2 is a good compromise: abbreviate down to just the point where meaning starts to evaporate. (No question: this function deflates lead balloons).

Now, compare the following two lines:

```
newString:deleteLeadinBlanks'oldStrings

n:dlb'os          / delet leading blanks from old strings
```

Which is easier to read?
Which is easier to understand?
Which is easier to type?
Which line contains the typo?

Long names makes it hard to see syntactic structure, which contributes as importantly to the meaning of an expression as does the "meaning" of a single function or an individual piece of data. Long names also make it hard to see typos, since any twelve-character name looks much like itself minus one letter (which is why proof-readers get paid -- say, did you find ***both*** typos in the last example?).

If it's important, you'll type it a lot.
If you type it a lot, it should be short.

Prevention of cruelty to vowels

Some programmers try to reduce typing and keep their code descriptive by purging all or most vowels from long names, together with those consonants which seem redundant. For example, the path

```
slstrk.posn.Tbl.NwBlnc
```

is almost certainly the result of performing a vowelectomy on

```
salestrack.position.Table.NewBalance
```

37 characters worth of name reduced to 23: 35% shorter, without loss of meaning. How could this fail to be a Good Thing?

While the first name is shorter, it is definitely harder to type than the unabbreviated form. Convince yourself of this by typing both names from memory four or five times at normal typing speed. The chances are good that you will mistype the first name more often than the second. That fact alone should warn us off the practice.

Use names made of either a single letter, a single syllable, or the first letter of each word that describes the object.

For example:

```
st.pos.Tab.New
```

Typing accuracy as well as typing speed is at stake here. Sometimes we are actually compelled to work together face-to-face, in which case we rely on speech rather than email to convey information. We should not spend a lot of time having to explain how things are spelled, or which letters in a name are upper-case, or which vowels didn't get dropped and which consonants did.

Practice the following typing instructions with a friend:

"s l s t r k dot p o s n dot capital t b l dot capital n e w capital b l n c"
"salestrack dot position dot capital t table dot new capital b balance"
"s t dot pos dot capital t tab dot capital n new"

A name that is easy to pronounce is easy to remember and easy to type.

Defaulting the argument list

Functions of zero, one, two, and three arguments:

```
zero:{a+b}
one:{x+10}
two:{x+y}
three:{x+y%z}
```

Be consistent in your use of **x**, **y**, and **z** to mean the first, second, and third arguments to a function. Even if you elect not to use the default pattern provided by K, avoid using these letters as local variables, or as arguments occupying other positions in the argument list. For example:

```
ugh:{[y;x]
  :

urk:{[a;b]
  x:a+b
  :
```

Use of braces

A function containing one statement which returns a result:

```
foo:{x+y}
```

A function containing many statements which returns a result:

```
foo:{
  a:*x
  b:*|x
  a+b}
```

A multi-line function which does not return a result should always end with an isolated brace:

```
foo:{
  Sum::x+y
  Prod::x*y
}
```

and not a semicolon:

```
foo:{
  Sum::x+y
  Prod::x*y;}
```

A function with one line which does not return a result can be written on a single line, using a semicolon:

```
foo:{Sum::x+y;}
```

but consistent use of the isolated brace to mean "no result" suggests we write:

```
foo:{
  Sum::x+y
}
```

Indentation

Indent no more than two spaces within a function:

```
foo:{
  a:x+y
  :
```

Similarly, within control statements:

```
if[v=_n
  v:.()]

do[#x
  a:!#x
  b:+/a]

while[i<5
  a:f[b;i]
  i+:1]
```

"Case" statements may pair test and action expressions:

```
r::[v>0;v+1        / if v>0 increment v
    v<0;v-1        / if v<0 decrement v
        v]         / if v=0 v
```

Align lists and symbol-value pairs:

```
L:(1 2 3 4 5
   `one`two`three`four`five)

D:.+((`one;1)
     (`two;2)
     (`three;3)
     (`four;4))
```

Single-space indentation is acceptable:

```
foo:{
 a:x+y
:
```

Spaced-out code

K requires spaces in certain contexts and disallows them in others. The rules are simple, and are spelled out in the Syntax chapter of the K language manual. Otherwise, you are free to deploy whitespace in the service of readability.

Some programmers favor whitespace around all primitives and punctuation:

```
quote1 : { { x | -1 ! x } @ { ~ x = y }\ "'" = x }
```

Some use whitespace only around primitives:

```
quote2 : {{x | -1 ! x} @ {~ x = y}\ "'" = x}
```

Others avoid cosmetic whitespace altogether:

```
quote3:{{x|-1!x}@{~x=y}\"'"=x}
```

A practice we countenance allocates whitespace grudgingly, using blanks to nudge the eye:

```
quote4:{{x|-1!x} @ {~x=y}\ "'"=x}
```

Advocates of the last approach argue that it is easier to see the structure of a complex expression when it is presented in dense form:

```
f @ g\ z
```

The practice we recommend accepts a single blank only following the statement separator (semicolon):

```
a:b+c; d:e-f
```

Otherwise, use no whitespace, or

Only as much whitespace as makes syntactic structure salient.

Parentheses

Avoid unnecessary parentheses. The grammar of K is simple, and there is no precedence order for the primitives. For example, the expression to add `a` to the product of `b` and `c`,

```
a+b*c
```

should not be written

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

Redundant parentheses are visual red herrings. Although extra parentheses are useful as training wheels, most K programmers eventually internalize the preferred method of reading K, which is ***left-to-right***. To encounter a parenthesized expression is to assume that the parentheses are necessary.

Null statements

Since newline acts as a line-separator equivalent to semicolon, ending a line with both semicolon and newline creates a null statement in your function. The first line of the following function contains two statements: an assignment of a, followed by a null statement.

```
foo:{
  a:x+y;
  :
```

Newline is also a separator for the parts of a list:

```
a:(`one;1)
b:(`one
    1)
a~b
1
```

Avoid unnecessary semicolons.

How long is a line?

Ideally, a physical line of code contains exactly one K statement, and a K statement encodes exactly one thought.

Some thoughts are too large or complex to represent in a single readable K statement. Some thoughts are too small or trivial to justify the expenditure of a whole physical line.

An example of a line which contains several K statements, each of which encodes a small piece of a single thought:

```
a:x 0; b:x 1; c:x 2; d:x 3; e:x 4
```

(Observe how blanks are used after each semicolon to achieve visual separation. This is legitimate, since whitespace should help us see computational structure. Not so in

```
v:m[a; b; ]
```

where the spaces create a visual obstacle.)

Sometimes a thought consists of a simple operation on parts whose construction is complicated:

```
r:(...x 0...;...x 1...;...x 2...;...x 3...)
```

where the dots stand for complicated calculations on the parts of `x`. If the calculations are mutually independent, break the construction of `r` into a set of preliminary steps:

```
d:...x 3...
c:...x 2...
b:...x 1...
a:...x 0...
r:(a;b;c;d)
```

A good rule of thumb is that a line should consist of no more than 50 characters, including the initial spaces. This leaves room for 40 or 50 characters' worth of comment.

If your function looks tall and skinny, see whether you're breaking ideas up into pieces which are too small. If your function looks short and fat, see whether you're trying to express more than one idea on each line.

Internal assignment

Internal assignments can make code hard to read, especially if there's a lot of it. For example, in the following line

```
i:&(k>v@y)&(k:!s)<*(1+y)_ v,s:#n
```

there are internal assignments of `k` and `s`.

Internal assignments should never span lines. In the example above, `k` and `s` should be temporary assignments on the way to computing `i`. If K had a "where" clause, we would write this differently:

```
i:&(k>v@y)&k<*(1+y)_ v,s where k:!s where s:#n
```

The aim of this line is to construct `i`. Rather than compute `#n` three times and `!#n` twice, we snarf these values into temporaries the first time they are computed.

If `k` and `s` are to be used on subsequent lines, they should be broken out as explicit assignments:

```
s:#n
k:!s
i:&(k>v@y)&k<*(1+y)_ v,s
:
```

Now the reader can scan the left edge of the function text and find all non-temporary assignments.

Test your code by mentally drawing arrows from each assignment to each use of the name assigned. Arrows should never go up or to the right: that means that you're re-using a name. Arrows going down should always originate at the left-edge of a line. Arrows going left are temporaries, and should never also go down.

It's a good idea to pick one or two letters for the purpose of temporary assignment, and use them consistently and exclusively for this purpose. (`t` and `u` are good. And if you do so, then modify the arrow test above to allow for re-use of those names on successive lines.)

How long is a function?

Strive to keep functions as short as possible. Most K functions are four or five lines long. Although certain programming tasks may demand larger structures, anything over ten lines should trigger mild alarm, and a function containing more than fifty lines will probably attract roaches.

How many arguments should a function have?

One of the most interesting properties of the K language is that function application is itself a function. Application takes a function on the left and a list of arguments on the right, and applies the function to the list, mapping each item of the list to the corresponding argument of the function.

Used properly, this is a wonderfully convenient feature, since functions can be designed to apply to many small, named things, which are themselves parts of a list or dictionary. Keeping the parts in a composite data-structure is good for organizational reasons. Designing the function to see those parts as separate, named entities reduces code and improves readability.

For example, suppose that `v` is a list of five heterogenous items, and `f` is a function of which re-arranges those items:

```
v:(12;`xyz;"abc";1.1;10 20 30)
f:{[i;s;c;r;l] (s;c;i;l;r)}
```

Application of `f` to `v` serially maps the items of `v` to the arguments of `f`:

```
  f . v
(`xyz;"abc";12;10 20 30;1.1)
```

Compare `f` with the monadic list-functions `g` and `h`:

```
g:{x[1 2 0 4 3]}          / what?

h:{
  i:x 0; s:x 1; c:x 2; r:x 3; l:x 4
  (s;c;i;l;r)}            / why?
```

Good function design is the outcome of taking multiple perspectives on functionality: from within the function, we want the data already decomposed and ready for processing; from outside, the function is a node in a network of calculation routines, and we strive for uniformity and simplicity. Balance these competing forces.

Design functions with meaningful arguments.

Modularity

Function modularity is a Good Thing, and one way to achieve it is through the judicious use of subfunction abstraction. Artfully modularized systems are easier to understand than ones which are either impenetrably monolithic or which have been decomposed haphazardly.

Kernighan and Plauger's rules of modularity still apply to K:

Use subfunctions.
Making the coupling between modules visible.
Each module should do one thing well.
Make sure every module hides something.

to which we add two new rules:

Hide shared subfunctions in subdirectories.
Localize unshared subfunctions.

As subfunctions are abstracted, the number of global functions increases. This may help us in the reading of program text, but interfere with our attempts to interactively explore the system. If `d` is a directory, then we would like `!d` to consist of just the entry-points of `d`. So where do the non-entry point subfunctions go?

First, define `d.u`, the utility directory of `d`. Banish all shared subfunctions of `d` to `d.u`. A shared subfunction is one which is called by more than one function in `d` and/or `d.u`. In large systems, or where the entry points themselves are shared subfunctions, it may pay to place all the code in `d.u`, and make the functions in `d` simple covers on the "real" entry-points living in `d.u`. For an example of this approach, see Q.

Next, localize all subfunctions which are called by only one function. For example:

We want to write a function `tree` which produces an indented list representation of the structure of some portion of the K tree. `tree` takes an initial directory `x` and recursively descends from `x` until it finds a non-dictionary. Each recursive step increments a counter variable `y`, which tells the level of descent and is used to calculate the number of spaces to prefix to the (unqualified) directory name.

Here is a version of `tree`:

```
tree:{
  :[@c:!`${x,".",y}/$x
    ()
    (,(y#" "),$*|x),,/(x,/:c)_f\:y+1]}

` 0: tree[,`;0]

k
```

```
q
 u
  AGG
  ORD
 SQL
  P
  Result
 A
  P
   Source
   Target
 B
  View
  Fields
```

Observe that the user of **tree** has to supply an initial value for the counter (always 0), and that the starting directory, conceptually an atom, must be a one-element list.

We can't avoid making **tree** dyadic, since K doesn't support polyadic functions, and we certainly don't want to gunk up the code with a mostly-useless enlist-if-atomic condition for the directory path.

We'd also like to have **tree** print the list and return nil, and that seems to involve testing the counter to decide whether to return a result (if the call is recursive) or print with no result (if the call is top-level). We are now well beyond gunk.

In languages without local functions, we would probably settle for having two functions:

```
tree:{` 0:treeRec[,x;0]}
treeRec:{ ...
```

wher **treeRec** is the recursive function just described, and **tree** is the entry-point.

In K, the solution is to make the recursive routine a local subfunction:

```
tree:{
  tr:{:[@c:!`${x,".",y}/$x
        ()
        (,(y#" "),$*|x),,/(x,/:c)_f\:y+1]}
  ` 0:tr[,x;0]
 }
```

Attribute dictionaries

An attribute is just a variable in a dictionary whose name ends with a dot:

```
var..att
```

att is an attribute of **var** because it is an entry in the dictionary **var.**. (That's not a typo: the name of the attribute dictionary is "**var.**".)

K owns a handful of attribute names: **t**, **d**, **c**, &c. These are the system attributes. Other utilities own other attributes: Q owns **q**, the TeX report-writer owns **T**, &c.

Although any application can specify a range of attributes which it will own, we strongly suggest that sets of application-specific attributes be gathered under a single attribute-dictionary. For example, if the application **abc** uses ten attributes **xxx**, **yyy**, **zzz**, ..., then we recommend:

```
var..abc.xxx
var..abc.yyy
var..abc.zzz
:
```

and not:

```
var..xxx
var..yyy
var..zzz
:
```

There are two reasons for this policy.

First, application variables typically serve as host to attributes managed by several independent systems. We want to minimize the chances of name-clash at the attribute level. For example, **x** and **y** might each define a "status" attribute.

Second, **!var.** should be small and informative. If **var** uses **x**, and **x** places ten attributes on variables which use it, then **!var.** should tells us that **var** uses **x**. This is good information-hiding.

Comments

Every line of a K function should be commented. Code and comment should be kept on the same line. The comments should be aligned for ease of reading. Indentation of the comment block should mirror that of the function.

```
sendLeft:{                          / update from left to right link at x=_i
  s:get[_v;`G]                      /   absolute source field name
  h:(::;*:)?get[_v;`H]              /   0 (::) or 1 (*:)
  j:enlist@*x                       /   itemwise index
  p:.[att[_d;`J];undot s]           /   left link partition
  i::[h;p j;p . x]                  /   index map into right table
  d::[h;_v j;_v . x]                /   first (*:) or all (::)
  @[s;i;:;d]                        /   update source field
}
```

Conventions for comments:

If the line is a **local assignment**, the comment should describe the meaning or role of the variable.

If the line is a **side-effect**, the comment should be in the imperative mode, indicating the action performed.

If the line is a **control structure**, the comment should describe the meaning of the condition or loop.

"Dangerous curves" should be documented in all caps, or contain some other eye-catching visual device.

Header comments

How to comment the header of a K function is a matter of some controversy. We recommend that the header be treated as just another line in the function, with this one difference: the comment should describe the meaning of the function as a whole.

Some programmers believe that the header comment should contain standardized information about the function as a whole. For example:

```
sendLeft:{
  / ds: update from left to right link
  / ts: xyz 1/1/95
  / x:  _i or _n if _i is ()
  / rs: none
  / gr: _v
```

The header comment block contains a function description, author/timestamp data, argument and result documentation, and information about which global variables are referenced and assigned. Unfortunately, this style tends to bloat functions with non-executable lines. (Recall that the typical K function contains but five lines.)

The same information can be packed into the `h` ("help") attribute for the function:

```
\d sendLeft..h

ds:"update from left to right link"
ts:"xyz 1/1/95"
arg.x:"_i or _n if _i is ()"
rs:"none"
gr:"_v"
```

Any K variable (and functions are just variables) can have a help dictionary, which then can be displayed or inspected. This is consistent with K's general approach to the constructs of programming: if it matters, make it first-class. Eventually, this form of help may be directly supported by K's interactive debugger.

Header comments and debugging

Some programmers repeat the name of the function as a comment in the header. This is to assist in debugging. For example,

```
  foo:{        / foo
    goo x
  }

  goo:{        / goo
    !x
  }

  foo -3
{              / goo ... + 1
domain error
!x
^
>
```

(Remember that "`f a`" evaluates `f` (to a function) and then applies that value to `a`. `{x+y}` is no more in need of a name than `17` is.)

This approach has one serious drawback (apart from leading you to spend the header comment on redundant information). Namely, in certain circumstances it can give bad information:

```
  hoo:goo
  hoo -3
{        / goo ... + 1
:
```

Here is a way to recover the name of the broken function which entails no extra work:

```
  foo:{goo x}
  goo:{!x}

  foo -3
domain error
{!x}
 ^
>
'

{goo x}
 ^
>
```

By signalling up one level in the stack (`'`), the error is displayed in `foo` on the call to `goo`.

The right conditional

K has two forms of conditional evaluation: `:[]` and `if`. The following rules for when to use which conditional are designed to help you write code where useful information is conveyed by your choice of conditional.

`:[]` has the structure:

```
r::[cond1
    true1
    :
    condN
    trueN
    false]
```

`if` has the structure:

```
if[cond
   true1
   :
   trueN]
```

Use `if` when side-effects are desired; for example, to assign default values to the arguments of a function:

```
foo:{
  if[x~_n;x:101]
  :
```

Although `if` does not support if-then-else logic, it should be used even when that logic is required but where side-effects are intended:

```
if[b:x>5;foo[x]]
if[~b;goo[x]]
```

and not:

```
:[x>5;foo[x];goo[x]]
```

The conditional should be used only when a result is desired:

```
s::[x~_n;!y;,y]
```

Use `if` when testing to return from a function with an explicit result:

```
foo:{
  if[()~x;:0]
  :
```

Reverse the condition when the function can return nil:

```
goo:{
  if[~()~x
    :
```

Use **if**, not **:[]**, when signalling from within a function:

```
if[x=0;'"cannot be zero"]
:
```

Consistent use of **if** and the conditional will make your code more readable: seeing **if**, you know that a side-effect is sought and a result is not; seeing **:[]**, you know that a result is intended unconditionally.

De-looping

List algorithms in K are simpler, shorter, clearer, and almost always faster than their looping counterparts. Summing all the elements of a matrix, all the rows of a matrix, all the columns of a matrix `m`:

```
+/m                 / sum the columns of m
+/'m                / sum the rows of m
+//m                / sum m
```

Summing the rows with a nested loop:

```
vsum:{[m]           / shoot this code
  r:(#m)#0
  do[#m
    v:m[i]
    s:0
    do[#v
      s:s+v[i]]
      r[i]:s]
  r}
```

Sometimes it is necessary to write loopy code, either because the algorithm is inherently iterative, or because we aren't smart enough to find the list solution in the time available. The literature devoted to this topic is vast, and we won't replay it here. Instead, we'll concentrate on an example which shows how, in some cases, a clear looping solution can serve as a stepping-stone on the way to loop-elimination.

We want to write a function which takes a list of equal-length string-lists `L` and a list of strings `S` and returns the row-indices where each string occurs in the same item of each string-list:

```
L:(("this"
    "is"
    "the"
    "first"
    "list")
   ("and"
    "this"
    "is"
    "yet"
    "another"))

S:("is";"an")

search[L;S]
0 4
```

The logical units of the problem are a single list of strings and a single string. We require a function which takes a string-list `x` and a string `y` and tells us which rows of `x` contain `y`:

```
ss:{&0<#:'x _ss\:y}      / y occurs in x@ss[x;y]
```

Now, using **ss**, we write the loopy version:

```
search1:{                        / loopy search
  r:!#*x                         /   initialize result to all rows
  i:0                            /   initialize counter
  do[#x                          /   loop over each string-list
    r@:ss[x[i]@r;y[i]]           /     save result indexed by hits
    i+:1]                        /     increment counter
  r}                             /   indices of rows where all hit
```

At each iteration, we derive a new result by indexing the previous result of **ss** by the new result of **ss**. The arguments to **ss** are the **i**-th string-list indexed by the previous result and the **i**-th string. (Alarm bells should now be going off for K programmers familiar with **+/** (and which are not?)).

Most loop-eliminations are achieved by using some form of each, but this is not one of them*: the result of each iteration depends on the result of the previous iteration, and each-iterations are independent of one another. So the next place to look is **f/**, "over":

An "over" solution will have the form:

```
f/[x;y;z]
```

where **f** will be applied iteratively to three arguments: **x**, the result of the previous iteration; **y**, a list of string-lists; and **z**, a list of strings to search for in corresponding lists of **y**. A good **f** is:

```
{x@ss[y@x;z]}
```

which we get by re-lettering the inner calculation of the **do**-loop and discarding the loop-index **i**.

Now define **search2** as this function over the appropriate values

```
search2:{{x@ss[y@x;z]}/[!#*x;x;y]}
```

Observe that we prime the iteration with **!#*L**, the indices of the first string-list.

```
  search2[L;S]
0 4
```

Derive non-looping solutions from well-designed looping solutions.

* Not strictly true: the problem can be solved with each, but the converging solution using over is both faster and more readable.

Window scripts and window functions

Consider the following simple problem. A window **Add** with two inputs **A** and **B**, both integer atoms, an output atom **C**, and a button **Plus**. When **Plus** is pressed, the contents of **A** and **B** are added together and the result placed in **C**. The contents of **Add** should be arranged thus: **A** next to **B**, **C** below **A** and **B**, and **Plus** below **C**.

Here are three ways to write this in K:

```
\d Add1                    / go to Add1 directory
A:B:C:0                    / initialize atoms
C..e:0                     / result is output
Plus:"C:A+B"               / define button action
Plus..c:`button            / classify as a button
\d ~                       / go to Add1. directory
a:(`A`B;`C;`Plus)          / arrange entries

Add2.A:Add2.B:Add2.C:0     / "Add2" is repeated seven times
Add2.C.e:0
Add2.Plus:"C:A+B"
Add2.Plus..c:`button
Add2..a:(`A`B;`C;`Plus)

Add3:.+((`A;0;)            / windows are dictionaries
        (`B;0;)
        (`C;0;.,(`e;0))
        (`Plus;"C:A+B";,`c`button))
Add3.:.,(`a;(`A`B;`C;`Plus))
```

If the window is defined in a script, follow the form of **Add1**, not **Add2**. The **Add3** code is hardest to read, and should never be used interactively, or in a script. However, if the design calls for encapsulating the window-construction logic in a function, then you have no alternative (since system commands such as **\d** are not available inside functions):

```
add:{                      / makes an Add window
  r:.+((`A;0;)
       (`B;0;)
       (`C;0;.,(`e;0))
       (`Plus;"C:A+B";,`c`button))
  s:.,(`a;(`A`B;`C;`Plus))
  (r;s)}

_d[`Add17`Add17.]:add[]  / make an Add window
```

Reference error hell

Here's a simple way to generate a reference error:

```
   f:{a.b}
   a:10
.k.a.b
term: reference
```

On line 1, `f` is parsed, and K resolves the references to `a.b` by manufacturing the directory `a` in which the single entry `b` is set to nil:

```
   f:{a.b}
   \v
a f
   a
.,(`b;;)
```

On line 2, an attempt is made to assign the name `a` to `10`. If successful, this would wipe out `b`, turning `f` into junk. Therefore, K refuses the assignment.

Observe the effect of this refusal, in the case where `a.b` has been defined before `f`;

```
   a.b:101
   f:{a.b}
   a:10
.k.a.b
term: reference
   a.b
.()
```

A reference error is an error in the application. Don't simply trap and proceed: identify the source of the error in your code and fix it.

A reference error is usually not hard to track down. In almost every case, it arises as the result of using replacement where indexed assignment is sufficient. For example, consider the case where `W` is a window with entries `A` and `B`, `f` is a function which resets `W` to its default state, and `g` is a function which uses `W.A` and `W.B`:

```
   f:{W::.+(`A`B;0 0)}      / makes a W
   f[]                      / make a W
   `show $ `W               / display W
   g:{W.A+W.B}              / add A and B of W
   f[]                      / reset W
.k.W.B
term: reference
```

This is bad design. If `W` is meant to persist throughout the lifetime of the process, then `f` should not replace it wholesale, even with a structurally identical copy. `f` is doing too much. Where `f`

does just enough, there is no reference error:

```
f:{W[`A`B]:0 0}          / resets A and B of W
g:{W.A+W.B}              / add A and B of W
W[`A`B]:f[]              / initialize W
`show $ `W               / display W
f[]                      / reset A and B of W
```

Never use `v:a` when `v[i]:a` will do.

Engine design

Design the engine independently of the screen. The engine is a system of variables, constants, and pure functions. Pay no attention to screens, file I/O, real-time feeds, &c. Deal with those components later. In K, this is easy, since in the final analysis the engine is able to "see" only other variables on the K tree. So you might as well develop the engine entirely in the context of variables and worry later about when, how, and how often those variables get populated by external agents.

Factor the engine into functional relationships between the variables which will hold the data for your application. For example, a simple calculator consists of a set of input variables, a set of output, or result variables, and a set of functional relationships between inputs and outputs. Decide on the proper data-structures for inputs and outputs (atoms or lists, integers or reals, &c.), and design and implement your functions accordingly. Now is also the right time to decide on names for things, and on how variables and functions are to be encapsulated. Some questions to ask are:

Will I need more than one instance of X?
Can X ever go to empty?
What is the default state of X?
Is X an instance of a more general kind of thing Y?

Design the functional components to expect correct data. Don't waste time writing code to handle bad or incomplete data. Push the buck for this job up or out a level. Expect that by the time a function is called, the data will be filtered, defaulted-out correctly, &c. For example, don't pollute a function which expects a list by adding a test to convert atoms to one-element lists. Defer reponsibility for that to whoever calls the function.

Don't error trap unless there is a compelling reason to do so. For example, don't provide for division by zero unless the data can logically ***be*** zero. Plan to error trap as high in the calling tree as you can.

Don't design elaborate result-structures with error-codes and messages. For example, don't design your functions to return lists in which the first element is a return code, and the second the data or a message (a popular, but wrong-headed strategy). Emulate the K primitives, which signal errors.

Don't over-design data-structures. For example, don't use a dictionary of atoms where a list will do. Higher level routines can always be written to convert more complicated forms of data into the simple forms the engine requires.

Be skeptical about tools. Don't design tools prematurely, or with too general a purpose in mind. Keep the tools as simple as the application requires. Don't spend too much time developing the tool-set. Most tools contain functionality which is never used. Avoid using tools which are too heavy for the job at hand. Don't overpopulate the tree with utility functions and variables which your application will not use. If you want one tool out of a pre-packaged set of twenty,

extract that tool and tuck it into your application. If you want a single piece of functionality built into a larger system, speak with the author and learn how to implement it yourself. Remember that in K, ideas are worth more than code.

Minimize the number of moving parts.

Window design

Design and implement the window using a dummy engine. For example, a bond calculator may require functions **p2y** and **y2p** from a financial toolkit. But you can implement the window part of the calculator without having the actual functions ready to hand:

```
\d .fu
p2y:{[p;c;m] .1*p}
y2p:{[y;c;m] 10.*y}
\d .Bond
Price:10#100.
Yield..d:".fu.p2y[Price;Coup;Mat]"
Yield..t:".fu.y2p[Yield;Coup;Mat]"
:
```

Strive to make engine and window one. For example, a simple window which increments and decrements the value of a variable does not need to distinguish the two:

```
\d Win
Value:0
Inc:"Value+:1"
Dec:"Value-:1"
Inc..c:Dec..c:`button
\d ~
a:(`Value;`Inc`Dec)
\d ^
`show $ `Win
```

The window *is* the directory **Win**. The value *is* the variable **Value**. The function engine *is* the pair of variables **Inc** and **Dec**. Ask yourself whether your application is simple enough to avoid making the distinction between window and engine.

Distinguish input and output variables.

Connections

A simple calculator engine consists of a set of input variables, a set of output variables, and a set of functional relationships between inputs and outputs. The engine is presented to the user through a window, whose parts are a subset of the input and output variables.

In connecting the inputs to the outputs and making some of them visible to the user, you must decide how to implement the connections. You have three choices:

the outputs calculate the inputs through dependency evaluation
the inputs calculate the outputs through trigger execution
the user calculates the output results by means of a button

This choice will be guided by considerations of utility, performance, and the logical characteristics of the functional relationships. Here are some examples.

Outputs calculate inputs:

```
\d W1
A:B:0
C..d:"A-B"
C..e:0
\d ^
'show $ 'W1
```

Inputs calculate outputs:

```
\d W2
A:B:C:0
A..t:B..t:"C:A-B"
C..e:0
\d ^
'show $ 'W2
```

User calculates outputs:

```
\d W3
A:B:C:0
C..e:0
Spread:"C:A-B"
Spread..c:'button
\d ^
'show $ 'W3
```

Dependencies

Dependencies should be side-effect free. Dependency definitions are similar to **f** and **g** functions: if **def** is **var..d**, and **var** is invalid, and K needs a value for **var**, it will evaluate **def**, the value for which becomes the value of **var**. K needs the value of a variable each time it is referenced, either by code or because it is on the screen.

We call **var..f** and **var..g** format and validation "call-forwards", since K will call **var..f** to format **var**, and **var..g** to validate incoming candidate values for **var**. Analogously, we can think of **var..d** as the "value call-forward" of **var**.

The evaluation of a dependency should have no side-effects. Logically, the form of a dependency definition is

```
var..d:"foo[A;...;Z]"
```

where **foo** and **A** ... **Z** are the global variables on which **var** depends. If any one of these variables is assigned, or otherwise becomes invalid, then **var** is declared to be invalid. On the next reference, whether through code or because it is on the screen, **var..d** will be evaluated for **var**'s new value.

No dependency should ever have the form

```
var..d:"R:foo[A;...;Z];..."
```

nor should **foo** conceal within itself assignments to the tree, or other side-effects.

Event code

Side-effects should be made explicit. Typically, all side-effects occur in triggers, which are fired on assignment-events, and in screen-event code, which is attached to atttributes such as **`var..k`** and **`var..kk`**. No trigger, or any other event-code locus, should ever have the form

```
"foo[...]"
```

since this style of definition hides the effects of the trigger within **`foo`**. Instead, effects should be broken out and made explicit in the code. What holds for triggers holds also for screen-event code. Namely,

If only one variable is assigned:

```
var..t:"A:..."
```

If more than one, within the same directory **`D`**, and the values are available as a list:

```
var..t:"D[`A ...`Z]..."
```

And, in the general case, where there are multiple side-effects:

```
var..t:"A:...
        :
        Z:..."
```

f and g

Updating the display of a variable automatically updates the value of the variable:

```
A:0
`show $ `A                    / now edit A to be 12

A
12
```

Assignment to a variable which is on the screen causes K to update the display:

```
A:13                          / display is updated to 13
```

There are no explicit display reads or writes in K. Think of the display of **A** as a string made from the value of **A**. Two arrows connect **A** with its display: the output arrow, going from **A** to its display-string, and the input arrow, going from the display-string to **A**. By default, the output arrow is the function **$:** (format); by default, the input arrow is the function **type$**, where **type** is one of **0**, **0.0**, **`**, &c.; i.e., the datatype of (a visual atom of) **A**. Both functions are monadic on the data of **A**.

The **f** attribute is a monadic function of the data which returns a string. If **A..f** is set, then whenever K needs to update the display, it will call **f** on the data to get the new display string. The **g** attribute is a monadic function which takes a string and returns a value. Whenever K needs to update the variable from the display, it will call **g** on the string to get the new value of the variable. For example,

```
A..f:{$x-2}                   / output 2 less than the value
A..g:{2+. x}                  / be 2 more than the input
```

f and g should not contain side-effects. The job of **f** is to produce a string from a value; the job of **g** is to produce a value from a valid input string. If **g** cannot convert the input string to a value, it should signal an error:

```
A..g:{if[0>r:. x;'"negative number"];r}
```

Effects should be extracted from **g** and placed either in the trigger code, or eliminated altogether by making other variables dependencies. That is, never ever do:

```
A..g:{r:checkNum x;B:calc r;r}
```

Instead, keep **g** pure, and then either have **A** calculate **B** or have **B** calculate itself from **A**:

```
A..g:checkNum

A..t:"B:calc ._v"
B..d:"calc A"
```

Fast loads

Loads should take no time.

Suppose you write a script **a.k** which involves an expensive initialization, say by means of a synchronous request for data from a server:

```
.A:(`server;1234) 4:(`retrieve;)
```

You mean to load **a.k** from within the application **b.k**, which looks like this:

```
\l a
\d .B
Show:"`show $ `.A"
Show..c:`button
\d ^
`show $ `B
```

but now **b** takes too long to come up.

So now you decide to improve things by having **Show** load the **a** script:

```
Show:".\"\\l a"
```

This is wrong. Instead, modify the **a** script by moving the retrieval code into a function:

```
\d .a
retrieve:{(`server;1234) 4:(`retrieve;)}
```

and changing the definition of **Show** in **b.k**:

```
\l a
\d .B
Show:"`show$.[`.A;();:;.a.retrieve[]]"
:
```

Composition and each-elimination

Learn to spot expressions with sequential each's like this one:

```
*:'|:'v              / first of each reverse of each v
```

and replace them with function-compositions like this:

```
(*|:)'v              / last of each v
```

In general, seek to replace patterns like

```
f'g'...h'v
```

with

```
(f@g@...@h@)'v
```

Application (@) can be elided when the symbol to the left is that of a K verb:

```
(+f@)'v
```

Here's another example:

```
  |:,':1 2 3 4       / reverse each pair-wise join
(1 2
 2 3
 3 4

  (|,)':1 2 3 4      / (reverse join) pair-wise
(1 2
 2 3
 3 4)
```

Compositions are faster than sequential each's (one iteration replaces many), and easier to read (and code!).

```
(~@!:)'paths         / dictionaries in list of paths
1(~=)':vec           / vec[i+1] differs from vec[i]
```

All the forms of dot

In K, there are several ways to apply a function and index a variable. Underlying them all is the ***ur***-function

```
.
```

and pronounced "dot". Application of the `n`-adic function `f` to a list of arguments `v` of count `n` is expressed:

```
f . v
```

Application of the monadic function `g` to `w` is expressed:

```
g . ,w
```

sugar for which is

```
g @ w
```

Even more simply:

```
g w
```

Brackets are more sugar:

```
f[*v;...;*|v]

g[w]
```

Maximize readability by using the simplest syntax available.

Defaults

Use nil to mean "default value".

In multidimensional indexing, K uses nil (`_n`) to mean "all":

```
m[;2 3]           / columns 2 and 3 of all the rows
```

Follow suit; for example,

```
copy:{[table;fields]
  :
  if[_n~fields;fields:!table]
  :
```

Interpret () as "none".

```
copy:{[table;fields]
  if[()~fields;:()]
  if[_n~fields;fields:!table]
  :
```

If the arguments to a function are not independent, order them left-to-right in dependency-order. That is, argument `i` restricts the choice made by argument `i-1`. In particular, try to arrange things so that if argument `i` is nil, all arguments to the right of `i` are logically nil. For example, in the example above, it would be a mistake to order the arguments,

```
copy:{[fields;table]
:
```

Let nil mean "none" if "none" is the default.

```
foo:{
  if[_n~x;x:()]
  :
```

Projection

Never elide semicolons in a projection. For example, if `f` is a triadic function, the seven possible projections should be written:

```
f[;;]              / not f[]
f[;;c]
f[;b;]             / not f[;b]
f[;b;c]
f[a;;]             / not f[a]
f[a;;c]
f[a;b;]            / not f[a;b]
```

Which each?

Projection and each are more general than each-right and each-left. A function can be projected on any of its arguments, and a function can be applied to each item of many lists. Each-right and each-left apply dyadic functions only, itemwise right or itemwise left.

Any expression involving each-right and each-left can be transformed into an equivalent expression using only projection and each. For example,

```
1 2 3 foo/:10 20 30 40

foo[1 2 3;]'10 20 30 40
```

Typically, expressions involving each-right and each-left are easier to read, and certainly easier to write, than expressions couched in terms of projection and each. For example,

```
a foo/:\:b
```

expands to

```
{foo[x;]'y}[;b]'a
```

Prefer each-right and each-left to projection and each.

What's in a script?

A script-template used by one programmer consists of the following sections:

```
\l subscript                    / load subscripts
:

\e 1                            / global state-settings
:

\d .chunk                       / carve out a chunk of the tree

foo:{                           / define functions
:

Var:...                         / define variables
:

Var..d:...                      / define attributes
:
```

Only top level scripts contain shows, and they come last:

```
`show $ ...
:
```

Some programmers like to keep variable initialization and attribute definition together:

```
A:10                            / A section
A..f:fmt
A..g:val

B..d:"A+10"                     / B section
B..f:fmt
B..e:0

C:19
```

while others prefer to keep them separate:

```
A:10                            / value section
C:19

B..e:0                          / enabling section

A..f:fmt                        / format section
B..f:fmt

A..g:val                        / validation section
```

SAM: a simple application model

SAM is an abstract model of K applications. Think of SAM as having an inner core and an outer layer.

The inner core of SAM consists of variables and constants interconnected by functional dependencies and triggers. All functions, and all dependencies expressed in terms of them are completely side-effect free. All side-effects in the core are explicitly located in triggers. Changes in the state of the core happen only as the result of activity in the outer layer, which in turn is restricted to the form of

variable assignments caused by

- window edits
- radio box check button events
- set messages from other processes (including real-time feeds)

and

code execution caused by

- button presses
- click and double-click events
- close callbacks

Moreover,

variable assignments only

- invalidate other variables
- fire triggers

and

code execution only

- assigns variables
- sends messages to other processes

Subwindows

Compare the behavior of the following two implementations:

```
\d A1
Show:"`show $ `.k.B1"
Show..c:`button
\d ^
B1:20
`show $ `A1
```

Press **Show** to display **B1**, and then quit from the window-manager menu of **A1**.

```
\d A2
Show:"`show $ `B2
Show..c:`button
B2:20
\d ~
a:`Show
\d ^
`show $ `A2
```

Repeat the steps for **A2**. Notice that **B1** does not close down, but that **B2** does. **B2** is a sub-window of **A2**.

Popups and rearrangement

Consider the task of presenting the user with two (or more) windows **A** and **B**. You know that the user will never operate more than one window at a time.

Logically, there are three objects: a menu **M**, window **A**, and window **B**.

In order to avoid polluting the desktop with lots of windows (the Front key is not universally supported, different window managers handle popups differently, &c.), use the following technique:

```
\d .W
M.A:".W..a:`M`A"          / W shows M and A
M.B:".W..a:`M`B"          / W shows M and B
M..c:`button              / M is a menu
\d .W.A                   / A is a subwindow
one:0
two:0
\d .W.B                   / B is a subwindow
V:0
Inc:"V+:1"
Inc..c:`button
\d .
`show $ `W                / show W
```

Press **M.A** to arrange **W** to show **A**; press **M.B** to arrange **W** to show **B**.

Avoid popups.
Minimize the number of windows in an application.

New wine for old bottles

The following rules appear in Kernighan & Plauger's ***The Elements of Programming Style***, second edition, NY: McGraw-Hill, 1978.

Write clearly -- don't be too clever.

Say what you mean, simply and directly.
Use library functions.
Avoid temporary variables.
Write clearly -- don't sacrifice clarity for "efficiency".
Let the machine do the dirty work.
Replace repetitive expressions by calls to a common function.
Parenthesize to avoid ambiguity.
Choose variable names that won't be confused.
Avoid the Fortran arithmetic IF.
Avoid unnecessary branches.
Use the good features of a language; avoid the bad ones.
Don't use conditional branches as a substitute for a logical expression.
Use the "telephone test" for readability.

Use DO-END and indenting to delimit groups of statements.
Use IF-ELSE to emphasize that only one of two actions is to be performed.
Use DO and DO-WHILE to emphasize the presence of loops.
Make your programs read from top to bottom.
Use IF ... ELSE IF ... ELSE IF ... ELSE ... to implement multi-way branches.
Use the fundamental control flow constructs.
Write first in an easy-to-understand pseudo-language; then translate into whatever language you have to use.
Avoid THEN-IF and null ELSE.
Avoid ELSE GOTO and ELSE RETURN.
Follow each decision as closely as possible with its associated action.
Use data arrays to avoid repetitive control sequences.
Choose a data representation that makes the program simple.
Don't stop with your first draft.

Modularize. Use subroutines.
Make the coupling between modules visible.
Each module should do one thing well.
Make sure every module hides something.
Let the data structure the program.
Don't patch bad code -- rewrite it.
Write and test a big program in small pieces.
Use recursive procedures for recursively-defined data structures.

Test input for validity and plausibility.

Make sure input cannot violate the limits of the program.
Terminate input by end-of-file or marker, not by count.
Identify bad input; recover if possible.
Treat end-of-file conditions in a uniform manner.
Make input easy to prepare and output self-explanatory.
Use uniform input formats.
Make input easy to proofread.
Use freeform input when possible.
Use self-identifying input. Allow defaults. Echo both on output.
Localize input and output in subroutines.

Make sure all variables are initialized before use.
Don't stop at one bug.
Use debugging compilers.
Initialize constants with DATA statements or INITIAL attributes; initialize variables with executable code.
Watch out for off-by-one errors.
Take care to branch the right way on equality.
Avoid multiple exits from loops.
Make sure your code "does nothing" gracefully.
Test programs at their boundary values.
Program defensively.
10.0 times 0.1 is hardly ever 1.0.
Don't compare floating point numbers just for equality.

Make it right before you make it faster.
Keep it right when you make it faster.
Make it clear before you make it faster.
Don't sacrifice clarity for small gains in "efficiency".
Let your compiler do the simple optimizations.
Don't strain to re-use code; reorganize instead.
Make sure special cases are truly special.
Keep it simple to make it faster.
Don't diddle code to make it faster -- find a better algorithm.
Instrument your programs. Measure before making "efficiency" changes.

Make sure comments and code agree.
Don't just echo the code with comments -- make every comment count.
Don't comment bad code -- rewrite it.
Use variable names that mean something.
Use statement labels that mean something.
Format a program to help the reader understand it.
Indent to show the logical structure of a program.
Document your data layouts.
Don't over-comment.