The Role of Composition in Computer Programming

Donald B. McIntyre
Luachmhor, Church Road, Kinfauns, Perth PH2 7LD, Scotland UK
Telephone: 011-44-(0)1738-860-726
e-mail: donald.mcintyre@almac.co.uk

A system is composed of components; a component is something you understand
Professor Howard Aiken, quoted by K.E. Iverson

Abstract

Iverson has greatly enlarged the mathematical notion of function composition and made it available to computer programmers. This paper explains the concept, and uses practical examples to show how concise, powerful programs can be written and read. Alternative forms are given, and ways of overcoming initial difficulties are explained.

The systematic use of composition extends the well-known advantages of APL; namely conciseness, consistency, and generality.

Keywords executable mathematical notation, function composition, composite verbs, adverbs, conjunctions, trains, forks, hooks, gerunds, geometry, Iverson, **APL**, **J**

1 Introduction

Mathematicians form a new function **h** from given functions **g** and **f** by the rule that **h(x) = g[f(x)]**. This composite **h**, which inherits many of the properties of the functions which compose it, is often written **g o f** or **gf**, but sometimes **f o g** or **f g** [18; 21; 22; 2; 20; 36]. We can investigate the properties of composite functions as we can other functions; for instance a numeric function can be differentiated. Examples in mathematical literature seem restricted to monadic functions with scalar arguments. Iverson has increased the possibilities, making function composition the central feature in **J**, his new Executable Mathematical Notation.

Composition is not restricted to *verbs* (functions); new *adverbs* (monadic operators) and *conjunctions* (dyadic operators) can also be composed and used with composed verbs to solve practical problems. The formal rules for composition are given in Iverson's ***Dictionary*** [16], which must be consulted for a complete description of symbols and syntax. There are several introductions to the notation [e.g. 11-13; 16; 23-27; 31].

The purpose of this paper is to help those who try to master the new style of programming by providing annotated examples taken from experience in using and teaching this splendidly concise and versatile notation.

Original format: 8.5 x 11in

2 What is function composition?

Combination of two or more functions creates a composite function. For example, compose a new function by combining the functions **-:** (*halve*) and **-** (*negate*):

```
   (-:@-)  0 1 2 3
0 _0.5 _1 _1.5
```

As usual, expressions in parentheses must be evaluated before being used; here this evaluation is *function composition*. *Negate* applies to the argument, and the conjunction **@** (*atop*) passes the result to *halve*. Because adverbs and conjunctions are evaluated before verbs [16, p.79], the parentheses are not necessary and the same result is given by: **-:@- 0 1 2 3**

A composite function can be given a name; e.g. **h=.+:@+**, which doubles the sum if there are two arguments, and doubles the complex conjugate if there is only one. The example illustrates that the trident (*verb conjunction verb*) is a verb [16, p.82].

Function composition is not, in general, commutative. Taking an example from James [18, p.72-73], define **g=. >:** and **f=. *:** (**>:** is *increment*; ***:** is *square*). If **h=. g@f** and **e=. f@g** then **h 2** is **5** whereas **e 2** is **9**

If **v=. h"0** then **v** is composed by the trident (*verb conjunction noun*) and is therefore a new verb [16, p.82], composed using a verb which is itself composite. The rank conjunction (**"**) restricts the rank of **h** to **0**; i.e. **h** applies to the rank-0 cells of its argument [16, p.75-78].

Composite functions are entities with their own properties; e.g. we may take derivatives of composite numeric functions. If **y=. h x=. i.5**, then **v D.1**

gives the first derivative of the composite function **v**:

```
   v D.1 x  is  0 2 4 6 8
```

h"0 is a verb, **D.** is a conjunction, and the combination (*verb conjunction*) is an adverb [16, p.83]. Consequently **d=. h"0 D.** is an adverb; and because the combination (*noun adverb*) is a verb [16, p.83], **(1 d)** and **(2 d)** are also verbs. They give the first and second derivatives respectively of the composite verb **h**:

```
    x,y,(1 d y),: 2 d y
0 1  2  3  4
1 2  5 10 17
2 4 10 20 34
2 2  2  2  2
```

Another example of a verb composed by a noun and adverb is **amend**, which implements scattered indexing. The noun gives the indices of cells to be amended; **}** is the *amend* adverb [16 p.161; 28;29]:

```
  y=. i.3 4  [  n=. 1 1 ; 2 2; 0 3
  amend=. n}
  97 98 99 amend y
0  1  2 99
4 97  6  7
8  9 98 11
```

A new verb can be composed by bonding a noun to a verb with a conjunction; e.g. **%&180** divides its argument by **180**.

3 Compositions of 2 or 3 verbs

The following (and similar) values confirm that in what follows the composite functions on the left give the same results as execution of the expressions on the right.

```
   f=. %  [. g=. +  [. h=. *
   x=. 2 [ y=. 5
```

[and **]** are verbs that return their left and right arguments respectively. **[.** and **].** are analogous conjunctions [16 p.155,156].

3.1 Composition of 2 verbs used as dyad

```
x g@h y     ↔       g (x h y)   Atop
x g&h y     ↔ (h x) g (h y)     With
x (g h) y   ↔     x g (h y)     A hook
```

3.2 Composition of 2 verbs used as monad

```
  g@h y    ↔ g (h y)     Atop
  g&h y    ↔  g (h y)    With
  (g h) y  ↔  y g (h y)  A hook
```

3.3 Composition of 3 verbs *Forks*

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

The trains in parentheses on the left are *forks*, and the fork contributes the key idea in function composition. Here are some variants:

```
  (] g h) y  ↔    (g h) y
x ([ g h@]) y ↔  x (g h) y
x (] g h) y  ↔  y g (x h y)
x ([ g h) y  ↔  x g ( x h y)
x (f@[ g h@]) y ↔ (f x) g (h y)
```

"Adverbs and conjunctions are executed before verbs, and the left argument of an adverb or conjunction is the entire *verb phrase* that precedes it" [16 p.79]. Hence, although execution proceeds from right to left, a sequence of verbs joined by conjunctions is parsed as if parenthesized from left to right. The consequences of this are easily overlooked. When in doubt, consult the boxed display and use parentheses to control the desired order of execution:

```
   f@g@h          f@(g@h)
+-----------+    +-----------+
|+-----+|@|h|    |f|@|+-----+|
||f|@|g||  | |    | | | |g|@|h||
|+-----+| | |    | | | |+-----+|
+-----------+    +-----------+
```

The boxed display of the verb *tree display* is an example [16, Appendix]:

```
  tree=. 5!:4@<      e=. f@g@h
  tree               tree 'e'
+-----------+              +- f
|+------+|@|<|            +- @ --- g
||5|!:|4|| | |     -- @ --- h
|+------+| | |
+-----------+
```

Because conjunctions have long left-scope and short right-scope, the phrase defining **tree** (*noun conjunction noun conjunction verb*) parses as **((n c n) c v)**, which in turn resolves into (***v c v***), and finally to *verb* [16, p.82-83]. Similar utilities are:

```
   linear=. 5!:5@<     Linear display
   erase=. 4!:55@;:    Erase objects
   setrl=. 9!:1        Set random link
   nc=. 4!:0@<         Name class
```

4 Trains

"The first example of a train was provided by the *fork*, defined by Iverson and McDonnell [17] as a formalization of the informal use in mathematics of expression such as **f+g** and **f-g** to denote the sum and difference of functions" [8, p.74; see also 22 p.43, 57]. It is probably true to say that the fork plays the central role in tacit definition; i.e. in writing programs that make no explicit reference to the arguments.

Iverson defines a train as an isolated sequence which the parsing rules do not resolve into a single part of speech [16, p.81-83]. Meanings are assigned to 25 tridents (trains of 3 elements) and 12 bidents (trains of 2 elements). Eight of these resolve to verbs (functions), and are therefore examples of function composition. Compositions resulting in adverbs (monadic operators) and conjunctions (dyadic operators) are also implemented and prove useful in programming; see Sections 8 and 9.

It is important to distinguish between a simple sentence (as in a typical APL program) and a train, which, though it may look similar, involves composition. A sentence such as **f g h y** (where **f** , **g**, and **h** are verbs and **y** is a noun) can be immediately executed and involves no composition. The sentence **f@g@h y** usually produces the same result, but there is an essential difference: whereas the functions in **f g h y** are applied independently, one after the other, **f@g@h** and the train **(f g h)** are both *unified* functions, with individual properties (such as rank). The functions enclosed within them are no longer independent.

Because expressions within parentheses are evaluated before being applied, the train **(f g h)** is composed as a single entity before execution begins.

It is easily shown that **g h y** is not necessarily the same as **g@h y**, and that **g@h y** is not necessarily the same as the fork **([: g h) y** (**[:** makes **g** a monad).

```
e=. [: g h  [.  f=. g@h
g=. +/      [.  h=. +/"1
y=. i.3 4
```

g h y and **h g y** and **e y** all give **66,** whereas **f y** gives **6 22 38**.

The ranks of any verb, including composite verbs, can be displayed by using the adverb *basic* **(b.)** thus:

g@h b. 0 yields **1 1 1**. The 3 numbers give the rank of the monad, and the ranks of the left and right arguments of the dyad. The conjunction *at* **(@:)**, however, is equivalent to *atop***(@)** except that the ranks of its results are unbounded. Thus if **p=. g@:h** then **p** does not inherit the rank of **h**; and **p y** is **66**.

5 Composition by forks and conjunctions

A fork is a train of 3 verbs; e.g. **mean=. +/ % #** defines the arithmetic mean to be the sum (of the items) divided by the number (of items). This is similar to the mathematician's expression **f + g** [22, p.43, 57]. The central function (or *root*) is a dyad, and takes for its arguments the results of the other two functions. The tree display shows why the word *fork* is appropriate:

```
tree 'mean'
  +- / --- +
--+- %
  +- #
```

Verb trains with an odd number of elements of three or more are evaluated from the right in groups of three as forks. To avoid domain errors, first erase any existing assignments: **erase 'f g h i j k'**

```
  g h i j k
+-----------+
¦g¦h¦+-----+¦
¦ ¦ ¦ ¦i¦j¦k¦¦
¦ ¦ ¦ ¦+-----+¦
+-----------+
```

Many examples in this paper are trains with an odd number of verbs, *easily read as successive forks (tridents) from right to left.*

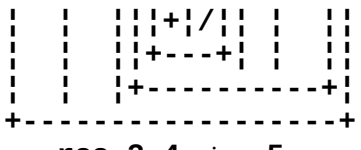
The fork **(i j k)** is evaluated first, and its result is the right-tine of the next fork: **(g h(i j k))**. Verbs within a fork may be either primitive or composite, the latter being unified by conjunctions (commonly **@**) or parentheses.

A hook is a train of just two elements (Sections 3.1, 3.2), thus providing a meaning for trains of even length. Examples of hooks are given in Section 6.

```
  f g h i j k    i.e. f (g h (i j k))
+---------------+
¦f¦+-----------+¦
¦ ¦¦g¦h¦+-----+¦¦
¦ ¦¦ ¦ ¦ ¦i¦j¦k¦¦¦
¦ ¦¦ ¦ ¦ ¦+-----+¦¦
¦ ¦¦ ¦ ¦+-----+¦¦
¦ ¦+-----------+¦
+---------------+
```

For readability I *use spaces to separate the elements in a train*, but *leave no spaces on either side of a conjunction*. This makes it easier to see the existence of trains; conjunctions being the glue that composes the entities (commonly verbs) participating in the trains. Contrast the following definitions of **rss** (square root of sums of squares):

```
  rss=. %:@(+/@*:)      Conjunctions
  rss=. [: %: +/@*:     A fork
  rss
+------------------+
¦[:¦%:¦+----------+¦
¦  ¦  ¦¦+---+¦@¦*:¦¦
```



rss 3 4 gives **5**

Cap (**[:**) completes a fork by making the verb on its right apply monadically. A train of 5 verbs:

```
rss=. [: %: [: +/ *:
```

Read this: *square root* of *sum* of *squares*. It begins on the right with square (***:**) and ends with square root (**%:**), which is its inverse. This pattern often occurs when functions are composed, and the conjunction *under* **(&.)** is provided to make use of it. Thus:

```
rss=. +/&.*:
```

Read this: *sum under square*.

```
g&.h y  ↔  f g (h y)
```

where **f** is the inverse of **h** . If **h** has no inverse, its *obverse* can be defined [16, p.28, 133, 177].

Although the domain of **[:** is empty, it is useful also in other ways – as indeed are verbs that always give the same result (like **0:**), or merely return an argument, (like **[** and **]**); e.g. we might define the absolute value and the residue as **abs=. | : [:** and **res=. [: : |** in order that both **2 abs 3** and **res 3** will give domain errors.

5.1 A truth table as a fork

```
2 2 2 #: i. 2^3
```

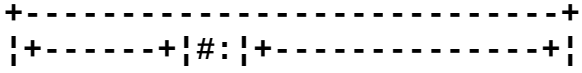
This expression produces an 8 by 3 truth table. An APL programmer will immediately see that the symbols must represent *encode* or *antibase* (**#:**), *iota* (**i.**), and *power* (**^**). An expression such as this is easily written as a fork.

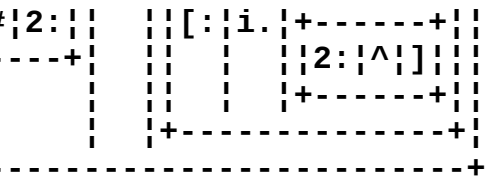
```
tt=. #&2 #: i.@(2&^)   A train of 3 verbs
```

The root verb is **#:**. The other two verbs are composite, constructed with the conjunctions **&** and **@**. I use only necessary parentheses; but when in doubt use fully parenthesized expressions, and then remove parentheses, one set after another, to discover which ones are needed. Boxed displays are an invaluable aid for understanding and correcting syntax. Without parentheses, **i.@2&^** would be interpreted as **(i.@2)&^**, and because **@** does not take noun arguments [16, p.172], this gives a domain error. **tt** can be defined as a train of 5 or even 7 verbs:

```
tt=. #&2 #: [: i. 2&^
tt=. (] # 2:) #: [: i. 2: ^ ]
```

Note, however, that **2:** is not a noun; it is a verb that always returns the result **2**. The boxed display shows how it is parsed as a collection of forks.





This is equivalent to the example **g h i j k** in Section 5, except that **g** is itself a fork: **(] # 2:)** Note, too, how often we find that programs call for an *odd* number of verbs. Forks appear strange at first, but we soon learn to see forks everywhere!

Because they make no explicit reference to any arguments, these are *tacit* definitions. With practice it becomes natural to think directly in this mode. It may help to start with an *explicit* definition, similar to an APL expression. First write a character string with **x.** and **y.** representing the left and right arguments:

```
s=. '(y. # 2) #: i. 2^y.'
e=. 3 : s        Explicit definition
t=. 13 : s       Tacit definition [7; 16, p.129]
linear 't'       Linear display
(] # 2:) #: ([: i. 2: ^ ])
```

The rightmost parentheses are displayed for readability.

Iverson introduced the monadic base-function in 1962 to describe the microprogramming used for indexing in the IBM 7090 computer [10, p. 73]. Although not included in APL\360 [6, p.21], it is implemented in J [16, p.143-144]. Using *antibase-2*, we get a simpler solution either with conjunctions or as a train of 7 verbs:

```
tt=. #:@i.@(2&^)          Conjunctions
tt=. [: #: [: i. 2: ^ ]   Train of 7 verbs
```

6 Hooks

A hook is a train of 2 verbs. Because a train of verbs, such as **(f g h)** and **(g h)**, had not previously been assigned meanings, Iverson was free to specify their interpretation. He defined them to be forks and hooks resulting in composed verbs as described in Section 3.

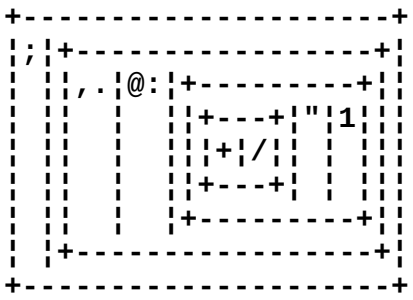
It is important to distinguish **g h y** and **(g h)y**. The first involves no composition: **g** applies to the result produced by applying **h** to the argument. The second, in contrast, requires the resolution of the parenthetical expression before any action involving **y** is taken. In a hook, *the left-function is always a dyad, and the right-function is always a monad.*

6.1 Bordering a table

A hook can append means to the columns or rows of a table, or border a table with row and column sums:

```
h=. , mean      A hook
```

```
   h y=. i.3 4      Append column means
0 1  2  3
4 5  6  7
8 9 10 11
4 5  6  7
   h"1 y   Append means to the rank-1 cells of y
0 1  2  3 1.5
4 5  6  7 5.5
8 9 10 11 9.5
   cols=. ,: +/&.>        A hook with &.
   rows=. ; ,.@:(+/"1)    A hook with @:
   rows
```



Note that although the outermost structure is a hook, this is not to be confused with **f g h i j k** given in Section 5. The latter is a train of *verbs*. The present example contains two conjunctions (**@:** and **"**), one adverb (**/**), and a noun (**1**). There are no forks.

Parentheses are required to prevent the interpretation:

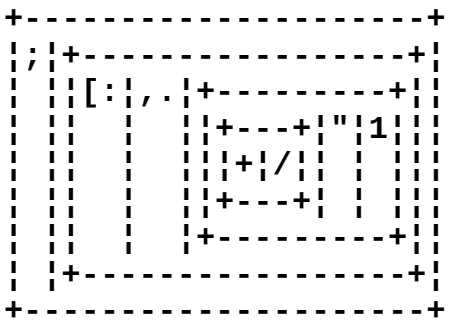
```
   (,.@:+)/"1
```

The *at* conjunction **(@:)**prevents *stitch* (**,.**) from inheriting the rank of **+/"1**, as it would if *atop* **(@)** were used. **$,.@(+/"1) y** is **3 1 1**

$,.@:(+/"1) y is **3 1**

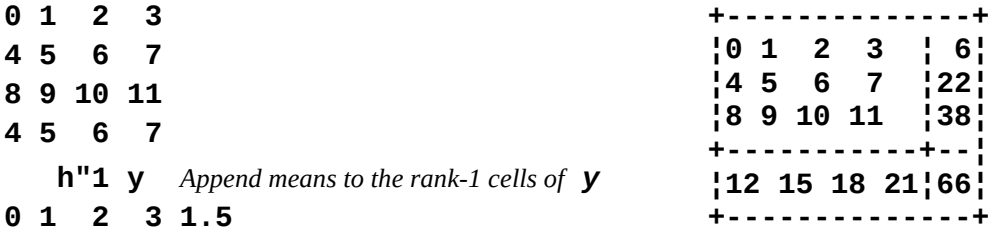
Alternatively, we can write **rows** as a train of 4 verbs, ending on the left in a hook. No parentheses are needed.

```
   rows=. ; [: ,. +/"1
   rows
```



Compose the function **totals**:

```
   totals=. cols@rows
   totals y
```



The dyad in the hook sometimes requires commute (**~**); e.g. select the non-negative items from a list, **y**

```
   setrl 7^5
   y=. 10-~?10#20
   h=. >:&0      or alternatively   h=. >: 0:
   (h y) # y     gives the required selection.
```

The *commute* adverb (**~**) enables us to place the monadic verb on the right and so form a hook:

```
   y #~ h y
   select=. #~ h
```

or use the fork alternative:

```
   select=. (h # ])
```

6.2 A Markov Chain example

m is the transitional frequency matrix for an embedded Markov chain from a Scottish coalfield [5, p.157-161]:

```
 0 11 36 21 52
28  0  4  4  0
34  2  0 45 13
29  1 45  0  3
28 23  9  8  0
```

Begin by replacing the **0**s on the diagonal by **1000**. The indices of the diagonal elements are given by:

```
   diag=. (<0 1)&|:
   ixd=. [: <"1 $ #: diag@i.@$
```

A train of 5 verbs – again an odd number.

```
   ix=. ixd@]
   ix m
```



```
   amend=. [`ix`]}
   n=. 1000 amend m
```

amend is a verb composed from a *gerund* (which has the force of a noun) and an adverb (**}**). The conjunction (**`**) ties three verbs together to compose the gerund [1; 8; and especially 16, p.161]. The three verbs specify

respectively the source, the indices of cells to be amended, and the target.

The diagonal elements are next replaced by:
(the square of the row sums) divided by (the grand sum).

```
   g=. *:@(+/"1) % +/@,      A fork

   5 ": z=. (amend~ g) n     A hook
232  11  36  21  52
 28 199   4   4   0
 34   2 222  45  13
 29   1  45 215   3
 28  23   9   8 211
```

The fork **(g amend])** can take the place of the hook **(amend~ g)**. An iterative process is then used to find the expected transitional probability matrix [5].

7 Function composition in practice

Annotated examples from a variety of programming applications demonstrate composition in practice.

7.1 Surface areas and volumes of spheres

The surface area of a sphere is given by:

```
surface=. 4&*@o.@*:       Conjunctions
surface=. 4: * [: o. *:   Train
```

Read this: ***4*** *times the result of applying* ***o.*** *monadically to the square of the argument.*

```
surface 2 3 4 5
50.2655 113.097 201.062 314.159
```

The volume of a sphere is given by:

```
vol=. 4r3&*@o.@(^&3)      Conjunctions
vol=. 4r3"_ * [: o. ^&3   Train
```

where **4r3"_** is a verb returning 4 over 3

Read this: *(The rational number* ***4*** *over* ***3****) times the result of applying* ***o.*** *to the cube of the argument.*

```
vol 2 3 4 5
33.5103 113.097 268.083 523.599
```

The fork **([: ^ 3:)** can substitute for **^&3**, but perhaps the form given is clearer. Taste changes as possibilities become more familiar.

7.2 Heron's formula for the area of an oblique triangle, given sides a, b, c

$\sqrt{s\,(s-a)\,(s-b)\,(s-c)}$ where **s** is half the semi-perimeter **-:(a + b + c)**

Consider various ways of composing a function **area**

```
s=. -:@(+/)          s is half the sum
s=. [: -: +/         Avoids parentheses
h=. -~ s             A hook commuting the arguments
h=. s - ]            A fork
g=. s , s - ]        A train of 5 verbs
area=. %:@(*/@(s , s - ]))
area=. [: %: [: */ s , s - ]
```

A train of 9 verbs (again an odd number) with no parentheses. Read it: *The square root of the product of* **s** *and the differences between* **s** *and each side.*
However **s** is a verb and not a noun.

```
area 15 17 20 is 124.274
```

7.3 Angles A, B, C of a triangle whose sides are given

```
rfd=. %&180@o.        Radians from degrees
rfd=. o. % 180"_      (180"_ is a verb)
dfr=. rfd^:_1         Degrees from radians
arctan=. dfr@(_3&o.)  Arctangent
arctan=. [: dfr _3"_ o. ]  5 verbs
```

The formula is

```
tan (A % 2) ↔ r % (s - a)
```

with similar relations for angles B and C. **s** as before.

```
r ↔ %:{(s - a) (s - b) (s - c)% s}
```

Take a top-down approach:

```
erase 'n'
r=. %:@(numerator % denominator)
r=. [: %: n % s       Train of 5 verbs
n=. */@(s - ])
n=. [: */ s - ]       Train of 5 verbs
tanA2=. r % s - {.    Train of 5 verbs
A=. +:@arctan@tanA2   Conjunctions
A=. [: +: [: arctan tanA2   5 verbs
```

Immediate execution is then:

```
   A 0 1 2 |."0 1 y=. 15 17 20
46.9722 55.9442 77.0836
```

i.e. *apply* **A** *to a cyclic rotation of* **y**

It is tempting to compose the following function:

```
f=. [: A 0 1 2&|."0 1
```

But **f y** gives a *length error* because the left argument of rotate (**|.**) is fixed as a vector. To discover how to correct this, define the explicit verb:

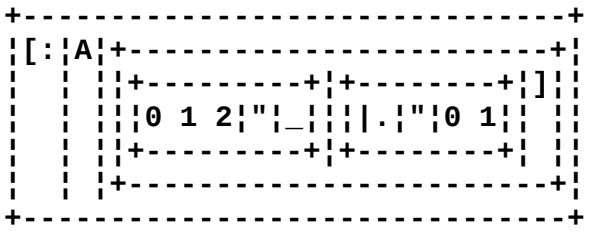
```
e=. 3 : 'A 0 1 2 |."0 1 y.'
e y  is then 46.9722 55.9442 77.0836
```

Convert to tacit form and display the result:

```
   g=. 13 : 'A 0 1 2 |."0 1 y.'
   linear 'g'
[: A 0 1 2"_ |."0 1 ]
```

Note that `0 1 2"_` is a verb but `0 1` is a noun.

```
   angles=. [: A 0 1 2"_ |."0 1 ]
   angles
```



The display shows a train of 5 verbs

7.4 Find substrings common to two strings

Use an equals table to detect substrings in common. Because the *oblique* adverb (`/.`) applies its verb to diagonals descending to the left, we reverse the second string.

```
   x=. 0 1 2  [ y=. 4 5 1 2
   equ=. =/ |.
   x equ y
0 0 0 0
0 1 0 0
1 0 0 0
   ]b=. x [/.@equ y
0 0 0
0 0 0
0 1 1
0 0 0
0 0 0
0 0 0
```

Use the rank-1 cells (rows) of `b` to compress the left argument. All cells in the result are empty except for those containing atoms of any substring in common.

```
   b #&.>"1 x
+----+
|  | |
++-+-|
|  | |
++-+-|
|1|2|
++-+-|
|  | |
++-+-|
|  | |
++-+-|
|  | |
+----+
```

```
   ]m=. >:i.3 4
 1  2  3  4
 5  6  7  8
 9 10 11 12

   ]/. m
 1  0  0
 2  5  0
 3  6  9
 4  7 10
 8 11  0
12  0  0
```

The example on the right shows how `]/.` works, and further exploration using `</.` is helpful. Short diagonals are padded, which leads to an error when the suffix of `x` happens to equal the prefix of `y`:

```
   x=. 1 2 3 [ y=. 2 3 5 4
   x equ y
0 0 0 0
0 0 0 1
0 0 1 0
```

```
   b=. x ]/.@equ y
   b #&.>"1 x
```

This incorrectly reports finding the substring `1 2` instead of `2 3`. The error is prevented by padding the end of the equal table with zeros:

```
   equ=. (=/ |.) ,"1 #@[ # 0:
   x equ y
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 1 0 0 0 0
   x=. 1 2 3 0 5 6
   y=. 1 2 8 0 5 6 3 0
   x (=/ |.) y
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
1 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
```

Note that there are two substrings on one diagonal

A fork uses `equ` to find the substrings:

```
   h=. [/.@equ #&.>"1 [
```

Note that when a verb is used with the adverb *each* (`&.>`) – i.e. *under open* – and its argument is not boxed, the rank-0 cells of the result are always boxed.

```
   ]&.> i.2 3
+-----+
|0|1|2|
+-+-+-|
|3|4|5|
+-----+
```

Ravel after appending a column of empty cells to keep substrings separated, and to ensure that there will be an empty cell at the beginning to define the *fret* for the cut:

```
   g=. ,@((<$0)&,.)
```

Cut using the empty cells as frets:

```
   f=. <;._1
```

Less (`-.`) removes all empty cells:

```
   e=. -.&(<$0)
```

Compose a function using the *at* conjunction **@**

```
   x e@f@g@h y
+-------------------+
¦+---+¦+---+¦+-----+¦
¦¦3¦0¦¦¦1¦2¦¦¦0¦5¦6¦¦
¦+---+¦+---+¦+-----+¦
+-------------------+
```

Finally, *raze* the contents of the boxes:

```
   ss=. ;&.>@e@f@g@h
   x ss y               y ss x
+-------------+     +-------------+
¦3 0¦1 2¦0 5 6¦     ¦1 2¦0 5 6¦3 0¦
+-------------+     +-------------+
```

7.5 Find the pivot of a matrix

This is an essential step in computing the eigenvalues of a symmetric matrix by Jacobi's method [28; 25]. The pivot is the largest absolute value in the upper triangle. We are to find its indices and place certain computed values within an identity matrix at positions determined by the location of the pivot. We are not concerned here with the computation of the values to be placed in the matrix; we simply use the values 96 97 98 99.

```
   utm=. [: , </~@i.@#   Upper triangle mask
   ut=. utm # ,          Upper triangle
ixp=. $ #: (i. >./)@|@ut { utm # ,@i.@$   Indices of pivot
   pivot=. <@ixp { ]     Signed value of the pivot
```

The permutations and indices for amend are:

```
   perm=. 0 0&{ ; ] ; |. ; 1 1&{
   ixa=. perm@ixp@{.@]
```

The argument for **ixa** is **y,: = i. # y**

```
   imat=. {:@]           Amend the last item
   amend=. [`ixa`imat }  Gerund
```

Create a test matrix:

```
   setrl 7^5
```

For **smatrix** see Section 7.6.

```
   t=. smatrix 100%~ 50-~ ?6 6$100
   y=. _0.5 (0 0; 2 4; 4 2)} t
   6.2 ": y
 _0.50  0.12 _0.11 _0.09  0.11  0.18
  0.12  0.53  0.08 _0.11  0.08 _0.23
 _0.11  0.08  0.45 _0.15 _0.50  0.23
 _0.09 _0.11 _0.15  0.11 _0.12 _0.05
  0.11  0.08 _0.50 _0.12  0.59  0.40
  0.18 _0.23  0.23 _0.05  0.40  0.64
   pivot y
_0.5
   ixp y
2 4
   96 97 98 99 amend y ,: =i.#y
1 0  0 0  0 0
0 1  0 0  0 0
0 0 96 0 97 0
0 0  0 1  0 0
0 0 98 0 99 0
0 0  0 0  0 1
```

In studying this example it will be helpful also to execute the following:

```
    utm y
    ut y
    ixa y,:=i.#y
    imat y,:=i.#y
```

7.6 Determinants and the area of a triangle

When two functions are composed as **u . v** this is called the *dot product*. The *inner product* (matrix multiplication) and the *determinant* are two important examples:

ip=. +/ . * and **det=. -/ . ***

(Note that a space between **/** and **.** is required)

The inner product has many applications. It can, for example, be used in a fork to produce a matrix that is symmetric about its main diagonal, as required for Jacobi's method of computing eigenvalues.

```
    smatrix=. |: ip ]
    smatrix i.3 4
 80  92 104 116
 92 107 122 137
104 122 140 158
116 137 158 179
```

Table **t** gives the co-ordinates of points **a, b, c** at the corners of a triangle. If this table is bordered by a column of 1s, the determinant of the resulting 3 by 3 matrix is twice the area of the triangle [19, p.3].

```
a=.0.5 1.25 [ b=.4.4 2.45 [ c=.2.6 0.2
    (,"1 1:) t=. > a;b;c      A hook
0.5 1.25 1
4.4 2.45 1
2.6  0.2 1
    area=. -:@det@(,"1 1:)
```

The determinant (and hence the area) is positive or negative depending upon the sense of the circuit; positive if counterclockwise and negative if clockwise [19; 33].

A.F.Mobius (1827) was the first to recognize the geometrical significance of the sign of the determinant, an idea that proved fundamental to many of his important discoveries. His work was greatly extended by Hermann Grassmann [19, p.16-20; 9].

```
   area > a;b;c       area > a;c;b
_3.3075              3.3075
```

8 Composition of Adverbs

The sentence **f&.g y** means: first compute **g y**, then apply **f** to the result, and finally apply the function that is inverse to **g**. Read this: *f under g*. In this paper we use **+/&.*: +/&.> #&.>]&.> ;&.> ,&.>** and **area&.>** These are all examples of the train **v c v** (*verb conjunction verb*) that composes a new *verb*. We have also seen that a fragment can stand on its own; as in **(&.>)** and **("1)** and **(D.1)**; these are new *adverbs* composed by the trains **c v** (*conjunction verb*) and **c n** (*conjunction noun*).

Hui and Iverson have defined 6 tridents (trains of 3) and 7 bidents (trains of 2) that compose adverbs; and 13 tridents and 1 bident that compose conjunctions [8, p.76;16, p.82-83].

_9: to **9:** are *constant functions* that give the results **_9** to **9**. But any noun can be made into a constant function by the *adverb* **a=. "_**

```
   g=. [: i. 6:
   h=. (i.2 3) a
   f=. g : h
   f 7
0 1 2 3 4 5
   8 f 10
0 1 2
3 4 5
```

Like verbs, adverbs and conjunctions can be defined explicitly. Iverson has given a number of examples [e.g.12, p.113; 13, p.27, 33; 14, p.26, 39, 47, 48, 104; 15, p.80; 16, p.19, 20, 41, 56, 58). In one of these he defined an adverb for Newton's method of computing a root by means of the derivative [14, p.104].

Thomson has published an interesting extension of this, which includes finding the roots of multivariable functions and non-linear curve fitting [35].

Consider the polynomial with one variable:

```
(1*x^2) + (_1*x^1) + (_2*x^0)
```

Its tacit definition is: **ff=. #.&1 _1 _2 "0**

If **x=. 0 1 2 3** its value is **_2 _2 0 4**, which shows that **2** is a root; i.e. **ff 2** is **0**.

Because derivatives are obtained by the *conjunction* **D.** the first derivative is given by the *adverb* **D=. D.1** and a Newton-Raphson step is therefore:

```
x - (ff x) % ff D x
```

or **_2 3 2 2.2**

This is easily recast into a verb in either explicit (**e**) or tacit form (**t**):

```
e=. 3 : 'y. - (ff y.) % (ff D y.)'
   t=. ] - ff % ff D
```

Three applications of **t** :

```
   t t t x
```

or **t^:3 x**

_1.01176 2.01176 2 2.00005.

The limit, given by **t ^:_ x**, is **_1 2 2 2**

An *adverb* can be used whenever an expression containing a verb would be applicable to other verbs. The transformation is easily written in explicit form. The **1** preceding the **:** (and which must be separated from it by a space) refers to the valence of an adverb:

```
   Newton=. 1 : '] - x. % x. D'
```

The **x.** represents the function that will be the argument to the adverb **Newton**.

] stands for the noun that will be the argument to the newly composed verb. The tacit form of the adverb is created in a similar way, but is not so easily interpreted:

```
   t=. 11 : '] - x. % x. D'
   linear 't'
(([.+) ([. % ([. D))) (]`-`) \
```

A train of adverbs composes a new adverb; so we define *limit* as an adverb, and then create the adverb **N**

```
   limit=. ^:_
   N=. Newton limit
```

Trying 6 different starting points, roots are found at **_1** and **2**

```
   ff N _2 + i.6
_1 _1 _1 2 2 2
   ff _1 2   Verifies that these are indeed roots.
0 0
```

As Thomson shows, the problem of finding roots when there are several variables is solved in a similar way. The diagrams in his paper [35, p.213] are very helpful in understanding the motivation, and should be consulted. He begins with two functions, **f** and **g**, which I express as verb trains as follows:

```
   f=. [: <: *:@{. * {:
   g=. 2&o.@{. +  1&o.@{:
   h=. f , g       A fork
```

Taking starting points on the **x-y** plane, the first step in the iterative procedure is given by any of the following (a) Immediate execution, (b) a verb in explicit definition, (c) a verb in tacit definition, and (d) by an adverb.

```
   y=. 5 0
(a) y - (h y) (%.|:) h  D y
(b) e=.3 : 'y. - (h y.)(%.|:)h D y.'
(c) t=. ] - h (%.|:) h D
(d) a=. 1 : '] - x. (%.|:) x. D'
```

With 6 different starting points, we find the co-ordinates of the 4 points of intersection shown on Thomson's **x-y** graph**.** Our purpose is to show how adverbs can be composed. The choice of starting points and the validity of the computed results are not relevant here.

```
y=.>5 0;1 1;1 4;0.5 4.5;0.5 4.8;1 5
  7.2": y,"1 h a limit ("1) y
 5.00   0.00   4.67   0.05
 1.00   1.00   1.86   0.29
 1.00   4.00   0.49   4.23
 0.50   4.50   0.49   4.23
 0.50   4.80   0.44   5.15
 1.00   5.00   0.44   5.15
```

9 Composition of Conjunctions

A *conjunction* can be used when an expression containing *two* verbs is generalized so that other verbs can be substituted. Defined adverbs and conjunctions are miniature computer programs with both verb and noun arguments. The following example, which is an expanded version from Iverson [14, p.36], employs two defined conjunctions. The object is to plot one monadic function against another.

First define the two functions: **f** is the sine; in this case **g** merely returns its argument, but in practice it can be replaced by another of interest; e.g. Iverson uses the hyperbolic cosine [14, p.80].

```
   f=. 1&o. [. g=. ]
```

s gives the dimensions of the plot, and **x** the values of the argument.

```
   s=. 10 20 [ x=. -:i.8
```

To scale the y-axis in a descending direction, execute function **f** and subtract each resulting value from the largest value. The smallest value is then zero. The obvious way to do this is to apply the *verb* **u0** *atop* **f**, but in order to illustrate another feature of the language we define the *adverb* **u** instead. The system distinguishes **u0** and **u** for us by their *name classes* [16, Appendix].

```
   u0=. >./ - ]          A verb
   u=. u0@               An adverb
   (u0@f x) -: f u x     These match
```

Now scale the **x** -axis so that its smallest value is zero.

```
   v0=. ] - <./          Verb
   v=. v0@               Adverb
   z0=. (f u x) ,: (g v x)
```

Parentheses on the right are introduced for clarity.

z0 is a table with two rows; the first gives the values along the y-axis and the second values along the x-axis.

Scale them from 0 to 1 by dividing the rows by their biggest values.

```
   a=. z0 % >./"1 z0   Immediate execution
   scale=. ] % >./"1   A fork
```

The following then match:

```
   a -: scale (f u x) ,: (g v x)
```

Gather these procedures into a *conjunction*:

```
 N2=. 2 : 'scale@(x. u ,: y. v)@]'
```

The definition begins with a left argument of 2, which is chosen because this is the valence of a conjunction.

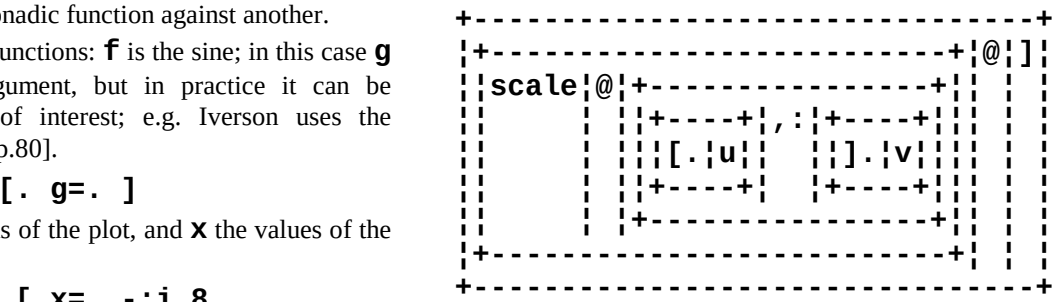
The following then match:

```
   a -: f N2 g x
```

N2 is the explicit definition of a *conjunction* that lets us follow the same procedure when providing different pairs of functions as arguments. **x.** and **y.** refer to these (monadic) arguments.

A tacit definition is also possible, but great care must be taken to include parentheses. It is important to display the boxed form to be sure that the definition has been written to give the desired parsing.

```
   N=. scale@(([. u) ,: (]. v))@]
   N
```



Because explicit definitions are close to the form used in immediate execution, they are easily adopted by programmers. In learning to write tacit definitions, use a left argument of 12 for automatic translation [7, p.204-206; 16, p.129].

```
N12=. 12 : 'scale@(x. u ,: y. v)@]'
```

A similar facility is available for verbs and adverbs by using 13 and 11 respectively.

The values of x and y are then scaled: multiplied by the decremented values of **s** and rounded. The reason for decrementing **s** is that whereas **s** gives the *number* of rows and columns, the axes have an *origin* of **0**. Finally the scaled data are encoded into linear indices within the array **(i.s)** and used to select one or other of the characters specified in **h**.

```
   round=. <.@+&0.5
   r=. [ #. [: |:@round <:@[ * ]
   ]b=. s r a
140 63 25 8 31 74 136 199
   h=. {&'.*'
   ]z=. h (i. s) e. b
```

```
........*...........
.....*.....*........
....................
...*..........*.....
....................
....................
....................
................*...
*...................
....................
...................*
```

Conjunction **VS** uses conjunction **N** and the defined verbs **h** and **r** to produce a plot of **y** against **x**. Noun-arguments are given by *verbs* **[** and **]**, while verb-arguments are given by *conjunctions* **[.** and **].**

```
VS=.12 : '[: h i.@[ e. [ r x. N y.'
     z-: s f VS g x.   These match
```

A fuller plot is given by:

```
     25 90 f VS g 4%~ i.60
```

10 Geometry and Graphics

Geometrical examples in 2- and 3-dimensions are readily visualised. The principles illustrated are of rather general applicability, especially for graphics.

10.1 A hidden line problem: classify points by their signed distance from a given line:

Given two points **a** and **b** defining a line, and a third point **c** on one side of it, classify a collection of points into three groups: those on the same side as **c**; those on the other side; and those collinear with **a** and **b**.

```
a=.0.5 1.25 [ b=.4.4 2.45 [ c=.2.6 0.2
```

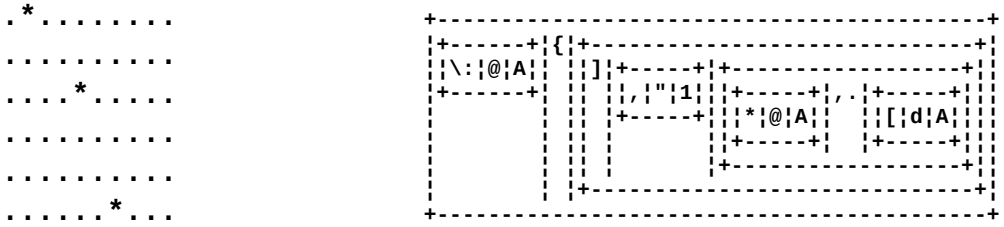
The collection of points is given by table **p**.

```
p=. > 3.5 5.1; 7 2.2; 1.3 3.6;
      3.3 2.9; 0.2 _0.5; 5.7 3.2;
       6 _1.1; 6 2.942
```

Combine **a** and **b** with each point in turn, and find the areas of all triangles so formed. The perpendicular distance from the apex of a triangle to its base is found by doubling the **area** and dividing by the length of the base. **class** appends the sign of the area and the distance, and sorts the output by the signed size of the distance.

```
  A=. [: area"2 ,"2 1
  base=. [: +/&.*: -/
  d=. +:@] % base@[      Distance to the line

  class=. \:@A { ] ,"1 *@A ,. [ d A
```



When reading this definition, first note the conjunctions, and then count the number of forks.

```
  z=.(a,:b) class c,p,-:a+b
  5.2 9.3 3 9.4": z     Format the table
3.50    5.100  1   2.7975
1.30    3.600  1   2.0108
3.30    2.900  1   0.7536
5.70    3.200  1   0.3345
2.45    1.850  0   0.0000   -:a+b
6.00    2.942 _1  _0.0003
7.00    2.200 _1  _1.0036
0.20   _0.500 _1  _1.5844
2.60    0.200 _1  _1.6211   Point c
6.00   _1.100 _1  _3.8636
```

. Three points lie on the same side of the line **a–b** as **c**; four lie on the other side; one is nearly collinear with **a**–**b**.

10.2 Complex numbers: easy and useful

"Knowing that if you double a force you double the vector that represents it, Hamilton looked on **2** *times* as the operator that doubles: it is a special case of what he called a *tensor*, an operator that stretches (not to be confused with the modern use of the word.). In the same way **-1** *times* is a reversor. Moreover if √**2** *times* is applied twice it doubles; and if √**-1** is applied twice it reverses.

Consequently **i** *times* (where **i** is √**-1**) is a versor, or operator that rotates a vector without changing its length; it is taken as producing a counter-clockwise rotation of 90°. Application of **2i** *times* would then be the composition of a rotation, a stretch, and a reversal)." [27, p.568]. Using the new notation:

```
   2&* 1 2              _1&* 1 2
2 4                  _1 _2
   g=. ] * [: %: 2:
   g 1 2                g g 1 2
1.41421 2.82843      2 4
   h=. ] * [: %: _1:
   h h 1 2
_1 _2
```

A complex number is simply the unification of the co-ordinates of the **x** and **y** axes. Mathematicians normally use the symbol **i** to join what are (unfortunately) called the *real* and *imaginary* parts. There is nothing imaginary about so-called imaginary numbers. Sylvester refused to follow the convention and used the symbol **θ** instead. Iverson uses **j**; e.g. **p=. 1j2** represents the point with co-ordinates **1** on the **x**-axis and **2** on the **y**-axis.

```
   2&* p                _1&* p
2j4                  _1j_2
   g p                  g g p
1.41421j2.82843      2j4
   h p                  h h p
_2j1                 _1j_2
```

A unit vector at an angle of **60°** from the **x**-axis, and a vector length 2 with an angle of 1 radian, are written:

```
   1ad60                2ar1
0.5j0.8660254        1.0806j1.68294
```

Separating the two parts:

```
   +. 1ad60
0.5 0.8660254
```

The monad **r.** gives the co-ordinates of unit vectors; i.e. points on the unit circle. The dyad **r.** converts from polar to Cartesian co-ordinates:

```
   r. rfd 60            2 r. rfd 60
0.5j0.8660254        1j1.73205
```

The hypotenuse of a right-angled triangle with sides **3** and **4** is therefore:

```
   +/&.*:@+. 3j4
5
```

10.3 The area of a polygon

First compute the co-ordinates of the vertices of a regular hexagon. **1j0** is a unit vector along **x**, i.e. (**1 0**). To reverse it to **_1j0** in three steps requires not the square root of **_1** but the cube root. The first step is therefore to:

```
   +. a=. 3%:_1
0.5 0.8660254
   +. a^0            +. a^3
1 0               _1 1.22461e_16
```

Three steps take us only half way round; six steps complete the regular hexagon:

```
   ,. +. a^i.6
   1           0
 0.5   0.8660254
_0.5   0.8660254
  _1 1.22461e_16
_0.5  _0.8660254
 0.5  _0.8660254
```

The general case is then:

```
   poly=. (-: %: _1:) ^ i.
   polygon=. ,.@+.@poly
   hex=. polygon 6
   polygon 4
          1           0
 6.12303e_17           1
         _1 1.22461e_16
_1.83691e_16          _1
   polygon 5
       1          0
 0.309017  0.9510565
_0.809017  0.5877853
_0.809017 _0.5877853
 0.309017 _0.9510565
```

To compute the area of a polygon, choose any point on the plane, and from it complete the set of triangles with this point as a corner and a side of the polygon as a side of the triangle. Take the arbitrary point: **p=. 4 _3**

To include all sides, the cycle must be complete:

hex , {. hex *This is a hook* **(, {.)**

The 6 sides are defined by boxed pairs of corners:

```
   y=. 2 <\ (, {.) hex
```

Bond the point; append it to each of the sides in turn; so completing 6 triangles:

p&,each y where **each=. &.>**

The train (*conjunction verb*) gives an *adverb*. The adverb **each(** or *under open*) opens each box, applies its verb, and then closes the box. Each of the boxes defines a triangle, and the sum of the signed areas of these triangles is the area of the polygon:

```
+/ > area each p&,each 2 <\ (, {.) hex
2.598
```

or, using **&.>**

```
+/ > area&.> p&,&.> 2 <\ (, {.) hex
```

or, using the *rank* conjunction instead of boxes:

```
   +/area"2 (2) p&,"1 2\ (, {.) hex
```

The verb **u** in the phrase **a u\ y** is applied (monadically) to each infix of length **a**. Because **a** is the argument of **u** (and not of **u),** the verb **u** cannot have a left argument. **p&,"1 2** is monadic, thus:

```
s=. '+/ > area each p&,each 2 <\ (,
{.) y.'
    e=. 3 : s
    e hex
2.59808
   T=. 13 : s        Convert to tacit definition
    linear 'T'
[: +/ [: > [: area&.> [: 4 _3&,&.>
2: <\ (, {.)
```

Build the triangles by composing simple functions:

h=. 2: <\ ] , {. and **p&,&.> h hex**

We can change **p** only by freeing it from the verb to which it is bonded. It can then be made an argument to the fork:

```
    g=. [ ,&.> h@]
```

Because the arguments must be in agreement [16, p.78; 29, p.137-140), **p** must be boxed: **p=. <4 _3**. The boxed co-ordinates of the corners of the triangles are then given by **p g hex**. It remains to open the boxes; determine the areas; and sum the areas.

```
    f=.  area&.>@g
```

>@area&.>@g is parsed as **(>@area)&.>@g**

Parentheses are not needed, however, if we write a fork:

```
    poly=. [: +/ >@f
    p poly hex
2.59808
```

Does the position of the point make any difference to the result? Try a series of points simultaneously:

```
     p=. <"1 i. 6 3 2
```

The result of **h hex** is a vector of 6 boxes, each 2 by 2, and **p** is a table of 6 by 3 boxes, each box containing a vector of 2 elements. Thus **p g hex** is a 6 by 3 array of boxes, each with a 3 by 3 matrix defining a triangle:

```
   p g hex
+-----------------------------------+
|  0     1  |  2     3  |  4     5  |
|  1     0  |  1     0  |  1     0  |
|0.5 0.866  |0.5 0.866  |0.5 0.866  |
+-----------+-----------+-----------+
     ...         ...         ...

+-----------+-----------+-----------+
|  30    31 |  32    33 |  34    35 |
|0.5 _0.866 |0.5 _0.866 |0.5 _0.866 |
|  1     0  |  1     0  |  1     0  |
+-----------+-----------+-----------+
```

This is the general case, but we need a special case in which all values in a given column are the same: e.g.

```
     p=. 6 3$ 4 _3; 0 0; 1 2
```

The result of **p g hex** is again a 6 by 3 table of boxes, but this time each column specifies the 6 triangles formed by *one* of the 3 points together with *each* of the 6 sides in turn, and this is what we want:

```
    p g hex
+-----------------------------------+
|  4    _3  |  0     0  |  1     2  |
|  1     0  |  1     0  |  1     0  |
|0.5 0.866  |0.5 0.866  |0.5 0.866  |
+-----------+-----------+-----------+

     ...         ...         ...
+-----------+-----------+-----------+
|  4     _3 |  0     0  |  1     2  |
|0.5 _0.866 |0.5 _0.866 |0.5 _0.866 |
|  1     0  |  1     0  |  1     0  |
+-----------+-----------+-----------+
    area each p g hex
+-------------------+
|_0.549|0.433|_0.5  |
+------+-----+------+
|1.933 |0.433|_0.567|
+------+-----+------+
   ...   ...   ...
+------+-----+------+
|_2.049|0.433|0.5   |
+-------------------+
```

The numbers in the middle column are all the same, because in that case the "arbitrary" point is at the origin, from which all corners of the hexagon are equidistant.

The area of the polygon is the sum of the areas of the triangles in any column of **p g hex**:

```
    +/ > area each p g hex
2.59808 2.59808 2.59808
```

Composing a function:

```
    poly=. [: +/ >@(area each@g)
    poly=. [: +/ [: > area&.>@g
    p poly hex is 2.598 2.598 2.598
```

We see that the position of the arbitrary point does not affect the value given for the area of the polygon.

Finally we show that the area is unaffected by rotation. Create a rotation matrix for a random rotation angle.

```
    setrl 7^5          Set Random Link
    y=. 100 %~ ?10000
```

```
13.15
    rot=. 2 1&o. ,: _1 1"_ * 1 2&o.
    rotate=. rot@rfd
```

Apply this rotation; make an arbitrary translation; and confirm that the area is unchanged:

```
    2 3 +"1 hex +/ .* rotate y
2.97378  3.2275
2.28987  3.95707
1.31609  3.72957
1.02622  2.7725
1.71013  2.04293
2.68391  2.27043
   p poly hex
2.59808 2.59808 2.59808
```

10.4 The line of intersection of two planes

This is one of the first problems to be encountered in 3-dimensional geometry: the gable of a house is an example; to make a picture of any object bounded by plane sides we need only draw edges – which are lines of intersection.

Taking an example from Murdoch [32, p.206-207], the equations of two planes are:

```
      x - 2y + 4z = 6
      2x + y - 3z = 8
```

Represent this information by a **2** by **4** matrix:

```
    m=. 1 _2 4 6,: 2 1 _3 8
    n=. |: m              Transpose
```

The direction numbers of the line of intersection are given by the determinants of three minors:

```
   k=. 1 2, 2 0,: 0 1
   h=. det@(k&{"1 2)@}:
   h n
2 11 5
```

Normalize to get the *direction cosines*:

```
   (h n) % rss h n
0.1632993 0.8981462 0.4082483
   norm=. % rss          A hook
   dc=. norm@h
```

Every point on the line must satisfy the equations for both lines. Set **z** to **0** and use *matrix divide* to find the co-ordinates of one point on the line:

```
   ct=. 0: ,~ {: %. |:@(2&{.)@}:
   ct n
4.4 _0.8 0
```

These are the *constant terms* of the parametric equation. Reset random link and choose 5 random parameters:

```
   setrl 7^5
   ]t=. 0.1* 50-~?5#100
_3.7 2.5 _0.5 0.3 _2.9
```

The co-ordinates of five random points on the line of intersection are then:

```
    ]z=. (ct n) +"1 (dc n) *"1 0 t
3.79579   _4.12314   _1.51052
4.80825    1.44537    1.02062
4.31835   _1.24907 _0.2041241
4.44899 _0.5305561  0.1224745
3.92643   _3.40462   _1.18392
```

The parametric equation of the line of intersection is:

```
   pequ=. ct@[ +"1 dc@[ *"1 0 ]
   z-: n pequ t        These match
```

Verify that these points satisfy the equations of the two planes, thus showing that the line lies in each of the planes, and is therefore the line of intersection [32, p.207].

```
    (n pequ t) +/ .* }:n
6 8
6 8
6 8
6 8
6 8
```

Find the normals to the two planes:

```
    normal=. norm@}:"1
    normal m
0.2182179 _0.4364358  0.8728716
0.5345225  0.2672612 _0.8017837
```

Show that these are indeed normal to the planes: on each plane choose two arbitrary points; the line joining any two points on the plane is perpendicular to the normal.

```
   (8 1 0-2 0 1) +/ .* normal {.m
1.11022e_16
   (1 6 0- 5.5 0 1) +/ .* normal{:m
1.11022e_16
```

The perpendicular distances from the origin to each plane [32, p.204]:

```
   dist=. ({: % rss@}:)"1
   dist m
1.30931 2.13809
```

The points where the normals from the origin meet the planes:

```
   (dist * normal) {.m
0.2857143 _0.5714286 1.14286
   (dist * normal) {: m
1.14286 0.5714286 _1.71429
```

Proof that these points lie on their respective planes:

```
   1 _2 4 +/ .* (dist * normal) {.m
6
   2 1 _3 +/ .* (dist * normal) {:m
8
```

The angle between the planes is the angle between their normals:

```
   'ab'=. (normal {.m);normal {:m
   dfr _2 o. a +/ .* b
134.415
   dfr _2 o. a +/ .* -b
45.5847
```

The two normals define a plane perpendicular to the line of intersection; consequently their vector cross-product is the line of intersection:

```
   u=. 1&|.@[ * _1&|.@]
   v=. _1&|.@[ * 1&|.@]
   vcp=. u - v
   c=. a vcp b
   c +/ .* a,.b       c  is normal to  a–b
0 0
```

The direction cosines of the line of intersection has been given in two ways:

```
  (dc n) -: norm c        These match
```

10.5 Projection of a line onto a given plane

This construction is used in computer graphics – the plane being the video screen or the bed of a plotter. The line itself is often the intersection of two planes.

Vector **p** makes angles of **110 80 22** degrees to the Cartesian axes **x** , **y** and **z** . Project it onto the plane **x - 2y + 4z = 6**, whose normal is **a**.

Compute the direction cosines of **p**:

```
   ]p=. 2 o. rfd 110 80 22
_0.3420201 0.1736482 0.9271839
```

q is the normal to the plane **p – a**; and **r**, which is the normal to the plane **a–q**, is the required projection of **p**.

```
   q=. norm p vcp a
   r=. norm a vcp q
   ]z=. norm a vcp norm p vcp a
_0.6419622 0.6094676 0.4652244
```

The required function is composed by a train of 5 verbs:

```
   proj=. [: norm ] vcp norm@vcp
    z -: p proj a        These match
```

Show that **z** lies in the given plane, whose normal is **a**:

```
    a +/ .* p proj a
0
```

A stereographic projection helps in visualizing these relations in 3-dimensions and confirming the results.

10.6 Volume of a Parallelepiped

Background and motivation for this example is given in another paper [30]. A parallelepiped is a solid with three pairs of parallel faces each of which is a parallelogram. The lengths of the three edges and the angles between them are given in a table. The data define the unit cell of the mineral chalcanthite, $CuSO_4.5H_2O$.

```
 (;:'a b c'),: ;:'alpha beta gamma'
+----------------+
¦a    ¦b   ¦c    ¦   axial lengths
+-----+----+-----¦
¦alpha¦beta¦gamma¦   interaxial angles
+----------------+
  <"0 ch=. 6.11 10.673 5.95,:
           97.583 107.167 77.55
+--------------------+
¦6.11  ¦10.673 ¦5.95 ¦
+------+-------+-----¦
¦97.583¦107.167¦77.55¦
+--------------------+
```

The formula for the volume is:

```
 abc √(1- cos² alpha - cos² beta -
cos² gamma +  2 cos alpha . cos beta
. cos gamma)
```

Define utility functions:

```
   sin=. 1&o.        sine of angle (radians)
   cos=. 2&o.        cosine of angle (radians)
   cosd=. cos@rfd    cosine of angle (degrees)
```

The formula for the volume is a *fork*, though it is interesting to note that the root (***)** is elided in the usual mathematical expression. The volume is determined by the square root of a function of the angles, which is then multiplied by a scaling factor dependent only on the lengths of the sides. The formula is therefore a *fork* of the form: **volume=. f * %:@g**

Because we know that **f** is the product of the lengths and **g** is some function of the cosines of the angles, the fork is written more completely as:

```
   volume=. */@{. * %:@h@cosd@{:
```

The formula for **h** is given as follows:

```
   1 - a² - b² - c² + 2d
```

Rearrange it as

```
   1 + 2d - (a² + b² + c²)
```

or, using *increment*

```
   (>: 2d) - (a² + b² + c²)
```

Erase any existing assignments: **erase 'p q'**
Once again we have a fork, and we can write:

```
   h=. p - q          where p and q are:
   p=. >:@+:@(*/)     1 + twice the product.
```

```
   q=. +/@*:          Sum of squares
   volume ch
361.035
```

There is another approach. The Grassmann Determinant Principle for space [19, p.3-33] permits the volume of a *space-segment* to be computed from the determinant of the matrix defining the co-ordinates of its corners. This has a mineralogical origin, because Grassmann called the space-segment a *spat*, from *Kalkspat*, the German word for calcite ($CaCO_3$), a common mineral which cleaves readily into parallelepipeds. Grassmann's work is of immense importance [4], but his original publications are notoriously difficult. Hyde, however, published a readable introduction [9].

The function **h** is equivalent to the determinant of the matrix given by the fork:

```
 g=. (0 3 2, 3 0 1,: 2 1 0)"_ { 1&,
    g
+---------------------+
¦+---------+¦{¦+-----+¦
¦¦0 3 2¦"¦_¦¦ ¦¦1¦&¦,¦¦
¦¦3 0 1¦ ¦ ¦¦ ¦+-----+¦
¦¦2 1 0¦ ¦ ¦¦ ¦       ¦
¦+---------+¦ ¦       ¦
+---------------------+
   h=. det@g
   volume=. */@{. * %:@h@cosd@{:
   volume ch
361.035
```

The data were given in the usual way as lengths of the sides and the angles between them, but the solution is much simpler if the data are given instead as three vectors in a Cartesian framework. The transformation involves rather cumbersome formulas derived from spherical trigonometry. The executable notation given here illustrates algebraic manipulation of composite functions.

Axial lengths *or* (depending on context) interaxial angles:

```
   a=. 0&{ [.  b=. 1&{ [.  c=. 2&{
   axisa=. sin@b , 0: , cos@b
```

cos(rho) and **cos(sigma)** are based on formulas given by Terpstra & Codd [34, p.287].

Compare the following versions:

```
 CosRho=. (cos@c - */@cos@(a,b)) %
sin@b       A fork
 CosRho=. (cos@c - [: */@cos a,b) %
sin@b       Parentheses required
 CosRho=. sin@b %~ cos@c - [: /@cos
a,b         No parentheses
 CosRho=. sin@b %~ cos@c - cos@a *
cos@b       No parentheses
```

Read this: *cos (rho) is sin b divided into {cos c - (cos a times cos b)}*

```
   CosSigma=. sin@b %~
(>:@+:@(*/@cos) - +/@*:@cos)
   CosSigma=. sin@b %~ [: %:
>:@+:@(*/@cos) - +/@*:@cos
```

Read this: *cos (sigma) is sin b divided into the square root of {(1+ twice the product of the cosines) - the sum of the squares of the cosines}*

```
 axisb=. CosRho,CosSigma,cos@a
 dm=. ,&0 0 1 @(axisa,:axisb)@rfd@b
 dm=. 0 0 1"_ ,~ (axisa,:axisb)@rfd@b
 dmat=. ({. *"0 1 dm)"2
```

We want **dmat** to apply to rank-2 cells.

```
   ]d=. dmat ch
5.83779      0 _1.80341
1.97316 10.394 _1.40843
      0      0     5.95
```

This matrix is called the *d-matrix* because it defines the unit cell of the *direct lattice*. Its rows give the Cartesian co-ordinates of the axes **a**, **b**, and **c**. The volume of the cell is simply the determinant of the d-matrix:

```
   det d
361.035
```

Bordering the matrix gives a signed volume:

det (,"1 1:) d,0 is **361.035** and

det (,"1 1:) 0,d is **_361.035**

After an arbitrary translation from the origin (moving a corner of the cell from the origin to **1 _2 3**), the matrix must be bordered to become 4 by 4, but the volume is unchanged:

```
   e=. 1 _2 3 +"1 d,0
   det (,"1 1:) e
361.035
```

Why is the volume of the parallelepiped computed so easily as taking the determinant of the 3 by 3 *d-matrix*? The answer is in Grassmann's extension into three dimensions of the method for computing the area of a triangle (or polygon). One of the corners of the figure is at the origin of the co-ordinate system. Because the co-ordinates of that corner are **0 0 0**, the expected 4 by 4 matrix degenerates to a 3 by 3; the need to border the matrix with 1's vanishes. Klein gives an excellent discussion of the volume of a tetrahedron [19, p.29].

The axes of the *reciprocal lattice* are the normals to the *faces* of the parallelepiped cell. Just as the *d*-matrix defines the *direct* lattice, so the *r-matrix* defines the *reciprocal lattice*. The *r-matrix* is in fact the transpose of the inverse

of the *d-matrix*. Moreover, the transpose of the inverse of a matrix is the same as the inverse of its transpose.

```
   cl=. * 1e_15&<@|   Clean very small values
(cl |:@%. d)  -:  cl %.@|: d   Match
```

Consider now a 5 by 2 by 3 array describing the unit cells of five minerals:

```
   ch=. chalcanthite=. 6.11 10.673
5.95,: 97.583 107.167 77.55
   or=. orthoclase=. 8.562 12.996
7.193,: 90 116.01 90
   an=. anorthite=. 8.177 12.877
14.169,: 93.17 115.85 81.22
   ax=. axinite=. 7.15 12.57 13.05,:
91.383 75.5 93.383
   ky=. kyanite=. 7.12 7.85 5.57,:
89.983 101.117 106
   $minerals=. ch,or,an,ax,:ky
5 2 3
```

Because we assigned **dmat** a rank of 2, it applies to the individual 2 by 3 cells of the array; hence

```
   $x=. dmat minerals
5 3 3
```

volume"2 minerals and **det x** both give the volumes of the unit cells of all five minerals.

The shape of the transpose of the inverse is, however, not at all what want: **$ |: %. x** is **3 3 5**

The reason is that while *matrix inverse* **(%.)** has a rank of 2, the rank of **(|:)** is unbounded (infinite). This is confirmed by using the *basic* adverb **b.** which enables the ranks of the associated verb to be displayed. This display should always be examined when the rank of a composite function is in doubt.

```
   %. b. 0
2 _ 2
   |: b. 0
_ 1 _
```

Consequently it is not the individual 3 by 3 cells that are transposed! Because the rank of **u@v** is the rank of **v**, the ***composed*** function **|:@%.** has the required rank of 2, but attempted execution of **%.|: x** or **%.@|: x** give length errors.

```
   |:@%. b. 0          %.@|: b. 0
2 _ 2                _ 1 _
   $ |:@%. x
5 3 3
```

This example demonstrates an important aspect of function composition: the rank of **u@v** is **mv lv rv** [16, p.172].

10.7 Rotations and projections

When 3-dimensional objects are described by Cartesian co-ordinates, they can be rotated by an inner product with a 3 by 3 matrix. This is analogous to **rot** in Section 10.3.

It is best to define separate matrices for rotations about the **x** , **y**, and **z** axes. Any desired rotation is then given by the inner products of these rotation matrices. Assuming that the co-ordinates are arranged in an **n** by **3** matrix, an inner product with an appropriate **3** by **2** matrix projects them onto the desired graphic plane. This projection matrix can be pre-multiplied by the rotation matrices to give a single **3** by **2** matrix that both rotates and projects in a single inner-product operation [3].

11 Summary and Conclusions

Iverson has greatly extended the methods of composition familiar to mathematicians, and applies composition to adverbs and conjunctions as well as to functions (verbs). His contribution marks an important advance in computer programming. Annotated examples show programmers how composition is used in practice.

12 Acknowledgements

I am grateful to Kenneth E. Iverson and Roger K.W. Hui for making this work possible, and for help and advice given generously on numerous occasions.

13 References

[1] Bernecky, R., and R.K.W. Hui. *Gerunds and representations*. APL91 Conference Proceedings, APL Quote Quad, Vol.21, Number 4 (1991) p.39-45.

[2] Bick, T.A. *Introduction to Abstract Mathematics*. Academic Press, New York (1971) p.35.

[3] Bond, W.L. *Computation of interfacial angles, interzonal angles, and clinographic projection by matrix methods.* American Mineralogist, Vol.31 (1946) p.31-42.

[4] Crowe, M.J. *A History of Vector Analysis: The evolution of the idea of a vectorial system*. University of Notre Dame Press. (1967) 270p.

[5] Davis, John C. *Markov Chains:* In: *Statistics and Data Analysis in Geology*. John Wiley & Sons, New York. 2nd Edition (1986) Chapter 4.

[6] Falkoff, A.D., and K.E. Iverson. *The APL Terminal System: Instructions for Operation*. IBM Research, International Business Machines Corporation,

Thomas J, Watson Research Center, Yorktown Heights, New York (November 30, 1966) p.21.

[7] Hui, R.K.W., K.E. Iverson, and E.E. McDonnell. *Tacit Definition*. APL91 Conference Proceedings, APL Quote Quad, Vol.21, Number 4 (1991) p.202-211.

[8] Hui, R.K.W., and K.E. Iverson. *TAGS: Trains, Agendas, and Gerunds*. APL94 Conference Proceedings, APL Quote Quad, Vol.25, Number 1 (1994) p.74-77.

[9] Hyde, E.W. *Grassmann's Space Analysis*. John Wiley & Sons, New York. Mathematical Monograph Number 6 (1906) 59p. First edition under the title *Higher Mathematics*, 1896. [A simple and concise presentation of the principles of Grassmann's Space Analysis. Grassmann's original work (1844, 1878, 1894) is very difficult.]

[10] Iverson, K.E. *A Programming Language*. John Wiley and Sons, New York (1962) p.73.

[11] Iverson, K.E. *A Personal View of APL*. IBM Systems Journal, Vol.30, No.4 (1991) p.582-593.

[12] Iverson, K.E. *Arithmetic*. Iverson Software, Inc., Toronto, Ontario, Canada (1991) 118p.

[13] Iverson, K.E. *Programming in J*. Iverson Software, Inc., Toronto, Ontario, Canada (1992). 76p.

[14] Iverson, K.E. *Calculus*. Iverson Software, Inc., Toronto, Ontario, Canada (1993) ISBN 1-895721-05-9.

[15] Iverson, K.E. *J: Introduction and Dictionary*. Version 7. Iverson Software Inc., Toronto, Ontario, Canada (1993). ISBN 1-895721-06-7

[16] Iverson, K.E. *J: Introduction and Dictionary*. Iverson Software Inc., Toronto, Ontario, Canada (1994) ISBN 1-895721-08-3.

[17] Iverson, K.E., and E.E. McDonnell. *Phrasal Forms*. APL89 Conference Proceedings, APL Quote Quad, Vol.19, Number 4 (1989) p.197-199.

[18] James, R. (James & James) *Mathematics Dictionary*. Van Nostrand Reinhold, New York (1992) 5th Edition, p.72-73, 175-176.

[19] Klein, F. *Elementary Mathematics from an Advanced Standpoint: Geometry*. Dover Publications, Inc., New York (1939) p.3-33. Translated from the 3rd German edition, 1925.

[20] Knopp, K. *Elements of the Theory of Functions*. Constable, London (1952) p.88.

[21] Kudravtsev, L.D. In: *Encyclopaedia of Mathematics*. Kluwer Academic Publishers, Dordrecht, Netherlands. Vol.2 C (1988), p.283-285; Vol.4 (1989) p.126-131.

[22] March, H.W., and H.C. Wolff. *Calculus*. McGraw-Hill, New York (1917) 360p.

[23] McDonnell, E.E. *At Play with J*. Vector. The Journal of the British APL Association. Vol. 10, No. 3 (1994) p.100-105.

[24] McDonnell, E.E. *At Work and Play with J: Control Structures in J Version 8*. Vector. The Journal of the British APL Association. Vol. 11, No. 1 (1994) p.136-138.

[25] McDonnell, E.E. *At Work and Play with J (Parallel Jacobi)*. Vector. The Journal of the British APL Association. Vol. 11, No. 3 (1995) p.111-118.

[26] McIntyre, D.B. *Mastering J*. APL91 Conference Proceedings, APL Quote Quad, Vol.21, Number 4 (1991) p.264-273.

[27] McIntyre, D.B. *Language as an Intellectual Tool: From Hieroglyphics to APL* IBM Systems Journal, Vol.30, No.4 (1991), p.554-581

[28] McIntyre, D.B. *Jacobi's Method for Eigenvalues*. Vector. The Journal of the British APL Association. Vol. 9, Number 3 (1993a) p.125-133.

[29] McIntyre, D.B. *AMENDMENT: "A Change for the Better"*. Vector. The Journal of the British APL Association. Vol. 9, No. 3 (1993b) p.134-142.

[30] McIntyre, D.B. *An Executable Notation, with Illustrations from Elementary Crystallography*. In: Computers and Geology - 25 years of progress. Edited by John C. Davis and Ute C. Herzfeld. Oxford University Press, New York and Oxford. (1993c) p.231-240.

[31] McIntyre, D.B. *J: A First Lesson* and *J: A Second Lesson.* The Education Vector. *In:* Vector. The Journal of the British APL Association. Vol. 10, No. 4 (1994) p.18-29, and Vol.11, No. 1 (1994) p.36-44.

[32] Murdoch, J.C. *Linear Algebra for Undergraduates*. John Wiley and Sons, New York. (1957) Appendix 2, *Three-dimensional analytic geometry.*

[33] Sylvester, J.J. *On Staudt's Theorems concerning the Contents of Polygons and Polyhedrons, with a Note on a New Resembling Class of Theorems.* Philosophical Magazine, Vol.4 (1852) p.335-345. Reprinted in Sylvester's Collected Works. Cambridge University. (1904) Vol.1, Number 48, p.382-391.

[34] Terpstra, P., and L.W. Codd. *Crystallometry*. Academic Press, New York. (1961) p.180, 286-287.

[35] Thomson, N. *Applying Matrix Divide in APL and J*. APL94 Conference Proceedings, APL Quote Quad, Vol.25, Number 1 (1994) p.211-215.

[36] Webb, J.R.L. *Functions of Several Real Variables*. Ellis Horwood, New York (1991) p.41.