Tacit Definition

Roger K. W. Hui
Iverson Software, Inc.
33 Major Street
Toronto, M5S 2K9
Canada
Telephone: 416 925 6096

Kenneth E. Iverson
Iverson Software, Inc.
33 Major Street
Toronto, M5S 2K9
Canada
Telephone: 416 925 6096

Eugene E. McDonnell
Iverson Software, Inc.
1509 Portola Avenue
Palo Alto, California
USA
Telephone: 415 321 5260

Introduction

The predominant form of function definition in APL is explicit in the sense that the arguments are referred to explicitly in the sentences that represent the function being defined. For example, if *M* is the character matrix:

```
    Z←SUM Y
    Z←+/Y
```

then *Y* refers explicitly to the argument of the function *SUM* established by `⎕FX M`.

In the dialect J (Defined in References 1 to 3, and summarized in Appendix A), the list `m=. '+/y.'` represents the same function in the sense that `m : ''` provides a function that may be applied directly, as in `m : '' 2 3 4`, or may be assigned a name, as in `sum=. m : ''`. Again the argument is referred to explicitly in the representation.

Because the operator `/` in the expression `+/` produces a derived function, the expression `+/` is a tacit definition of summation that involves no explicit reference to the argument. Moreover, in some dialects (including J), a name can be assigned to the resulting function, as in `sum=.+/` .

The present paper provides an inductive argument that a wide class of explicit definitions can be expressed in tacit form using the facilities of J, and discusses a translator to tacit form that itself occurs as a primitive in J.

The paper concludes with two sections that illustrate the use of tacit programming. Section C shows the relation to traditional APL by providing annotated translations of the first ten items in the FinnAPL Idiom Library [4]. Section D provides definitions for a general inner product and for orthogonal systems. It may be compared with the notation used by McConnell [5], and with the APL treatment of them in Iverson [6].

In J (and in the present paper), we replace the APL terms *function*, *operator with one argument*, and *operator with two arguments* by *verb*, *adverb*, and *conjunction*, respectively.

A. Tacit Programming

In simple cases, a tacit definition is easy to write, and its convenience is evident. For example, a function for summation would be defined tacitly by `sum=.+/` and explicitly by either `sum=. '+/y.' : ''` or `⎕FX 2 7 ⍴'Z←SUM YZ←+/Y '`.

To appreciate the more general use of tacit definition, it is necessary to understand three key notions of J: cells and rank, forks [7], and composition.

Cells and Rank. A *k*-cell of an array is the sub-array defined by its last k axes. For example, if `b` is the 2 by 3 by 4 array:

```
abcd
efgh
ijkl

mnop
qrst
uvwx
```

then `abcd` is a 1-cell of `b`, the table from `m` to `x` is a 2-cell of `b`, and `g` is a 0-cell or *atom* of `b`. A cell of rank one less than the rank of `b` is called a *major cell* or *item* of `b`.

Each primitive function has an assigned rank, and applies to each cell of that rank. For example, ravel has unbounded rank and applies to the entire array:

```
   ,b
abcdefghijklmnopqrstuvwx
```

Moreover, the rank conjunction denoted by " produces a verb whose rank is determined by the right argument of " . For example:

```
   ,"2 b
abcdefghijkl
mnopqrstuvwx
```

Finally, the treatment of reduction and scan differs somewhat from earlier dialects: f/a applies the dyadic case of f between the *items* of a, and f\a applies the *monadic* case of f to *prefixes* of one or more items of a. For example, if a=.i. 2 3 4, then:

```
   +/1 2 3 4 5           +/\1 2 3 4 5
15                    1 3 6 10 15
   a                     +/a
 0  1  2  3          12 14 16 18
 4  5  6  7          20 22 24 26
 8  9 10 11          28 30 32 34
                        +/"2 a
12 13 14 15           12 15 18 21
16 17 18 19           48 51 54 57
20 21 22 23              +/"1 a
                        6 22 38
                       54 70 86
```

Forks. If f, g, and h are verbs, then the (isolated) sequence f g h produces a verb, called a *fork*, whose monadic and dyadic cases are defined by the following diagrams:

```
         g                    g
        / \                  / \
       f   h                f   h
       |   |               / \ / \
       y   y               x y x y
```

For example, a (+ * -) b is (a+b)*(a-b).

Since % denotes division, and #y the number of items of y, then the arithmetic mean or average may be defined tacitly by mean=. +/ % # . Moreover, using the array a=.i. 2 3 4 shown earlier, mean may be applied to obtain the mean of the two tables, the mean of the rows for each of the two tables, and the means of each of the rows of the array:

```
   mean a       mean"2 a        mean"1 a
 6  7  8  9    4  5  6  7    1.5  5.5  9.5
10 11 12 13   16 17 18 19   13.5 17.5 21.5
14 15 16 17
```

Partitioned means may also be obtained. For example:

```
   mean\ 1 2 3 4 5 6
1 1.5 2 2.5 3 3.5
```

Moreover, each of the expressions such as mean"2 and mean\ produces a derived verb, and can therefore be used in further tacit definitions such as m2=. mean"2 and pm=. mean\ .

The arithmetic mean was defined by a fork as mean=.+/%# . The geometric mean is defined similarly by gm=. */ %:~ # , and may also be modified by rank and partition conjunctions and adverbs. The verb %: is the root (as in 3%:8 is 2) and %:~ is the "commuted" root (as in 8 %:~ 3 is 2).

In some cases the effects of the rank conjunction and other facilities of the tacit form are easily achieved in other dialects. For example, the cases +/"k shown earlier can be obtained as +/[i] a for suitable values of i. However, this is not generally the case, as may be seen in trying to mimic cases such as mean"k and mean\ and gm"k in other dialects. To appreciate the convenience of tacit definition, the reader might reproduce in other dialects each of the definitions discussed here.

Compositions. The conjunction & is called *with*, and applies to nouns (variables) a and b as well as to verbs f and g as follows:

```
     a&g y       is a g y
     f&b y       is x f y
     f&g y       is f g y
   x f&g y       is (g x) f (g y)
```

For example, since ^ and ^. denote power and log, respectively, then ^&3 is the cube, and 10&^. is the base-10 logarithm, and 0&< is a proposition to identify positive arguments.

Such compositions can be particularly fruitful in forks. For example (since +. and *. and <. denote *or* and *and* and *floor*):

q=. 0&< *. <&100	test for the interval 0 to 100
r=.] = <.	test for integers
s=. q *. r	test for integer from 0 to 100
t=. <: = < +. =	tautology (<: is less or equal)
ld=. ^.&f % f	log f over f

Other. A number of other constructs in J similarly enhance the utility of tacit definitions. The more important are the *under* (or *dual*), *atop* (a second form of composition), the power conjunction `^:`, and further forms of partitions.

We conclude this section with a few less trivial examples. The `!n` rows of the table `r=. (n-i.n)#:i.!n` are all distinct, and therefore provide a possible representation (called the *reduced* representation) of permutations of order n. The following tacit definitions of `dfr` (*direct from reduced*) and `rfd` provide transformations between reduced and direct representations:

```
   dfr=. /:^:2@,/"1
   rfd=. +/@({.>}.)\."1
   r=. (n-i.n)#:i.!n=.3
   r            dfr r          rfd dfr r
0 0 0        0 1 2          0 0 0
0 1 0        0 2 1          0 1 0
1 0 0        1 0 2          1 0 0
1 1 0        1 2 0          1 1 0
2 0 0        2 0 1          2 0 0
2 1 0        2 1 0          2 1 0
```

Since the elements of the reduced representation indicate the number of transpositions needed to effect a permutation p, the parity of a permutation is `_1^+/rfd p`, and:

```
   par=. ([-:/:^:2) * _1&^@(+/)@rfd
```

yields the parity of its argument if it is a permutation, and *zero* if it is not.

We will call an array of shape `n#n` a solid of order `n`. The indices of such a solid are given by `(n#n)#:i.n#n`, and their parity is given by `sks=. par"1@((##:i.@#)~)`. McConnell [5] calls the result of `sks` a *skew* system of order `n`; interchanging any pair of axes results in a change of sign; for example, `(0 2 1|:b)-:-b=.sks 3`.

Skew systems will be used further in Section D, but we will here show McConnell's procedure for the determinant of a matrix `m`, that is, `+/"1^:(#m) (*//m) * sks #m`, based on the fact that the outer product of its rows `(*//m)` contains all of the products that occur in the computation of the determinant. For example:

```
   [m=.?3 3$10     +/"1^:(#m)(*//m)*sks #m
1 7 4           _225
5 2 0
6 6 9
```

Details of the calculation are shown below (with the tables boxed to provide a compact display):

```
   <"2 */ /m
┌────────┬───────────┬───────────┐
│30 30 45│210 210 315│120 120 180│
│12 12 18│ 84  84 126│ 48  48  72│
│ 0  0  0│  0   0   0│  0   0   0│
└────────┴───────────┴───────────┘
   <"2 sks #m
┌──────┬──────┬──────┐
│0  0 0│0 0 _1│ 0 1 0│
│0  0 1│0 0  0│_1 0 0│
│0 _1 0│1 0  0│ 0 0 0│
└──────┴──────┴──────┘
   <"2(*//m) * sks #m
┌──────┬────────┬─────────┐
│0 0  0│0 0 _315│  0 120 0│
│0 0 18│0 0    0│_48   0 0│
│0 0  0│0 0    0│  0   0 0│
└──────┴────────┴─────────┘
```

The product `(*//m)*sks #m` followed by summation on all axes is a special case of McConnell's generalized inner product to be introduced for handling the vector product in Section D.

B. Translation from explicit to tacit form

Suppose s is a sentence on nouns `x.` and `y.` that results in a noun, and s makes no use of `x.` or `y.` as argument to an adverb or conjunction. We define an adverb `T` which translates s into an equivalent tacit verb. Without loss of generality, assume that s contains no copulae; for if it does, `d=.rhs` (say), recursively replace instances of `d` by `(rhs)`.

If s contains no verbs, s `T` is:

`[`	if s is `x.`
`]`	if s is `y.`
`a"_`	if s is a noun `a` which is neither `x.` nor `y.` (`a"_` is a constant verb with value `a` and infinite rank)

Otherwise, let `f` be the root verb in s; so s is either `f q` or `p f q`, where `p` and `q` are sentences shorter than s and are (by induction) translatable by `T`. Thus s `T` is `f@(q T)` if s is `f q`, and s `T` is `(p T)f(q T)` if s is `p f q`.

In J Version 3.1, `T` is expressed as a case of the colon (`:`) conjunction. The argument is a string. For example:

```
   '((i.#y.)=i.~y.)#y.' T =. : 11
┌───────────────────────────────┬─┬─┐
│┌──────────────┬─┬────────────┐│#│]│
││┌──┬─┬───────┐│=│┌──────┬─┬─┐││ │ │
│││i.│@│┌─┬─┬─┐││ ││┌──┬─┐│@│]│││ │ │
│││  │ ││#│@│]│││ │││i.│~││ │ │││ │ │
│││  │ │└─┴─┴─┘││ ││└──┴─┘│ │ │││ │ │
││└──┴─┴───────┘│ │└──────┴─┴─┘││ │ │
│└──────────────┴─┴────────────┘│ │ │
└───────────────────────────────┴─┴─┘
```

The translator is a straightforward adaptation of the parser. The idea is to parse the sentence as usual, but also apply a set of parallel actions to produce the tacit verb. As described in the dictionary [2], parsing proceeds from right to left, moving successive elements (or their values in the case of names) of the sentence from a queue onto a stack. An eligible portion of the stack is executed and replaced by the result of execution. Eligibility for execution is defined by the rules in Appendix B (reproduced from Table 2 of [2]), and is completely determined by the classes of the first four elements of the stack. The translator maintains a parallel stack. Actions on the stack have parallel actions on corresponding objects on the parallel stack. In particular, when the rules call for applying a verb to its argument(s), resulting in a noun `n`, the parallel action is to compose the verb with tacit verbs that produced the arguments, resulting in a new tacit verb (which, when applied to the original arguments `x.` and `y.`, produces `n`).

The following example illustrates the process. Like the parsing example in Section II E of the dictionary, it uses a line for each step in the parse. Successive columns show the index of the line, the queue, the stack, and the parallel stack. The name `y.` has value `'aba'`.

```
1   ((i.#y.)=i.~y.)#y.
2   ((i.#y.)=i.~y.)#                   'aba'             ]
3   ((i.#y.)=i.~y.)                   #'aba'            #]
4   ((i.#y.)=i.~y.                   )#'aba'           )#]
5   ((i.#y.)=i.~                'aba')#'aba'          ])#]

6   ((i.#y.)=i.                ~'aba')#'aba'         ~])#]
7   ((i.#y.)=                i.~'aba')#'aba'       i.~])#]
8   ((i.#y.)                =i.~'aba')#'aba'      =i.~])#]
9   ((i.#y.)                =f1 'aba')#'aba'       =g1])#]
10  ((i.#y.)                   =0 1 0)#'aba'        =g2)#]

11  ((i.#y.                   )=0 1 0)#'aba'       )=g2)#]
12  ((i.#                'aba')=0 1 0)#'aba'      ])=g2)#]
13  ((i.                #'aba')=0 1 0)#'aba'     #])=g2)#]
14  ((                i.#'aba')=0 1 0)#'aba'   i.#])=g2)#]
15  ((                     i.3)=0 1 0)#'aba'   i.g3)=g2)#]

16  (                     (i.3)=0 1 0)#'aba'  (i.g3)=g2)#]
17  (                   (0 1 2)=0 1 0)#'aba'    (g4)=g2)#]
18  (                     0 1 2=0 1 0)#'aba'      g4=g2)#]
19                       (0 1 2=0 1 0)#'aba'     (g4=g2)#]
20                             (1 1 0)#'aba'        (g5)#]

21                               1 1 0#'aba'          g5#]
22                                      'ab'            g6
```

Notes on the indicated lines:
1) Initially, the queue contains the sentence to be parsed, and the stack and the parallel stack are empty.
2) The value of name `y.` is moved to the stack. The parallel action moves `]` (which produces `y.`) onto the parallel stack.
3) The action moves `#` to the stack; the parallel action moves `#` to the parallel stack.
4) The action moves `)` to the stack; the parallel action moves `)` to the parallel stack.
9) When an adverb is executed (in this case `~`), a temporary verb result is produced. Here, `f1=.g1=.i.~` .
10) The action applies verb `f1` to noun `'aba'`, that is, `i.~'aba'` or `'aba'i.'aba'` (`'`*`aba`*`'⍳'`*`aba`*`'` in earlier dialects), resulting in `0 1 0`. The parallel action composes verb `g1` with the corresponding verb `]`, resulting in a verb `g2=.g1@]` .
15) The action applies the monad `#` to noun `'aba'` (the number of items of `'aba'`), resulting in `3`; the parallel action composes `#` with the corresponding verb `]`, resulting in `g3=.#@]` .
17) The action applies the monad `i.` to noun `3` (`⍳3` in earlier dialects), resulting in `0 1 2`; the parallel action composes `i.` with the corresponding verb `g3`, resulting in `g4=.i.@g3` .
20) The action applies the dyad `=` to nouns `0 1 2` and `0 1 0`, resulting in `1 1 0`; the parallel action composes `=` with the corresponding verbs `g4` and `g2`, using fork and resulting in `g5=.g4=g2`.
22) The action applies the dyad `#` to nouns `1 1 0` and `'aba'` (replicate, `1 1 0/'`*`aba`*`'` in earlier dialects), resulting in `'ab'`; the parallel action composes `#` with corresponding verbs `g5` and `]`, using fork, resulting in `g6=.g5#]`. `g6` is a tacit verb equivalent to the original sentence `((i.#y.)=i.~y.)#y.` .

The translator has other parallel actions not exercised by the example. The complete set is:
a) The Monad and Dyad actions apply a verb to its argument(s). The parallel action may compose the verb with the corresponding objects on the parallel stack, depending on whether they are functions of the original argument (denoted `f` and `g` below) or not (`a` and `b`).

	Parallel Stack	Parallel Action
Monad	`v a`	`v a`
Monad	`v f`	`v@f`
Dyad	`a v b`	`a v b`
Dyad	`a v g`	`a&v@g`
Dyad	`f v b`	`v&b@f`
Dyad	`f v g`	`f v g`

b) If an adverb or conjunction has a noun argument, and the corresponding object on the parallel stack is not a noun, then the original sentence cannot be translated to tacit form. Otherwise proceed as usual.

When a sentence `s` is not translatable, the result of `s T` is the verb `s : ''` (if monadic) or `'' : s` (if dyadic).
c) Both copulae and references to names are cognizant of the corresponding objects on the parallel stack. Indirect assignment is not permitted.

d) When an element **e** of the sentence is moved from the queue to the stack, the parallel action moves `[` or `]` or **e** to the parallel stack, according to whether **e** is `x.` or `y.` or otherwise.

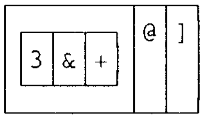
This ready adaptation of the parser emphasizes its simplicity and power. The derivation of the translator makes clear that fork [7] is essential. The composition produced by `@` also has a key role, but the alternative composition `&` on verbs is non-essential; in fact, `f&g` can be defined in terms of fork and `@` :

```
    f&g y.          f g y.           f@g. y.
 x. f&g y.   (g x.)f(g y.)   x.(g@[ f g@])y.
```

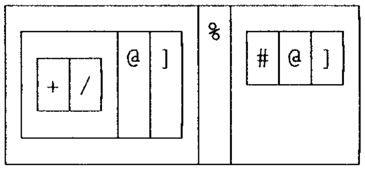
A tacit verb can be re-executed without reparsing the original sentence; therefore, `T` is a compiler.

We conclude this section with further examples of applying the adverb `T`. The following is an excerpt from a session on the system on 1991 4 23. The verb result of `T` is automatically displayed, and at the end of each example is given a (manually constructed) verb which would have the same display:

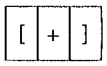
```
   '3+y.' T =. : 11
```



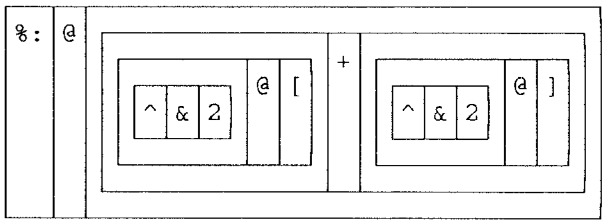
```
3&+@]
   '(+/y.)%#y.' T
```



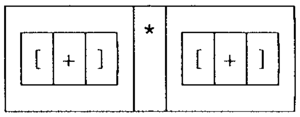
```
+/@] % #@]
   'x.+y.' T
```



```
[+]
   '%:(x.^2)+y.^2' T
```



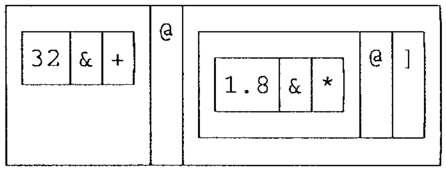
```
%:@(^&2@[ + ^&2@])
   't*t=.x.+y.' T
```



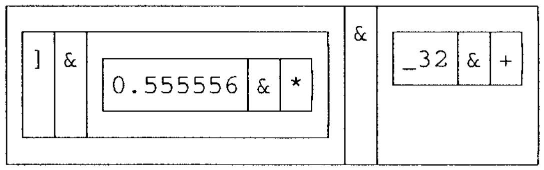
```
[+] * [+]
```

A tacit verb is more clearly structured than its linear explicit equivalent, and is therefore more amenable to automatic manipulation. For example, `Inv=.^:_1` is an adverb which inverts a verb, and:

```
   '32+1.8*y.' T
```



```
32&+@(1.8&*@])'
   '32+1.8*y.' T Inv
```



```
 ]@(0.555556&*)@(_32&+)


   ffc =. '32+1.8*y.' T
   cff =. ffc T Inv

   ffc 0 20 100
32 68 212
   cff ffc 0 20 100
0 20 100
```

C. Ten FinnAPL Idioms

Each of the ten items will begin with an exact quotation from the Library. One or more tacit definitions will then be given, which may include a straightforward translation of the APL expression, a translation that defines a verb that may then be applied to appropriate arguments, and perhaps a solution to an interesting related problem.

Notation of the form *X*←*D*1 occurring in each idiom indicates limitations on the type and rank of the arguments to which the solution applies: *A*,*B*,*C*,*D*, and *I* denote Any, Boolean, Character, Any Numeric, and Integer. Unless otherwise noted, the tacit solutions apply to arguments of any rank and type. Moreover, any tacit solution `f` can be applied to each of the rank-k cells of an argument by using `f"k`.

****

1 `⌊.5×(⍋⍋X)+⌽⍋⍋⌽X` X is D1
Ascending cardinal numbers (ranking, shareable)

```
   X=. 67 70 68 72 67 67 70 65
   <.0.5*(/:/:X)+|./:/:|.X
2 5 4 7 2 2 5 0
     F1=. <.@-:@ (/:@/: + /:@/:&.|.)
     F1 X
2 5 4 7 2 2 5 0
```

Although `F1` is an accurate translation of the idiom as printed, note that (a) the result gives ascending *ordinal* numbers (not *cardinal*) and (b) the rankings are incorrect. An inspection of the argument shows that the lowest score (65) has rank 0, as it should, but that the next lowest score (67), which appears three times, has rank 2, whereas it should have rank 1. A corrected algorithm is the fork `F1a=./:~ i. ]`, in which the indices of the items of the argument are looked for in the sorted argument.

```
    F1a X
1 5 4 7 1 1 5 0
```

Although both of these properly gives equal ranks to equal items, the ranks might also be compressed to a dense sequence, such as `1 3 2 4 1 1 3 0`. The solution to this modifies `F1a` by looking for the indices of the argument in the sorted nub of the argument:

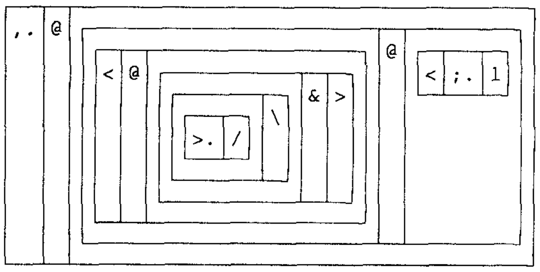
```
    F1b=./:~@~. i. ]
    F1b s
1 3 2 4 1 1 3 0
```

****

2 `Y[A⍳⌈\A←⍋A[⍋(+\X)[A←⍋Y]]]` `X←B1; Y←D1`
Maximum scan (⌈\\) over subvectors of Y indicated by X

The following sequence of definitions shows how the tacit verb F2 may be developed:

```
    Y=. 3 1 4 9 8 2 7 1 0
    X=. 1 0 0 1 0 1 0 0 1
    cut=. <;.1
    X cut Y
┌─────┬───┬─────┬─┐
│3 1 4│9 8│2 7 1│0│
└─────┴───┴─────┴─┘
    maxsc=. >./\
    each=. &.>
    maxsc each X cut Y
┌─────┬───┬─────┬─┐
│3 3 4│9 9│2 7 7│0│
└─────┴───┴─────┴─┘
    ,. maxsc each X cut Y
3 3 4 9 9 2 7 7 0
    F2=. ,.@(maxsc each@cut)
    X F2 Y
3 3 4 9 9 2 7 7 0
```

```
    F2
```


The display of `F2` above illustrates the fact that tacit definitions are "compiled" in the sense that the definitions of component verbs are substituted for them.

Alternatively, `F2a=.,.@(<&(>./\);.1)` . The verbs `F2` and `F2a` are, of course, limited to booleans and numbers, but apply to arguments `Y` of higher rank. For example:

```
   ]Y=. ?9 5$45        X F2 Y

44 32 33 29  3      44 32 33 29  3
28 39 12 19 34      44 39 33 29 34
21 10 12 16  7      44 39 33 29 34
21 40 40  2 40      21 40 40  2 40
22 23 14 44 22      22 40 40 44 40
11  4 42  3 22      11  4 42  3 22
17 12 41 23 20      17 12 42 23 22
42  2 34 34 37      42 12 42 34 37
 5  0 30 39 28       5  0 30 39 28
```

****

3 This idiom differs from #2 only in using min (<.) for max (>.).

****

4 `Y[⍋Y]∧.=X[⍋X]` `X←D1; Y←D1`
Test if *X* and *Y* are permutations of each other

The verb `/:` differs from the dyadic grade in standard APL; the left argument is sorted into an order specified by the right argument. In particular, the form `/:~` uses the duplicate adverb `~` to permit a single right argument to be used also as the left argument.

```
   sort=. /:~
   F4=. -:&sort
   3 1 4 2 F4 1 2 4 3
1
   3 1 4 2 F4 1 2 2 2
0
```

The verb F4 applies to either numeric or character arguments, and to arguments of any rank. For example:

```
   Y=. 'rosy lips and cheeks'
   X=. 'or physics and leeks'

   X F4 Y
1
```

****

5 `X[A[⍋(+\(⍳⍴X)∊+\⎕IO,Y)[A←⍋X]]]` `X←D1; Y←I1`
Sorting subvectors of lengths *Y*

```
   X=.1 6 4 4 1 0 6 6
   Y=.3 3 2
   box=.[ cut~ i.@#@[ e. +/\@(0&,)@]
```

The portion of box beginning at `i.` is a fork with central verb `e.` and box itself is a fork with central verb `cut~`.

```
   X box Y
┌─────┬─────┬───┐
│1 6 4│4 1 0│6 6│
└─────┴─────┴───┘
   X (F5=. ,.@(sort each@box)) Y
1 4 6 0 1 4 6 6
```

****

6 `Y[A[X/⍋(+\X)[A←⍋Y]]]` `X←B1; Y←D1`
Minima (`⌊/`) of elements of subvectors of *Y* indicated by *X*

```
   ]Y=. i.-#X=.1 0 0 1 0 1 0 0 1
8 7 6 5 4 3 2 1 0
   F6=. <./ ;. 1
   X F6 Y
6 4 1 0
```

****

7 `A[⍋(+\X)[A←⍋Y]]` `X←B1; Y←D1`
Grading up subvectors of *Y* indicated by *X*

```
   X (F7=. ,.@(/:each@cut)) Y
2 1 0 1 0 2 1 0 0
```

****

8 `(⍴X)⍴(,X)[⎕IO+A[⍋⌊A÷¯1↑X]] ∆`
`A←(⍋,X)-⎕IO` `X is D2`
Sorting rows of matrix into ascending order

```
   F8=. sort"1
```

****

9 `(⍴X)⍴(,X)[A[⍋(,⍉(⌽⍴X)⍴⍳1↑⍴X)[A←⍋,X]]]`
`X←D2`
Sorting rows of matrix into ascending order

```
      F9=. F8
```

****

10 `(⍋⍋(G+1),⍳⍴⍴X)⍉(Y,⍴X)⍴X`
`G is I0; Y←I0; X←A`
Adding a new dimension after (before in 0-origin, the case in J) dimension G Y-fold

```
   F10=. ] # ,:@[
   m=. ~:&0 { 9&,@-
   $X
2 2 2
   $ X F10 Y=. 3
3 2 2 2
   G=. 1
   $ X F10"(m G) Y
2 3 2 2
   G=.0
   $ X F10"(m G) Y
3 2 2 2
   m 0
9
```

D. Inner Product and Orthogonal Systems

The matrix (or *inner*) product on two matrices M and N can be viewed as an outer product (`M*/N`) in which the last axis of the first argument is "run together " with the first axis of the last argument

```
      0 2 2 1 |: M*/N
```

and summation is performed over the resulting axis:

```
 +/"1 (0 2 2 1 |: M*/N)
```

Inner product as defined by McConnell [5], is a generalization of inner product in which each of a list of axes of the first argument may be paired with each of a corresponding list of axes of the second argument, with summation occurring over each axis that results from the pairings. Such an inner product is provided by the conjunction:

```
s=.'+/"1^:n@(x.&|:@[*"(n=.#x.)/ y.&|:@])'
IP=. s : 2
```

Thus:

```
  [M=.?3 3$20         [N=.?3 3$20
13 13 18            13  0  7
 7 10 16             1  8 13
 0  1 10            11 18 16
  M +/ . * N          M 1 IP 0 N
380 428 548         380 424 548
277 368 435         277 368 435
111 188 173         111 188 173
  (*//M) 0 1 2 IP 0 1 2 (sks #M)
308
   +/^:3(*//M)*sks #M
308
```

The first example shows the use of the general inner product to produce the matrix product; the last examples show its use to compute the determinant by the process developed in Section A. It should be noted that the definition of the conjunction `IP` is explicit, but that it may be *used* in tacit definitions in the normal manner.

McConnell uses a number of interesting inner products with the skew array `e=. sks 3` . In the following examples, the tables are boxed (by <"2) to provide a compact display:

```
   e=. sks 3
   <"2 e 2 IP 2 e
┌──────┬──────┬──────┐
│0 0 0 │ 0 1 0│ 0 0 1│
│0 0 0 │_1 0 0│ 0 0 0│
│0 0 0 │ 0 0 0│_1 0 0│
├──────┼──────┼──────┤
│0 _1 0│0 0 0 │0  0 0│
│1  0 0│0 0 0 │0  0 1│
│0  0 0│0 0 0 │0 _1 0│
├──────┼──────┼──────┤
│0 0 _1│0 0  0│0 0 0 │
│0 0  0│0 0 _1│0 0 0 │
│1 0  0│0 1  0│0 0 0 │
└──────┴──────┴──────┘
   e 1 2 IP 1 2 e
2 0 0
0 2 0
0 0 2
   0 1 2 IP 0 1 2~ e
6
```

The *Complete* inner product is defined as the inner product over the last k axes of both arguments, where k is the minimum of their ranks:

```
s=. '+/"1^:(x. r y.) x.*"(x. r y.)/y.'
cip=.'' :s
```

where `r=. <.&(#@$)`. For example:

```
  M r a=. 1 2 3      M cip a      M cip M
1                  93 75 32    1168
```

The verb

```
   orth=. sks@# cip ]
```

provides a complete inner product with the skew solid, and

```
   north=. !@#@$ %~ orth
```

provides a *normalized* version with the following properties which hold only if a is neither an atom nor a solid (in which case `north a` is an atom):

1. `b=.north a` is orthogonal to a; that is, `b cip a` is *zero*.
2. `north` is self-inverse when applied to any result of `north`.

For example:

```
   a=.1 2 3
   b=.1 5 7
   ;north each ^: 0 1 2 3 4 < a */ b
┌──────────────┐
│1  5  7       │
│2 10 14       │
│3 15 21       │
├──────────────┤
│_0.5 _2 1.5   │
├──────────────┤
│   0 1.5    2 │
│_1.5   0 _0.5 │
│  _2 0.5    0 │
├──────────────┤
│_0.5 _2 1.5   │
├──────────────┤
│   0 1.5    2 │
│_1.5   0 _0.5 │
│  _2 0.5    0 │
└──────────────┘
```

The leading ; is merely to provide a vertical display since the narrow column will not permit displaying the result in a large enough type size.

```
    a cip north a*/b
0
   (a*/b) cip north (a*/b)
0 0 0
```

If `skew=. orth@(*/)`, then `a skew b` is the *skew* or *cross* product of vectors a and b; it is perpendicular to their plane, and its length is the area of the parallelepiped they embrace.

To obtain an approximation to the derivative of a rank-0 function f at a point y, we choose a small "delta" `dt=.1e_6` and evaluate `%&dt (f y+dt)-f y`. To obtain the "partial derivatives" with respect to each element of a higher-rank argument y, we use the increment dt times `inc y`, where `inc=. =/&(i.@$)~`. For example:

```
   <"2 inc i. 3 3
┌─────┬─────┬─────┐
│1 0 0│0 1 0│0 0 1│
│0 0 0│0 0 0│0 0 0│
│0 0 0│0 0 0│0 0 0│
├─────┼─────┼─────┤
│0 0 0│0 0 0│0 0 0│
│1 0 0│0 1 0│0 0 1│
│0 0 0│0 0 0│0 0 0│
├─────┼─────┼─────┤
│0 0 0│0 0 0│0 0 0│
│0 0 0│0 0 0│0 0 0│
│1 0 0│0 1 0│0 0 1│
└─────┴─────┴─────┘
```

Thus the adverb

```
   D=. '%&dt@(x.@(]+*&dt@inc)-x.)' : 1
```

provides the derivatives of a function to which it is applied. For example:

```
   a=.1 2 3
   rev=. |."1
   linear=. +/ . * &(m=.?3 3$10)
   linear a                rev a
21 5 27                 3 2 1
   linear D a              rev D a
3 5 8                   0 0 1
0 0 5                   0 1 0
6 0 3                   1 0 0
```

The derivative of the linear function is, of course, the value of the matrix m that defines it.

The *curl* of a function is orthogonal to the derivative. For example if `F11=.(]*|.)"1` then `orth F11 D a` is `0 _2 0`, in agreement with the curl of the same function in [6]. Consequently, `curl=. 'orth@(x. D)' : 1` defines the curl of a function to which it is applied. Thus, `F11 curl a` is again `0 _2 0`.

REFERENCES

1. Hui, Roger K.W., Kenneth E. Iverson, Eugene E. McDonnell, and Arthur T. Whitney, APL\?, *ACM Quote-Quad* Volume 20, Number 9, July, 1990.
2. Iverson, Kenneth E., The Dictionary of J, *Vector* (Journal of the British APL Association), Volume 7, Number 2, October 1990.
3. Iverson, Kenneth E., *Tangible Math and the ISI Dictionary of J*, October, 1990.
4. *FinnAPL Idiom Library*, Finnish APL Association, 1982
5. McConnell, A.J., *Applications of the Absolute Differential Calculus*, Blackie and Son, 1931.
6. Iverson, K.E., The Derivative Operator, *ACM Quote-Quad*, No. 4, Vol. 9, June 1979.
7. Iverson, Kenneth E., and Eugene E. McDonnell, Phrasal Forms, *APL Quote-Quad*, Volume 19, Number 4, August 1989.

APPENDIX A

This is a brief summary of the J notation used in this paper. Iverson [3] has a complete description. Primitive words are spelled with one or two letters; the second is a period or colon. In what follows, f and g are verbs, and m, n, x, and y are nouns.

Monads

`$y`	shape of y
`#y`	number of items in y
`i.y`	y integers from 0
`~.y`	nub y
`,.y`	itemize y ($,.y is 1,$y)
`;y`	matrix of y
`/:y`	upgrade of y
`{.y`	head of y
`}.y`	behead of y
`[y`	y
`]y`	y
`~.y`	nub y
`,.y`	itemize y (,$,.y is 1,$y)
`<.y`	floor of y
`%y`	reciprocal y
`^.y`	natural log of y
`-:y`	half of y
`|:y`	transpose y
`|.y`	reverse y

Dyads

`x$y`	shape items of y by x
`x,:y`	append itemized x to itemized y
`x[y`	x
`x|:y`	move axes x to tail end
`x e.y`	1 if x in y
`x-:y`	1 if x matches y
`x^y`	x to the y power
`x>.y`	larger of x and y
`x~:y`	1 if x not equal to y

Adverbs

`f~y`	y f y
`f/y`	insert f between items of y
`f\y`	apply f to prefixes of y
`f\.y`	apply f to suffixes of y
`x f/y`	f outer product of x with y
`x f~y`	y f x

Conjunctions

`f"n y`	apply f to rank-n cells of y
`f^:n y`	apply f n times to y
`f@g y`	f g y
`x f@g y`	f x g y
`f&g y`	f g y
`x f&g y`	(g x) f g y
`f&.g y`	(g Inv) f&g y
`x f&.g y`	(g Inv) x f&g y
`m&f y`	m f y
`f&m y`	y f m

Phrasal Form

`(f g h) y`	(f y) g (h y)
`x(f g h)y`	(x f y) g (x h y)

Copula

`=.`	local assignment

Other

`_`	negative sign

APPENDIX B: Parsing Rules

[=(av	**v**	**n**	&	*Monad*	LEGEND
c	**n**	**v**	**n**	*Monad*	a Adv
[=(avn	**n**	**v**	**n**	*Dyad*	c Conj
[=(avn	**nv**	**a**	&	*Adverb*	n Noun
[=(avn	**nv**	**c**	**nv**	*Conj*	p Pron
[=(avn	**v**	**v**	**v**	*Fork*	v Verb
[=(	**v**	**v**	&	*Hook*	[Lmark
np	**=**	**cavn**	&	*Is*	= Is
[=(	**c**	**nv**	&	*Conj*	(Lparen
[=(	**nv**	**c**	&	*Conj*	) Rparen
(	**cavn**	)	&	*Punct*	& Any
&	&	&	&	*Get Next*	