

Tacit J and K

by ngn

November 29, 2021

Definitions and Notation

Tacit programming is programming without mentioning function arguments explicitly. Complex functions are constructed from simpler ones through the application of higher-order functions called *combinators*.

Noun-verb syntax is the basic value-level (as opposed to function-level) syntax used in APL, J, and K. A verb is applied to the whole expression on its right and, if possible, also to a single noun on its left. Thus, verbs are prefix monadic $\mathbf{f}x$ or infix dyadic $x\mathbf{f}y$ (by convention we use x, y, z for nouns and $\mathbf{f}, \mathbf{g}, \mathbf{h} \dots$ for verbs even though in K an identifier is always a noun). Noun-verb syntax can be extended with *adverbs* which are postfix and have higher precedence than verb applications, and *conjunctions*—infix and with the same precedence as adverbs. The application of an adverb or conjunction forms a derived verb. In APL adverbs and conjunctions are usually called *operators*.

A *train*, in the context of noun-verb syntax, is a sequence of nouns or verbs (its *carriages*) ending with a verb. Normally, such a sequence is meaningless, as the last verb has nothing on its right to play the role of right argument, but if given the semantics of a combinator, it can serve as a convenient mechanism for tacit programming. All three major array languages implement this idea, but in different ways.

Tacit J

J's and APL's adverbs and conjunctions are a limited form of higher-order functions and can be used as combinators for tacit programming.

Additionally, J treats 2- and 3-trains as the *hook* and *fork* combinators, namely:

$$\begin{array}{lll} (\mathbf{f}\mathbf{g})y \Leftrightarrow y\mathbf{f}(\mathbf{g}y) & (\mathbf{f}\mathbf{g}\mathbf{h})y \Leftrightarrow (\mathbf{f}y)\mathbf{g}(\mathbf{h}y) & (z\mathbf{g}\mathbf{h})y \Leftrightarrow z\mathbf{g}(\mathbf{h}y) \\ x(\mathbf{f}\mathbf{g})y \Leftrightarrow x\mathbf{f}(\mathbf{g}y) & x(\mathbf{f}\mathbf{g}\mathbf{h})y \Leftrightarrow (x\mathbf{f}y)\mathbf{g}(x\mathbf{h}y) & x(z\mathbf{g}\mathbf{h})y \Leftrightarrow z\mathbf{g}(x\mathbf{h}y) \end{array}$$

Longer trains are reduced, starting from the right, to nested forks, possibly ending with a hook, for example:

$$\mathbf{f}\mathbf{g}\mathbf{h}\mathbf{i}\mathbf{j}\mathbf{k}\mathbf{l}\mathbf{m} \Leftrightarrow \mathbf{f}(\mathbf{g}\mathbf{h}(\mathbf{i}\mathbf{j}(\mathbf{k}\mathbf{l}\mathbf{m})))$$

We refer to this interpretation of trains as *J-style*, or *zebra* trains, or *odd-even* trains.

Inserting a monadic verb in a J train is inconvenient, so the language supports a special verb-like token—“[:” , called *cap*, to suppress the left argument of a fork’s middle verb:

$$\begin{aligned} & ([:\mathbf{gh})y \Leftrightarrow \mathbf{g}(\mathbf{h}y) \\ x([\mathbf{gh})y & \Leftrightarrow \mathbf{g}(x\mathbf{h}y) \end{aligned}$$

In the early 2010s Dyalog APL adopted J-style trains, replacing hooks with *atops* and eliminating *cap*:

$$\begin{aligned} & (\mathbf{fg})y \Leftrightarrow \mathbf{f}(\mathbf{g}y) \\ x(\mathbf{fg})y & \Leftrightarrow \mathbf{f}(x\mathbf{g}y) \end{aligned}$$

Neither APL nor J assign meaning to trains that have a noun at an odd 0-indexed position starting from the right. For instance, `1+` and `*2-` give syntax errors.

J-style trains have the following drawbacks:

- The fork combinator is given the most minimal syntax possible—the 3-train, even though it’s not that fundamental or commonly needed.
- Verbs at odd vs even positions are applied differently—to the argument(s) vs to the results from neighbouring verbs:

$$\begin{array}{ccccccc} x(\mathbf{fghijklm})y & \Leftrightarrow & \mathbf{f} \rightarrow \mathbf{h} & \rightarrow \mathbf{j} & \rightarrow \mathbf{l} & & \\ & & \swarrow & \downarrow & \downarrow & \downarrow & \searrow \\ & & x & \mathbf{g} & \mathbf{i} & \mathbf{k} & \mathbf{m} \\ & & & \Downarrow & \Downarrow & \Downarrow & \Downarrow \\ & & & xy & xy & xy & xy \end{array}$$

It’s hard to keep track of that in a longer train, as the factors affecting it lack locality—in order to determine how a carriage is applied, the reader must understand and count all carriages to its right.

- Fork-hook syntax feels like a sublanguage that doesn’t fit in. Grouping carriages in threes is substantially different from ordinary noun-verb syntax and its mirror image—adverb-conjunction syntax.

noun-verb grammar	$\mathbf{e}:\mathbf{N} \mathbf{Ve} \mathbf{NVe}$
adverb-conjunction grammar	$\mathbf{e}:\mathbf{V} \mathbf{eA} \mathbf{eCV}$
fork-hook grammar	$\mathbf{e}:\mathbf{Vf} \mathbf{f} \mathbf{f}:\mathbf{V} \mathbf{VVf}$

Tacit K

K has multiple incompatible versions. For simplicity we limit the examples in this section to K6 as implemented in `ngn/k`.

K functions are first-class values and support up to 8 arguments through M-expression syntax (`f[x;y;...]`). The simplest way to program tacitly is to implement combinators as ordinary functions. For instance, `fork` could be

{[f;g;h;x] g[f x;h x]}
 {[f;g;h;x;y] g[f[x;y];h[x;y]]}

partially applied to **f**, **g**, and **h**. The monadic and dyadic versions must be separate because K functions have fixed *valence* (number of arguments).

K has trains too, but in contrast with J's, they don't form forks. A noun-verb combo creates a partial application called *projection*:

$$(xf)y \Leftrightarrow xfy$$

A pair of verbs (including the noun-verb projections above) forms a *composition* in which the left verb is applied to the result from the right verb. The composition's valence is the valence of the right verb:

$$(\mathbf{fg})x \Leftrightarrow \mathbf{f}(\mathbf{g}x)$$

$$(\mathbf{fg})[x;y] \Leftrightarrow \mathbf{f}(x\mathbf{g}y)$$

Note that **f** and **g** here are meant as verbs, not identifiers **f** and **g**. K's grammar treats all identifiers, parenthesized expressions, and lambdas as nouns. They cannot be applied infix, but dyadic application is possible through an M-expression like $(\mathbf{fg})[x;y]$.

Longer trains follow the same rules right-to-left, so all carriages except the last are interpreted as monadic. The last carriage determines the valence of the entire train. Some examples of K trains are:

$$(1+)x \Leftrightarrow 1+x$$

$$(-*)x \Leftrightarrow -*x$$

$$(%*) [x;y] \Leftrightarrow \%x*y$$

$$(%1+!2*)x \Leftrightarrow \#!1+2*x$$

$$(1-_%*) [x;y] \Leftrightarrow 1-_%*[x;y] \Leftrightarrow 1-_%x*y$$

The syntax of a K train matches the syntax of ordinary K expressions, so its application boils down to removing the parentheses around the train.

K can express a fork through the train

$$\mathbf{g}/(\mathbf{f};\mathbf{h})@\backslash:$$

where $(\mathbf{f};\mathbf{h})$ is a list of the functions **f** and **h**,

@ is the "apply" verb,

\: is the "each left" adverb,

and $\mathbf{g}/$ is "fold"—**g** applied over the pair of results from **f** and **h**.

Hook is

$$\mathbf{f}/1 \mathbf{g}\backslash$$

where $1 \mathbf{g}\backslash$ is one iteration of **g** over the argument, i.e. the pair $(x;\mathbf{g} x)$, and $\mathbf{f}/$ applies **f** between **x** and $\mathbf{g} x$