Module11: Advanced Dot Syntax

§ 11.1 Object Variables

§§ 11.1.1 Stranding Object Properties

§§§ 11.1.1.1 Stranded Vectors

Since the advent of *floating array* second generation APLs, which we have described generically as APL 2 in APL1_2.PDF, variables (and unnamed parenthesised expressions) may be stranded together to form nested vectors of enclosed elements - simply by juxtaposing array expressions.

Strand (Vector) Notation: A series of two or more adjacent array expressions results in a vector whose elements are the enclosed arrays resulting from each expression. (See *Language Reference* p12.)

> Aside: In *fixed array* second generation APLs, pioneered by Ken Iverson in **SharpAPL** and **J**, strand notation is entirely avoided. Instead a new canonical primitive function, *link* (;), is introduced which encloses the Larg and catenates the result to Rarg, enclosing Rarg if it isn't already a vector of enclosed elements. In *fixed* notation, enclosing a scalar **is** meaningful.

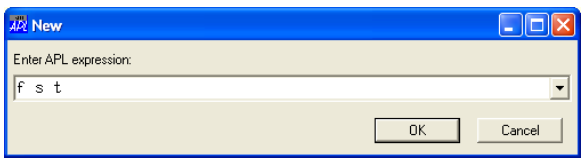
Given three variables *f* for first, *s* for second and *t* for third,

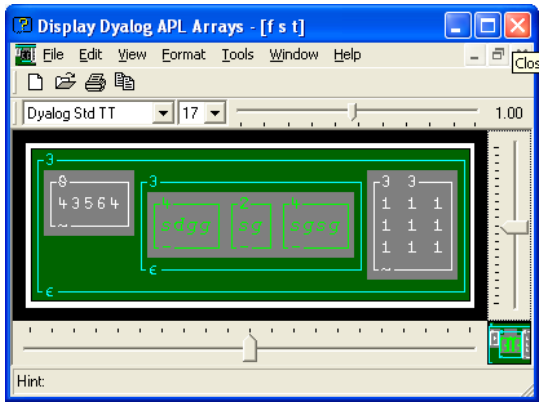
```
      f←43564                ⍝ Numeric scalar (NumSc)
      s←'sdgg' 'sg' 'sgsg'   ⍝ Vector of character vectors (VecCharVec)
      t←3 3⍴1                ⍝ Numeric matrix (NumMat)
      f s t                  ⍝ Strand (VecEncArr)
43564   sdgg  sg  sgsg   1 1 1
                         1 1 1
                         1 1 1
```

Strand notation has been generalised to *strand assignment*. The above 3 assignments can be achieved in one single statement:

```
      f s t←43564('sdgg' 'sg' 'sgsg')(3 3⍴1)
      DISPLAY f s t          ⍝ Display view of strand
.→-------------------------------------.
|       .→-------------------. .→----. |
| 43564 | .→---. .→-. .→---. | ↓1 1 1| |
|       | |sdgg| |sg| |sgsg| | |1 1 1| |
|       | '----' '--' '----' | |1 1 1| |
|       '∈-------------------' '~----' |
'∈-------------------------------------'
```

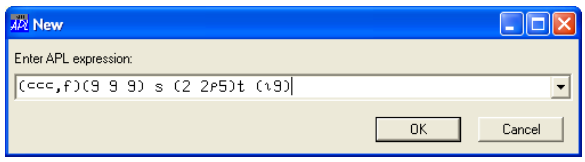
The **varChar** view of this strand is:



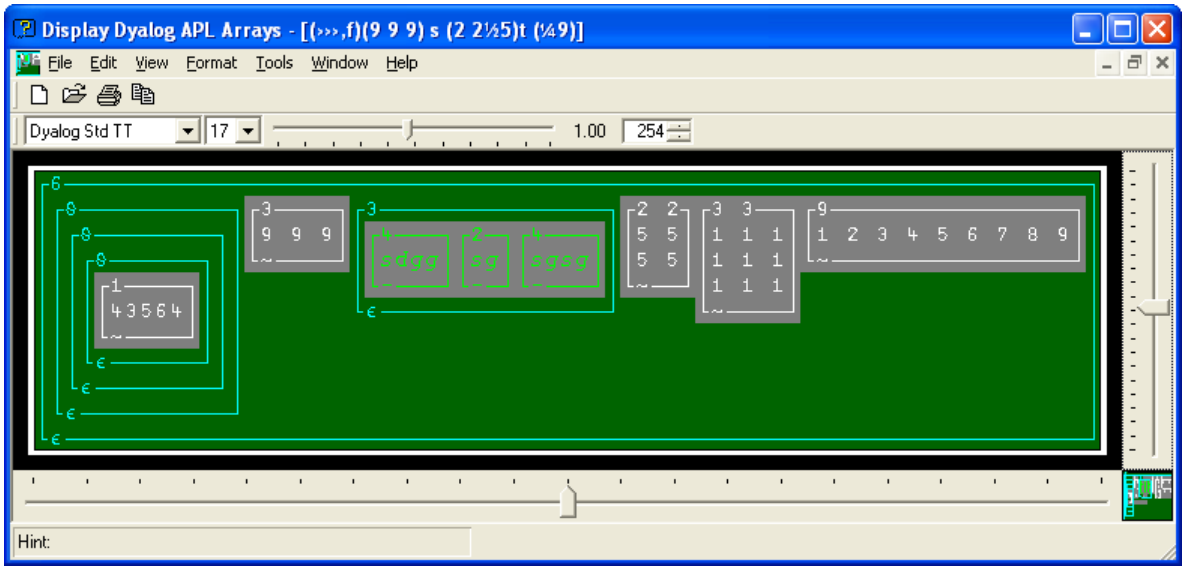


The resulting display clearly shows that the strand results in a three element nested vector of which the first element is a simple numeric scalar, the second element is a vector of character vectors and the third element is a simple numeric matrix.

11.1.1.1 Produce the example below in **varChar** and describe the elements of the six element stranded vector. Experiment with other strands.




```
(⊂⊂⊂,f)(9 9 9)s(2 2⍴5)t(⍳9)
```



Since object properties are essentially variables in the object space, we should be able to strand these into vectors of properties, and indeed we can.

11.1.1.2 Create a `Form`, `F`, and enter `F`-space. List its properties. Strand some of its properties, *eg*

```
      ⍴⎕←Accelerator AcceptFiles Active AlphaBlend AutoConf
 0 0   0 1 256 3
5
```

Assign a new `Posn` and `Size` in a single expression.

It often happens, especially in a **C** code environment, such as in `⎕NA` function calls, that arguments to and results of functions consist of amorphous strands of disparate variables. This general scenario is such a common occurrence in application code that, as of Dyalog version 10.1, it is possible to use **extended function header syntax**. With this syntax, the header line of a function may use strands of variable names in place of single names for arguments and results. For example a function that takes a date Rarg and returns the next day's date could have header line:

```
      ∇(Day Month Year)←nextDay(Day Month Year)∇
```

§§§ 11.1.1.2 Vectors of .. Vectors of Stranded Name Vectors

$§^{V..V}←S^{V..V}Arr$	⍝ Assigns nested array $S^{V..V}Arr$ to a compatible strand

Assignment is extended to pervade nested strands of names ($§^{V..V}$) to the left of the arrow. This allows multiple naming of parts of a structure in a single assignment. This name structure ($§^{V..V}$) is entirely constructed of lists of names. The depth of the overall vector structure ($V..V$) is determined by the

positions of parentheses. We give this new form of name-structure the generic symbol, $\S^{V..V}$, to indicate it is a vector name strand structure of arbitrary depth.

The conformability rules are similar to those for scalar (pervasive) dyadic primitive functions such as '+'. If *a b c* and *d* are stranded names to left of an assignment arrow then the data on the right of the assignment arrow must either be a scalar, in which case it is extended to fit the stranded structure, or a vector whose structure reflects the stranded structure of the names on the left.

```
((a b)(c d))←(1 2)(3 4)⇒ ((a←1)(b←2))((c←3)(d←4))
((⎕io ⎕ml)vec)←0 ⎕av   ⇒ ((⎕io←0)(⎕ml←0))(vec←⎕av)
(a(b(c d)e)f g)←1(2(3 4)5)6 7 ⇒ a b c d e f g←⍳7
(a(b(c d)e)f g)←1(2((3 3)(4 4))(⊂3 3⍴5))6 7
```

If a simple scalar is encountered at an earlier stage in the correspondence, then that scalar is extended to cover the structure beneath it, as is the case below:

```
(a b)(c(d(e f)))←1 (3 4)
```

This is a way of creating named arrays from a structure corresponding to vectors of .. vectors of names.

```
((first last) sex (street city country))←DATA
```

Each name in the structure may be space-qualified so we can step from

```
(first last)←'Ken' 'Chakahwata'
```

to

```
(A.first A.last)←'Ken' 'Chakahwata'
```

where

```
A.first↳'Ken'
A.last ↳'Chakahwata'
```

This is another step in the logical extension of second to third generation APL notation.

§§§ 11.1.1.3 Name Strands in `:For` Loops

The `:For` loop control structure allows multiple control variables using distributed assignment.

```
:For §V..V :In VecCS V..V Arr⋄..§...⋄:End      ⍝ Do ..§... for element C of VecCS V..V Arr
```

In this case the :*For* statement loops round once for each of the C elements of $Vec_C S^{V..V} Arr$. C symbolises the notional loop counter. ($Vec_C S^{V..V} Arr$ is ravelled if it is not already a vector.) The structure of each element ($S^{V..V} Arr$) should be compatible with the name strand structure $\S^{V..V}$, in which the ($V..V$) indicates an arbitrary depth vector of vectors .. of vectors. So the vector structure of names $\S^{V..V}$ reflects the vector structure of each element of vector $Vec_C S^{V..V} Arr$. The expression or expressions within the `:For` loop will most probably make reference to some of the names from the nested structure of names $\S^{V..V}$ and this likely scenario is indicated symbolically by ..§... .

For example,

```
:For a b c :In (1 2 3)(3 4 5)(5 6 7)(7 8 9)    ⍝ In this case C=4
     a b c     ⍝ on first loop ⊢a b c≡1 2 3
:EndFor
```

or

```
:For a(b(c d)) :In (1(2(3 4)))(5(6(7 8)))(15(16(17(2 2⍴18)))) ⍝ C=3
     a b c d  ⍝ on first loop ⊢a b c d≡1 2 3 4
:End
```

Alternatively,

```
:For §[V..V] :InEach S[V..V]Vec_CArr ◊..§...◊:End      ⍝ Do ..§... for each C in S[V..V]Vec_CArr
```

In this case, on the C[th] loop a strand consisting of the C[th] element of each of the vectors in $S^{V..V}Vec_CArr$ should have a structure compatible with the structure of the vector of vectors .. of vectors of names in $§^{V..V}$. (In this version of the `:For` loop, each element in the data vector is more likely to have a uniform structure.)

```
:For a b c :InEach (1 3 5 7)(2 4 6 8)(3 5 7 9)  ⍝ In this case C=4
     a b c  ⍝ on first loop ⊢a b c≡1 2 3
:EndFor
```

Formally, the alternatives may be presented as a table:

`:For`	§	`:In`	Vec_CArr
`:For`	$§^V$	`:In`	Vec_CS^VArr
`:For`	$§^{V..V}$	`:In`	$Vec_CS^{V..V}Arr$
`:For`	§	`:InEach`	Vec_CArr
`:For`	$§^V$	`:InEach`	S^VVec_CArr
`:For`	$§^{V..V}$	`:InEach`	$S^{V..V}Vec_CArr$

A corresponding table of examples may be written:

`:For`	`aa`	`:In`	$A_1\ A_2$ ... ⇒ `aa`←A_i on i^{th} loop round
`:For`	`aa bb`	`:In`	$(A_1\ B_1)(A_2\ B_2)$ ... ⇒ `aa bb`←$A_i\ B_i$
`:For`	`aa(bb cc)`	`:In`	$(A_1(B_1\ C_1))(A_2(B_2\ C_2))$ ...
`:For`	`aa`	`:InEach`	$A_1\ A_2$ ... ⇒ `aa`←A_i on i^{th} loop round
`:For`	`aa bb`	`:InEach`	$(A_1\ A_2$ ...$)(B_1\ B_2$ ...$)$ ⇒ `aa bb`←$A_i\ B_i$
`:For`	`aa(bb cc)`	`:InEach`	$((A_1\ A_2$ ...$)((B_1\ B_2)(C_1\ C_2)$...$))$

where `aa bb` and `cc` are valid variable names and $A_1\ B_2$ etc.. are arrays to be assigned.

11.1.1.3.1 If `aa bb cc` are replaced by `Posn`, `Size` and `Caption` properties of a `Form`, compare the data required in `:In` and `:InEach` loops.

§§ 11.1.2 Stranding Objects

§§§ 11.1.2.1 Pure Vectors of Namespace Objects

GUI objects are namespaces but namespaces are not necessarily GUI objects. The term namespace, or *space*, covers both flavours. Spaces are like variables – they can be arguments and results of functions.

In Dyalog APL we have numeric variables, character variables, and object variables. The strand

```
      ⍴⎕←# ⎕SE
 #   ⎕SE
2
```

exemplifies a simple 2 element pure vector of spaces. Notice that with the introduction of vectors of objects naturally comes the subtle generalisation of the primitive function *shape* (⍴).

```
      IntVec←⍴RVec                    ⍝ Shape of vector containing references to spaces
```

11.1.2.1.1 Create a 3 element strand of GUI objects.

11.1.2.1.2 Write a dummy function such as

```
      ∇ enterMouse Msg
[1]     Msg ∇
```

Attach it to the `MouseEnter Event` of a `Form` either by

```
      Event←'onMouseEnter' 'enterMouse'
```

or better by

```
      onMouseEnter←'enterMouse'
```

but not by superseded statements such as

```
      'F'⎕WS'Event' 'onMouseEnter' 'enterMouse'
      'F'⎕WS'Event' 'MouseEnter' 'enterMouse'
```

or

```
      Event←'MouseEnter' 'enterMouse'
```

Verify by tracing into the callback that `(⊃Msg)` is an object reference (dataType `RSc`). Notice the natural generalisation of *pick* (⊃) to apply to a vector containing refs.

```
      RSc←⊃RVec                       ⍝ Discloses first element (given ⊢⎕ML<2)
```

Use of the event prefix "`on`", as described in §§2.2.2, causes the first element in the message (`Msg`) that miraculously automatically appears for the right argument to any callback, to be a ref to the object rather than a character vector containing the name of the object.

The distinction between the name of an object and a ref to the object is clearly important. The distinction between the name of an array and the array itself is also important in understanding the *data representation* system function ⎕DR. ⎕DR applies to the array itself.

An object may have many *named* references to itself and so the real name of the object becomes a moot point. The question is similar to the Platonic question as to which is the real number `1`, `A` or `B`, in the expression `A←B←1`. Indeed, is `A` numeric, or just a named reference to a number? Is a named ref less valid than the name given to an object in ⎕WC? We argue that it should not be.

Notice that `Msg` appears to be a normal APL 2 nested vector, but actually two out of three of its elements are objects. (`Msg` contains dataTypes `RSc CVec RSc`.) APL 3 vectors can be composed of numbers, characters *and refs*. Henceforth we shall consider a named reference to be a proper name of an object.

The following lines place 100 `Button`s on a `Form`, position them and set their `Caption`s individually.

```
      'F'⎕WC'Form'
      FV←⍎¨('F',¨(⊂'.B'),¨⍕¨⍳100)⎕WC¨⊂'Button'
      FV.Posn←,45×⍳10 10
      FV.Caption←↓100 3⍴⎕A
```

11.1.2.1.3 Take the following simple vector of *Form*s, *RVec*, given by

```
≡RVec←⍬⍴⎕Á⎕WC⊂'Form'↳1
```

and create a *Button* on each. Assign a different *Caption* to each *Button*. Delete all the *Form*s.

11.1.2.1.4 Look up monadic *⎕WC* in the *Language Reference* or in [Help][Language Help] and convert the vector of namespaces, *RVec*, below into a vector of *Form*s.

```
≡RVec←⍬⍴⎕A⎕NS⊂''↳1
```

§§§ 11.1.2.2 Mixed Vectors

11.1.2.2.1 Check that the expressions `# 65 'a'` and `F'a'F'b'F'c'` are simple vectors. What happens if you replace `'b'` with `'b '`?

Tip: Use **varChar** to get it clear.

Now we have not just a single variety of (simple) *mixed* arrays in Dyalog APL, but the 1 original variety (numeric scalars mixed with character scalars) plus 3 exotic varieties; numeric and object, object and character, and all three - numeric, character and object. There are 3 ways of selecting 2 combinations from 3 basic types of array, and 1 way of selecting 3 combinations from 3 basic types. So the total number of varieties is four since

```
+/2 3!3↳4
```

Actually there are now eleven varieties of mixed arrays in Dyalog APL because in version 10.0 a new scalar *Null* item was introduced through the niladic system function, *⎕NULL*. *⎕NULL* returns a new type of scalar item, display form `[Null]`, which may be catenated to any simple APL vector to give another simple APL vector. This means that there are eleven different varieties of simple mixed arrays in Dyalog APL version 10.

```
+/2 3 4!4↳11
```

Note that the *⎕OR* of a function is a scalar, but, anomalously, it has depth 1 and therefore must be enclosed before it can be catenated to a simple vector. The resulting vector (*Vec*) is therefore necessarily, non-simple and ⊧`1<|≡Vec`

The depth of the following vector is `1` implying that it is a simple vector.

```
≡1E¯14 'A'(⎕NS'')#.Á #.Â ⎕NULL↳1
```

However the depth of the next example implies that the vector is thoroughly nested.

```
≡¯1.797693135E308 'A'(⊂3 3⍴⎕A⎕NS⊂'')(2 2 2⍴⎕Á⎕WC⊂'Form')↳¯4
```

§§§ 11.1.2.3 Control Structures with Objects

Monadic use of the *⎕NS* system function with an empty Rarg returns a ref to a "vanilla" namespace.

```
RSc←⎕NS⍬                                ⍝ Creates an empty 'unnamed' namespace. NB ⊧⍬≡⍬
```

Note that the Rarg of monadic *⎕NS* can be an array of names (or the *⎕OR* of a namespace), in which case these objects are copied into the new namespace.

Making a new ref to an unnamed namespace does not make a new copy but simply points to the original one. However, one unnamed namespace is not the same space as another.

```
(⎕NS'')=⎕NS''↳0        because        ⊧(⎕NS'')≠⎕NS''
```

```
NS=NS←⎕NS''↳1
```
because
```
≠NS≡NS←⎕NS''
```

11.1.2.3.1 Are the 3 spaces created by (⎕NS¨3⍴⊂'') identical or just similar (isomorphic)?

The :With control structure accepts a ref, RSc (which can be a *collection*), or a string, CVec, containing the name of a space. It also, therefore, accepts an unnamed namespace. This can be useful for localising more or less complex lines of code.

```
:With ⎕NS''
    CVec←'This string is ephemeral.'
:End
```

:For also applies to *collection* objects as found, for example, in Word and Excel. Collection objects encourage irregular syntax in VB because they always have an Item method and this method name may be elided in VB code. Thus **Application.Workbooks.Item(1)** is the same as **Application.Workbooks(1)** in VBA. This syntax is emulated in :For as applied to a collection in which each Item in Count is automatically instantiated sequentially in the loop. (Dyalog version 11.0 goes much further in incorporating this VB anomaly.)

```
:For It :In Documents ⍝ Documents is a Word Collection Object
     It.Name
:End
```

```
:For Sh :In Sheets    ⍝ Sheets is an Excel Collection Object
     Sh.Name
:End
```

§§ 11.1.3 Arrays of .. Arrays of Objects

§§§ 11.1.3.1 Reshaping Object Vectors

Generalisations of the APL primitive function *pick* (⊃) to select an object from a nested vector, or *shape* (⍴) to obtain the shape of a vector of refs are natural extensions that can go almost unnoticed.

Object spaces are essentially a new type of scalar since '⎕WC'Form'⋄⍬≡⍴F↳1 (but ⎕NC'F'↳9!). We can now assign objects to names, G←F , including strands, FGF←F G F , and then ⎕NC'FGF'↳2 which is more comprehensible than name class 9.

```
NAME←RArr                    ⍝ Creates name for object reference array
```

Beware that assignment cannot change the class of an existing variable (although this is ameliorated in version 11) and therefore if G had already existed as a class 2 object then G←F would have given a SYNTAX ERROR.

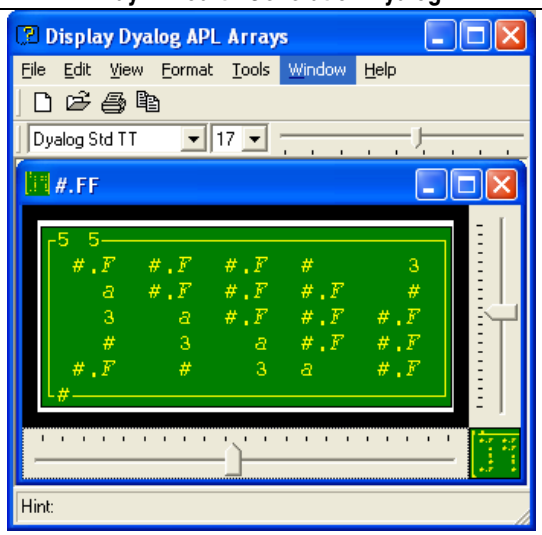
The extension of dyadic *reshape* (⍴) is harder to miss than that of monadic *shape* because it allows you to generate matrices and higher rank arrays of objects. Indeed, you can make arbitrary nested arrays of arrays of .. arrays of object spaces, numbers and characters.

```
RArr₂←IVec⍴RArr₁             ⍝ Reshapes array containing references to spaces
```

For example, consider the matrix

```
FF←5 5⍴F G F # 3 'a'     ⍝ Simple mixed RefArr
```

In **varChar**, `FF` displays as shown. Notice that the display form of `G` is `#.F`. This is the case even after `F` has been erased. `F` and `G` are references to the same object.



11.1.3.1.1 By creating a variable inside `F`-space, and examining the contents of `G`-space, demonstrate that `F` and `G` are not separate objects. Expunge the `Form` by expunging **all** references to it.

Aside: There could be 11 different colour codings for the different types of mixtures. Currently, **varChar** just picks the first available exotic colour for exotic mixtures.

Another definition that is hard to ignore is that of that of *dyadic* `⎕NS` with an empty right argument.

`CVec2←CVec1⎕NS⍬`	⍝	Returns the full name of space, named in `CVec1`

Note `⊢⍬≡⍬`

11.1.3.1.2 Create a rank 3 array of named namespaces.

Unless an unnamed namespace (or array of unnamed namespaces such as `2 2⍴⎕NS¨⊂''`) is used immediately then it evaporates to nothing at the end of the expression. If space are named by some reference, `RA←2 2⍴⎕NS¨⊂''`, or individually by

```
      2 2⍴'ABCD'⎕NS¨⊂''
#.A   #.B
#.C   #.D
```

then, like GUI spaces which are usually named on creation as in, for example,

```
      ⍴¨'EFGH'⎕WC¨⊂'Form'↳ 3  3  3  3
```

they do not evaporate until explicitly expunged by some particular action or mechanism.

§§§ 11.1.3.2 Generalised Primitives

A number of other primitive and system functions have been extended to handle space arrays. Where appropriate, they take arguments of arrays containing spaces and return arrays containing spaces.

`CArr←⍕RArr`	⍝	Returns `CArr` of character display forms
`RArr2←IArr⊃RArr1`	⍝	Picks from array (according to `⎕IO`)
`EncSc←⊂RArr`	⍝	Encloses array containing references to spaces
`VecEncRArr←BVec⊂RArr`	⍝	Vector of enclosed arrays of refs (given `⊢⎕ML<3`)
`RArr2←↑RArr1`	⍝	Mixes nested array to higher rank (given `⊢⎕ML<2`)
`RArr2←IntVec↑RArr1`	⍝	Takes (not yet overtakes) from array containing refs

For overtake, we would need an *identity element, eg* `@`, for the *take* function when applied to spaces.

`RArr2←↓RArr1`	Splits nested array to lower rank
`RArr2←IntVec↓RArr1`	Drops (not yet overdrops) from array containing refs
`RArr2←⌽RArr1`	Reverses array containing references to spaces
`RArr2←IArr⌽RArr1`	Rotates according to simple integer array, `IArr`
`RArr2←⊖RArr1`	Reverses array `RArr` along first axis
`RArr2←IArr⊖RArr1`	Rotates `RArr` along first axis according to `IArr`
`RArr2←⍉RArr1`	Transposes all axes of `RArr1`
`RArr2←IVec⍉RArr1`	Transposes `RArr1` according to axis positions `IVec`
`VecRArr←,ArrRArr`	Ravels arbitrarily nested array to vector
`RArr3←RArr2,RArr1`	Catenates conformable arrays along last axis
`RArr3←RArr2⍪RArr1`	Catenates conformable arrays along first axis
`ISc←≡RArr`	Depth of array containing references to spaces
`BSc←RArr2≡RArr1`	Whether arrays of refs all point to the same spaces
`BSc←RArr2≢RArr1`	Whether refs don't all point to the same spaces
`BArr←RArr2=RArr1`	Pervasive scalar equality of elements
`BArr←RArr2≠RArr1`	Pervasive scalar inequality of elements
`RArr2←IVec/RArr1`	Replicates array containing references to spaces
`RArr2←IVec⌿RArr1`	Replicates along first axis
`RArr2←IVec\RArr1`	Expands array containing references to spaces
`RArr2←IVec⍀RArr1`	Expands along first axis
`RVec2←RVec1~RArr`	`RVec1` without elements in `,RArr`

The extension of primitive *without* (~) is implemented in Dyalog APL version 11.

`IArr←RVec⍳RArr`	Index of `RArr` in `RVec`

`BArr←Arr⍷RArr`	⍝ Finds `Arr` in `RArr`
`RArr₂←RArr₁[Index]`	⍝ Elements from $RArr_1$ according to `Index` spec.

The `Index` specification may be *simple indexing*, *choose indexing* or *reach indexing*. The three corresponding flavours of *indexed assignment* also apply to arrays containing references to objects.

The operators *reduce* (`/`), *reduce first* (`⌿`), *scan* (`\`), *scan first* (`⍀`), *each* (`¨`), *compose* (`∘`), and the *axis* operator (`[ ]`) have all been generalised to deal with arrays containing references to objects. It could be argued (see § 3.3 and New Foundations in *Vector Vol.20 No.1 p132*) that the product operator (`.`) has also been generalised.

11.1.3.2.1 Check the structure of matrix `(# ⎕SE)∘.,(# #)`.

As with the introduction of nested arrays in APL 2, the introduction of arrays containing references to objects is so natural that it is obvious how to generalise the definition of many primitive APL functions and operators, especially the structural functions. There remain some candidate functions such as *type* (`∈⍵`), *membership* (`⍺∈⍵`), *unique* (`∪⍵`), *union* (`⍺∪⍵`) and *intersection* (`⍺∩⍵`), which are not yet implemented but which would seem to have natural generalised definitions. There are others, such as *take* (`⍺↑⍵`), *drop* (`⍺↓⍵`), *without* (`⍺~⍵`) and *find* (`⍺⍷⍵`), which have been partially generalised and are still to be fully generalised.

§§§ 11.1.3.3 Generalised System Functions

A number of system functions have also been generalised to accommodate object references.

`CVec←⎕CS RSc`	⍝ Changes to space `RSc` from space named in `CVec`
`MsgVec←⎕DQ RVec`	⍝ Dequeues events associated with objects in `RVec`
`Arr←IntSc⎕NQ RSc CVec …`	⍝ Enqueues event or method in `CVec` of object `RSc`
`CMat←⎕FMT RArr`	⍝ Character matrix of display forms
`RArr ⎕FAPPEND Tie`	⍝ Appends `RArr`, which can include `⎕OR` of space
`RArr ⎕FREPLACE(Tie Cpt)`	⍝ Replaces `RArr`, which can include `⎕OR` of space

Further generalisations of system functions and variables will appear in later versions of Dyalog. For example, `⎕NS`, `⎕NSI` and `⎕PATH` are candidates. Even `⎕CT` could be generalised to soften equality of namespaces, for example by ignoring the contents of column 2 of the result of `⎕AT`. And `⎕WC` is challenged by `⎕NEW`, to be described in Module20.

The question naturally arises as to how to make a **deep** copy of a space. Direct assignment only creates a **shallow** copy, *ie* it creates a pointer or ref to the single copy. A deep copy of a namespace may be created using a combination of `⎕OR` and `⎕WC`.

$CVec_2$ `⎕WC ⎕OR` $CVec_1$	⍝ Clone space named in $CVec_1$ to name in $CVec_2$

For example, given

```
a←⎕ns''⋄ a.x←33    ⍝ Create a namespace containing variable x

'b'⎕wc ⎕or 'a'     ⍝ Clone the namespace containing variable x
b.x↳33               x in b is 33
a.x↳33               as is x in a
b.x←4              ⍝ Assign x in b to 4
a.x↳33               x in a is still 33
b.x↳4                but x in b is now 4
```

11.1.3.3.1 Experiment with some of the above generalisations on mixed and nested arrays containing refs.

§ 11. 2 Understanding `(...).(...)`

§§ 11.2.1 Expanding Array.Strand

§§§ 11.2.1.1 New Rules

Rule 4: Dots bind tighter than strands. (Strands bind tighter than indexing brackets…) (Indexing brackets bind tighter than rational primitive functions…)…

Rule 4 helps one to interpret correctly the order of execution of 3rd generation APL statements involving *dot* syntax. The rule may be expressed simply as the order of precedence of dot binding w.r.t. strand binding. APL 1 claimed no special hierarchy of binding strengths amongst functions, nor, separately, amongst operators. There were, however, some anomalous cases like `∘.` and `[]` brackets which should be eliminated (see K.E.Iverson, *Rationalised APL*, IPSA Research Report No.1). Instead anomalous cases have been replicated and others have been introduced, moving APL ever closer to standard multi-rule 'evolved' computer programming languages like FORTRAN, **C**, VB, VB.NET and **C#**.

This rule allows one to take a next step and rewrite

```
A.first A.last←'Andy' 'Shiers'
```

as

```
A.(first last)←'Andy' 'Shiers'
```

both of which imply that

```
⊢A.first↳'Andy'
⊢A.last↳'Shiers'
```

This is another step in the logical extension of second to third generation APL notation. Rule 4 implies that parentheses are required above because

```
A.first last←'Kai' 'Jaeger' ⍝ That other hero!
```

implies that

```
⊢A.first↳'Kai'
⊢last↳'Jaeger'
```

Therefore we *do* need parentheses around the name strand `first last` if we expect both variables to be in namespace `A`.

Exactly the same applies to GUI objects and properties. According to Rule 4, dots bind tighter than strands and therefore

```
F.Accelerator AcceptFiles↳(F.Accelerator)(AcceptFiles)
```

which is probably *not* what we want because the parent of `F` probably does not even have an `AcceptFiles` property. Instead we must write

```
      F.(Accelerator AcceptFiles)↳(F.Accelerator)(F.AcceptFiles)
```

Then, for example,

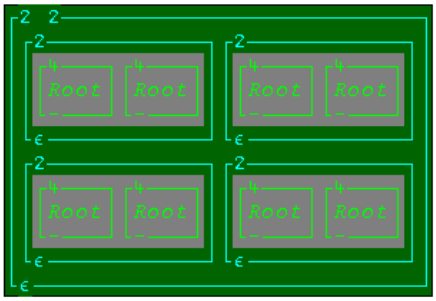
```
      F.(Accelerator AcceptFiles Active AlphaBlend AutoConf)
0 0  0 1 256 3
```

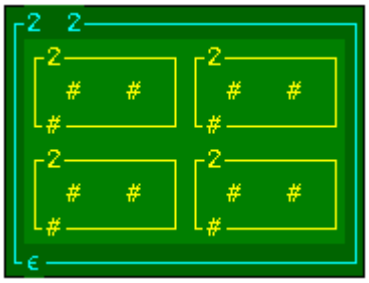
Syntax rules (*ie* parsing rules) of the APL language should be clearly distinguished from the semantics associated with specific symbols or tokens. The semantic details regarding execution of symbols representing primitive functions and operators are the **algorithms** that define the meaning of the symbols. This should be kept distinct from **grammatical rules**. We therefore define the rules for expanding the various forms of dot syntax in terms akin to the definition of a new dualistic niladic *dot* operator.

$S_D S^{V\ldots V} Arr \leftarrow \tilde{n}_D.\S^{V\ldots V}$	⍝ Variable strand pervades deep space nesting

If to the left of a *dot* is an array of refs, $\tilde{n}_D$, of arbitrary depth, D, and to the right is a stranded vector of vectors .. of names, $\S^{V\ldots V}$, then the name strand pervades the array of refs so that the structure of the result is the structure of the array of refs (S_D) and, within that, the structure of the name strand ($S^{V\ldots V}$), with the value (Arr) of each name returned at that point in the nesting.

11.2.1.1.1 Check that the structure of the result of `((# #)∘.,(# #)).Type` is consistent with $\tilde{n}_2.\S$.





Here is a final rule which helps one to read and write 3rd generation APL statements involving *dot* syntax and which we have inadvertently assumed above.

Rule 5: The parenthesised expression in `F.(...)` is executed in `F`-space.

As with strand notation, this rule seems natural in many circumstances, but it is extra to APL 1 & 2. But note that `⊢#.1≠#.(1)`.

The expression `(...)` may not return a result. This causes a `VALUE ERROR` in the case of a niladic function such as `F.(Detach)`. In this particular case the error can be avoided by using `F.Detach`.

If *dot* had been treated as a dualistic niladic operator (with a valid right operand `(...)`), then Rules 4 and 5 might both have been unnecessary. See New Foundations in *Vector Vol.20 No.1* for discussion of a rational alternative. Henceforth we shall consider new issues involving the interpretation of execution of *dot syntax* structures as ***intrinsic properties of the dot symbol*** rather than as new APL syntax rules.

Nevertheless, this new rule (Rule 5) is a valuable and profoundly useful addition to Rules 1 & 2. It is very useful to be able to execute arbitrary expressions in a distant space. For example;

```
      ⎕SE.(⎕IO ⎕ML←1 0)
      ⊢(⍴⍤⎕SE.(⎕CR⍤↓⎕NL 3))≡(12 40)(19 102)
```

and also

```
      #.FORM1.GROUP2.SUB2.TotalSLP<#.FORM1.GROUP2.SUB2.TotalGMP
```

simplifies to

```
      #.FORM1.GROUP2.SUB2.(TotalSLP<TotalGMP)
```

Properties and methods of GUI objects are just special cases of variables and functions and as such almost everything that is said about variables and monadic functions w.r.t. *dot* syntax applies equally to properties and methods.

11.2.1.1.2 Use **varChar** to examine the properties of a `Calendar` on a `Form` by way of expressions like `↳ 10⍴#.F.CAL.(⍎¨PropList)` and `#.F.CAL.(CircleToday Border)`

The object to the left of the dot *might not be a simple scalar object*, but may instead be an arbitrary *pure* (ie not mixed) array of objects. In this case **the expression to the right of the dot *pervades* the nested array of refs to its left**. When an array of refs is dotted with a variable name or object property name then the name pervades the array.

```
      (X Y).⎕IO ↳ (X.⎕IO)(Y.⎕IO)
```

which is the obverse, or complement, of

```
      X.(⎕IO ⎕CT) ↳ (X.⎕IO)(X.⎕CT)
```

which follows from Rule 5.

```
      (2 3⍴U V W X Y Z).⎕IO              ⍝ matrix of space origins
1 1 1
1 1 1

      ⍴(F.(C D E)).Type↳3               ⍝ get 3-vector of object Types

      (F1.(B1 B2) F2.(B3 B4)).Caption   ⍝ get ↳ Button Captions
  Salt  Mustard    Vinegar  Pepper
```

When an array of refs is dotted with a **strand of variable names** or a strand of object property names then the strand pervades the array. We need parentheses in order to strand the 2 variables V_1 and V_2, and then this strand pervades `X` and `Y`

```
      (X Y).(V1 V2)↳(X.(V1 V2))(Y.(V1 V2))↳(X.V1 X.V2)(Y.V1 Y.V2)
```

11.2.1.1.3 Create two `Form`s `F` and `G` and examine the structure of the results of expressions such as `(2 2⍴#.F).(Type State)` or `((F F)(G G G)).(Size (Posn Caption))`

Aside: We might have thought that these would be outer products written $S_D S^{V..V} Arr \leftarrow \tilde{n}_D \circ . \S^{V..V}$? Discuss.

§§§ 11.2.1.2 Parsing Rules

We are going way beyond the usual VB dot syntax in the APL direction. But before we explore the full generality of APL *dot* syntax, let us recapitulate the evolution of APL syntax as encapsulated in our list of grammatical rules extracted from key language ingredients.

As was regrettably the case with strand notation and Rule 3 in APL 2, the simple grammar of APL 1 is considerably complicated by the introduction of *dot notation* in APL 3. Beautifully simple formal syntax rules such as Rule 1 relating to function parsing interpretation, and Rule 2 relating to operator parsing interpretation, are supplemented by other *ad hoc* heuristic rules of precedence defining the order of execution of special new unclassified tokens in a line. The opportunity to completely obviate *strand notation* caused a split in the APL community in the mid 1980's. An opportunity to rationalize APL *dot notation* by way of a strict interpretation of dot as a dualistic niladic operator, which would reduce Rule 4 to Rule 2, has also been lost, as have other opportunities to rationalize along the way. In this regard **J** from Jsoftware is probably the most rational (and, unfortunately, most illegible) dialect of APL today.

In seeking a rule-based view of APL, many irrational and extraneous features of APL have been ignored in an attempt to present a simple unified view. Let us for a moment consider in a long list some other rules we might have considered to be essential.

Rule 0: System commands begin with a right parenthesis (as what else could?) and do their own thing (but this is not APL proper…).

Rule 0.1: Enter numeric vectors by using standard number formats and spaces between elements, or enter character vectors by surrounding a string in single quotes. This is the real beginning and is very natural.

Rule 1: Function sequences execute from right to left. (This is the usual *APL Rule* and follows advanced maths.)

Perhaps because Indo-European languages are read from left to right, most infix mathematical functions (such as minus) are left-associative; that is, a series of functions of the same precedence is evaluated from left to right. However, prefix operators (such as power) are usually right-associative. APL adopts right-asociativity universally for all functions.

Rule 1.1: There are a number of rules for function header syntax (and for the del (∇) or other function editors…).

`∇Niladic`	`∇R←Niladic`
`∇Monadic W`	`∇R←Monadic W`
`∇A Dyadic W`	`∇R←A Dyadic W`

And rules for shyness and ambivalence and name strands…

Rule 1.2: There are rules for *semicolon* (;) indexing and indexed assignment (ameliorated by squish-quad (⌷) indexing function in IBM APL2 and Dyalog version 11)…

Rule 1.3: *Indexing brackets* bind tighter than rational primitive functions.

hence `7 8[1]×2 ↳ 14` and not a `SYNTAX ERROR`

Rule 1.4: Right arrow (→) can be used niladically and monadically. (This breaks the *metagrammatical rule* that symbols may be employed both monadically and dyadically (ambivalently), but not either of these and niladically.)

Otherwise for example `++7` would be ambiguous as the `+` on the left might be interpreted monadically *or niladically*.

Rule 1.5: *Labels* are immediately followed by a colon (:) and may be used at *the beginning* of a program line to hold dynamically line numbers as class 1 variables.

Rule 1.6: *Comments* (⍝) at *the end* of a line may be used to hold arbitrary text (partially obviated by the introduction of *lev* (⊣) and *dex* (⊢) in SharpAPL).

Ken Iverson himself seemed to enjoy using pure APL to add comments in the following manner (even prior to *lev*):

```
      3+4,0⍴'Here we add 3 to 4.'↳7
```

Rule 1.7: *Diamonds* can be used on a line (and in execute strings) to separate statements.

11.2.1.2.1 Try expressions such as that below. This can make APL programs uncompilable.

```
      ⍎⎕AV[16⍴254 81 88 245]   ⍝ ⊢⎕IO≡1
98
98
98
98
```

Rule 2: Operator sequences execute from left to right.

It is easy to find non-associative functions (eg `~⊧((A-B)-C)≡A-(B-C)` where eg `A B C←?3⍴⊂3 3⍴100`).

Non-associative operators are also possible. They reveal the default order of execution of operator sequences.

```
~⊧(A(-.×).×C)≡A-.(×.×)C  ⍝ Notice that this sentence is known to be true because there is no error.
⊧(A(-.×).×C)≡A-.×.×C     ⍝ Similarly, notice that this sentence needs no 'explanatory' result.
```

Rule 2.1: There is a special rule for outer product (∘.) syntax. Rationalisation is compounded by introduction of the *jot* (∘) operator. (Jot could have been defined as enclose zilde, for example.)

Rule 2.2: *Slash* (/) and *slope* (\) can be both functions and operators. (This breaks the meta rule that symbols may be either functions or operators but not both.)

Rule 2.3: There are a number of rules for operator header syntax (and for the del (∇) or other editor…).

`∇(f MonisticMonadic) W`	`∇R←(f MonisticMonadic) W`
`∇A (f MonisticDyadic) W`	`∇R←A (f MonisticDyadic) W`
`∇(f DualisticMonadic g) W`	`∇R←(f DualisticMonadic g) W`
`∇A (f DualisticDyadic g) W`	`∇R←A (f DualisticDyadic g) W`

And rules for shyness… Neither nihilistic operators nor operators that return niladic derived functions figure in 2nd generation APLs. (See APL Linguistics in *Vector Vol.2 No.2* for a classification scheme.)

Rule 2.4: The *axis* operator has special rules, similar to those for bracket indexing (see Rules 1.2 and 1.3).

Rule 2.5: Then there are all the individual rules surrounding the syntax for the (multiply-classified) *assignment arrow*, including choose assignment, modified assignment and function assignment…

Rule 3: Strands bind tighter than indexing brackets.

Rule 3.1: There are different rules for different control structures, but all of them have to start with a colon followed by a keyword, take an arbitrary number of lines, and end with an :End(optionally immediately followed by the initial keyword).

Rule 3.2: Special uses of symbols ¯ E ⍞ ⍠ ↑ ⍺ ⍵ Δ ⍙ _ (apart from those already mentioned above) ∘ ⍝ ◇ ; : ∇.).

Rule 4: Dots bind tighter than strands.

Rule 4.1: There are many new rules associated with the definition of *DFns* and *DOps* in Module12, including proliferation of paired symbols, as in `##` above.

Rule 5: The expression inside the parentheses in `F.(...)` is executed in *F*-space.

Rule 5.1: There are new 'rules' associated with the expansion of dotted structures.

As rules proliferate their identification becomes harder. Ultimately it is the parser code that determines the rules and therefore there should be a move to focus on the details of the Dyalog APL parser to identify exactly what the rules are. Nevertheless it has been considered very important to explicitly enunciate the

major rules of APL because *the reader*, not just the machine, *has to be able to parse* a line accurately if (s)he is to understand it.

11.2.1.2.2 In which spaces are `k`, `l` and `m` most likely to be found in the expression

```
#.A[k].B[l].C[m]
```

Compare

```
#.A.B C.D
#.A.B C[2]
#.A.(B C).D
```

§§§ 11.2.1.3 Generalised Strand Assignment

Assignment into an **array of refs** dotted with a variable name or object property name requires either a scalar argument, which experiences scalar extension, or an array of conformable shape and structure to the shape and structure of the array of refs. Assignment is pervasive.

```
(X Y).⎕IO←0    ↳ (X.⎕IO←0)(Y.⎕IO←0)
(X Y).⎕IO←0 1  ↳ (X.⎕IO←0)(Y.⎕IO←1)
(F1 F2).Caption←'F1' 'F2'     ⍝ Set both Form Captions.
```

Assignment into an array of refs dotted with a **stranded structure of variable names** (or object property names) requires either a scalar argument, which experiences scalar extension, or an array of conformable shape and structure to the shape and structure of the array of refs dotted with the strand. The whole strand structure pervades each element of the ref array. Strand assignment is *totally pervasive*.

```
F.(Caption OnTop)←'The End' 1
(X Y).(first last)←('Søren' 'Kierkegaard')('Dan' 'Baronet')
```

Scalar extension can occur at various levels depending on the structure of the data array.

```
(X Y).(⎕IO ⎕ML)←0             ⍝ Scalar extension of scalar 0.
(X Y).(⎕IO ⎕ML)←⊂0 0          ⍝ Scalar extension of scalar ⊂0 0.
(X Y).(⎕IO ⎕ML)←2⍴⊂0 0        ⍝ No scalar extension required.
```

```
ñD.§V..V←SDSV..V Arr                ⍝ Strand assignment pervades deep space nesting
```

If an array of refs, $\tilde{n}_D$, is to the left of a *dot* and a stranded vector of names, $\S^{V..V}$, is to the right then the name strand pervades the array of refs. Data assigned to the expanded set of names given by $\tilde{n}_D.\S^{V..V}$ must have a structure that mirrors the structure of $(V..V)$ within an outer structure of depth D, $S_D S^{V..V}\ldots$ Each element within this container structure may be any arbitrary enclosed array. So the structure of the data, $S_D S^{V..V} Arr$, has the structure of the array of refs $\tilde{n}_D$, and within that the structure of the name strand $\S^{V..V}$, with the value of each name assigned to the corresponding arbitrary data array Arr in $S_D S^{V..V} Arr$ at that point. If a scalar is encountered in the data at an earlier stage in the correspondence, then that scalar is extended to cover the structure beneath it, as is the case in ordinary strand assignment without ref arrays:

```
(a b)(c(d(e f)))←1 (3 4)
```

Note there is no resulting difference between the following two assignments

```
F1.(Posn Size)←(55 40)(25 58)
F1.(Posn Size←(55 40)(25 58))
```

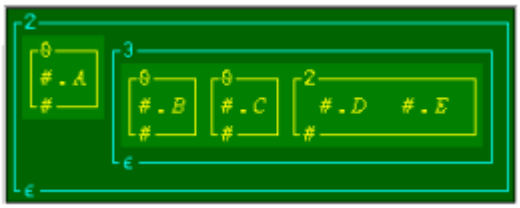
However, a significant general difference between $\tilde{n}_D.\S^{V..V}\leftarrow..\S\ldots$ and $\tilde{n}_D.(\S^{V..V}\leftarrow..\S\ldots)$ lies in the space location, or locations, of names (§) in expression ..§... .

The following 8 distinct spaces

```
      ((8↑⎕A)⎕NS¨⊂⍬)↳ #.A  #.B  #.C  #.D  #.E  #.F  #.G  #.H
```

can be organised using strand notation in an arbitrarily complex nested vector structure, for example

```
      ≡RArr←(A(B C(D E)))
¯3
```



Scalar assignment `RArr.V←99` pervades the nested array of refs. It has the effect of

```
      (A(B C(D E))).V↳99(99 99(99 99))
```

We can assign a set of numbers having this structure

```
      (A(B C(D E))).V←(99(1 2(55 66)))
      (A(B C(D E))).V↳(99(1 2(55 66)))
```

Or each element in the data structure can be an enclosed array.

```
      (A(B C(D E))).V←(99(1 2((2 4⍴55) (5 1⍴66))))
      D.V
55 55 55 55
55 55 55 55
```

11.2.1.3.1 Create a 3 by 4 array `A` consisting of 3 distinct unnamed namespaces. Assign variable `A.a` to some numbers. Assign `B` to a matrix with elements containing `A`. Check `B.a`.

11.2.1.3.2 Create an object vector of 26 `Form`s each with a `Button`,

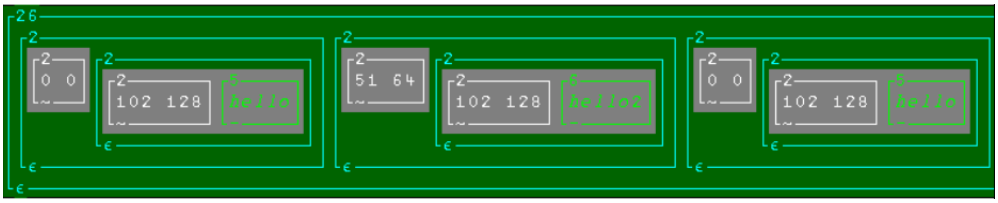
```
      RVec←⍎¨⎕A ⎕WC¨⊂'Form'
      (⎕A,¨⊂'.B')⎕WC¨⊂'Button'
      (25↑RVec).Posn←50+20×,⍳5 5
```

Change all the positions, sizes and captions in a single expression such as:

```
      RVec.B.(Posn Size Caption)←26⍴((0 0)(20 20)'hello')↵
                                  ((10 10)(20 20)'hello2')⍝
```

Display `cc` where

```
      cc←RVec.B.(Posn(Size Caption))
```



11.2.1.3.3 Redesign (not rewrite) your address book application STORE using namespaces such that data for the first individual in the list is found in

```
      Persons[1].(Name Address Text)
```

Invert the design such that variables in the namespace become namespaces, and what was a namespace becomes an element in a variable eg

```
      (Names Addresses Text).Person[1]
```

Is this design better? Maybe both is best?

§§ 11.2.2 Expanding Array.Array

§§§ 11.2.2.1 Expansion Rule

```
ñD1_D2←ñD1.ñD2                    ⍝ Expand deep space arrays to depth D2 within D1
```

Given a pure ref array $\tilde{n}_{D1}$ and another pure ref array $\tilde{n}_{D2}$ containing child objects of the corresponding elements of the first array, the result of dotting them together $\tilde{n}_{D1}.\tilde{n}_{D2}$ is a pure ref array of combined depth D_2 within D_1 such that the number of elements of the result is the product of the number of elements in each (operand) array.

11.2.2.1.1 Create a vector of references to 26 vanilla spaces.

```
RVec1←⍬⍬⍴⎕A ⎕NS¨⊂''
```

Then create a vector of references to 26 vanilla spaces in each of these.

```
RVec1.(RVec2←⍬⍬⍴⎕A⎕NS¨⊂'')
```

Use the WS Explorer to investigate the hierarchy. Notice the variable `RVec2` in every space in `RVec1`. Use **varChar** to assign and view the structure of the ref array expansion, for example, zoom out of

```
(5 5⍴RVec1).(5 5⍴RVec2)
```

The space arrays on the left and right of the dot (the operands) may be replaced by expressions that return space arrays. Therefore

```
⊢(⍬⍬⍴⎕nl 9).(⍬⍬⍴⎕nl 9) ≡ RVec1.RVec2
```

because

```
⊢RVec1.(⍬⍬⍴⎕nl 9) ≡ RVec1.RVec2
```

by Rule 5, and

```
⊢RVec1≡⍬⍬⍴⎕nl 9
```

Notice that dot binds tighter than primitive function match (≡), as one would expect of an operator.

Successive dot expansions follow Rule 2 as one would expect of an operator. The left-most expansion is performed first, followed by the next left-most expansion, etc… Thus the final number of spaces in expression `RArr1.RArr2.RArr3` is the product of the number of refs at each level.

For example, the expression

```
(⍬⍬⍴⎕NL 9).(⍬⍬⍴⎕NL 9).(RVec3←⍬⍬⍴⎕A⎕NS¨⊂'')
```

involves `26*3↳17576` spaces. And

```
⍴⊃,/⊃,/RVec1.RVec2.RVec3↳17576
```

11.2.2.1.2 What is the result of `⍴⊃,/⊃,/⊃,/RVec1.RVec2.RVec3.(⎕IO ⎕CT)` or, using *enlist* (∈), `⍴∈RVec1.RVec2.RVec3.(⎕IO ⎕CT)` assuming `⊢⎕ML≥1`

§§§ 11.2.2.2 New Idioms

We are now able to perform using APL primitives many new structural and data manipulations of arrays of spaces. We can, for example, set properties of arrays of GUI objects in succinct expressions such as

```
(F1 F2).(B1 B2).Caption←⊂'OK' 'Cancel'⍝ Set ↳ Button Captions
```

or we can dynamically create objects and manipulate their properties in the single expression:

```
(⍬⍬⍴'ABCD'⎕WC¨⊂'Form').(⍬⍬⍴'ÁÂÃÇ'⎕WC¨⊂'Group')↵
          .(⍬⍬⍴'abcd'⎕WC¨⊂'Button').Dragable←1⍝
```

Notice the wrapping (↵,⍝)

Having created these spaces and sub-spaces, we can construct arbitrary space structures like that produced by expression `(3 1⍴A B C).(2 2⍴Á Â).(a b c (⊂⊂,d))` from which we find

```
      ⍴¨ ¨ ¨ ¨ ¨(3 1⍴A B C).(2 2⍴Á Â).(a b c (⊂c,d))
                        1                                 1
                        1                                 1
                        1                                 1
                        1                                 1
                        1                                 1
                        1                                 1
```

We can assign a value (more than once – *eager* as opposed to *lazy* evaluation?) to variable `V` in every leaf space in

```
      (A B C∘.,B C D).(Á Â Ã∘.,Â Ã Ç).(a b c∘.,b c d).V←⍳2
```

or

```
      (3 1⍴A B C).(2 2⍴Á Â).(a b c(⊂c,d)).X←3 1⍴9 7 4
```

giving

```
      (3 1⍴A B C).(2 2⍴A B).(a b c(⊂c,d)).X
  9 9 9     9       9 9 9     9
  9 9 9     9       9 9 9     9
  7 7 7     7       7 7 7     7
  7 7 7     7       7 7 7     7
  4 4 4     4       4 4 4     4
  4 4 4     4       4 4 4     4
```

As seemed the case when APL 1 appeared, and again when APL 2 appeared, the new possibilities are endless. Two new idioms should be mentioned:

```
      RSc←(⍎⎕CS'').##                    ⍝ Returns scalar ref to parent space
```

```
      RSc←⍵.##                           ⍝ Returns scalar ref to current space
```

Remember ⊢⍵≡⎕NS⍬ where ⊢⍬≡⍬ .

11.2.2.2.1 Rewrite expression `(# #).(# #).(# #)` in 5 different ways without using dots.

§§§ 11.2.2.3 Generalised Modified Assignment

An arbitrary dotted variable (or stranded variable) structure may be used in a modified assignment.

```
      ñ_D.§[V..V]f₂←S_DS[V..V]Arr         ⍝ Modified strand assignment of dotted variable structure
```

In this case $\tilde{n}_D.\S^{V..V} f_2 \leftarrow S_D S^{V..V} Arr \Rightarrow \tilde{n}_D.\S^{V..V} \leftarrow \tilde{n}_D.\S^{V..V} f_2\ S_D S^{V..V} Arr$

If `X1`, `Y1` and `Z1` are variable names then an arbitrary name strand may be modified by a dyadic function and an array argument of suitable form. For example,

```
      (X1 Y1 Z1)+←1 2 3          ⍝ X1+←1 ⋄ Y1+←2 ⋄ Z1+←3
```

or

```
      (X1(Y1 Z1))+←1 2           ⍝ X1+←1 ⋄ Y1+←2 ⋄ Z1+←2
```

If these names are in space `N1`, then Rule 5 suggests that it should be possible to write

```
      N1.(X1(Y1 Z1))+←1 2        ⍝ N1.X1+←1 ⋄ N1.Y1+←2 ⋄ N1.Z1+←2
```

from outside space `N1`. Generalising further, we would expect that

```
      (N1 N2).(X1(Y1 Z1))+←1
```

should increment variables `X1`, `Y1` and `Z1` in both `N1` and `N2` by `1`, or that

```
      N0.(N1 N2).(X1(Y1 Z1))+←(1(2 3))(4(5 6))
```

would be equivalent to

```
      N0.N1.(X1 Y1 Z1)+←1 2 3 ⋄ N0.N2.(X1 Y1 Z1)+←4 5 6
```

This is indeed the case in Dyalog version 11.0. Further than this, selective modified assignment has been partially extended in a natural way to selective modified assignment.

Create 2 *Form*s each with 2 *Button*s

```
      (⍎¨'FG'⎕WC¨⊂'Form').(⍎¨'B1' 'B2'⎕WC¨⊂'Button')
```

and in the space of each button create 2 variables, *a* and *b*, with assigned values

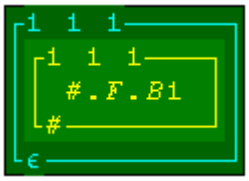
```
      (F G).(B1 B2).(a b)←↓↓2 2 2⍴⍳8
```

Then, selection can apply to the namespace references in access and assignment:

```
      (1 0/F G).(0 1/B1 B2).(a b)↳,⊂,⊂3 4
      (1 0/F G).(0 1/B1 B2).(a b)←,⊂,⊂9 10
      (1 0/F G).(0 1/B1 B2).(a b)↳,⊂,⊂9 10
```

The ref array may be any shape and structure,

```
      zz←(1 1 1⍴F G).(1 1 1⍴B1 B2)
```



as long as the shapes and structures correspond with that of the assigned data,

```
      (1 1 1⍴F G).(0 1/B1 B2).(a b)←1 1 1⍴⊂,⊂33 34
      (1 1 1⍴F G).(0 1/B1 B2).(a b)↳1 1 1⍴⊂,⊂33 34
```

However, the implementation is not yet finished for one would expect the following to work

```
      X←(1 1 1⍴F G).(0 1/B1 B2).(a b)
      (1 1 1⍴F G).(0 1/B1 B2).(a b)←X
      (1 1 1⍴F G).(0 1/B1 B2).(a b)+←X
RANK ERROR
```

although modified assignment with scalar extension works already

```
      (1 1 1⍴F G).(0 1/B1 B2).(a b)+←1
      (1 1 1⍴F G).(0 1/B1 B2).(a b+←1)
```

11.2.2.3.1 Discuss with your friends these and further generalisations of assignment and the class of the assignment arrow.

§§ 11.2.3 Expanding Array.Function

§§§ 11.2.3.1 Array.Niladic

A niladic function may be dotted with an array of object references, in which case the function is executed in every leaf in the array of references.

For example, we can create a non-simple ref array

```
      ⎕A⎕WC¨⊂'Form'
     ≡RArr←(A(B C(D E)))↳¯3
```

and execute niladic system function ⎕*WA* on each *Form*.

```
      RArr.⎕WA
100758844  100758556 100758540  100758368 100758352
```

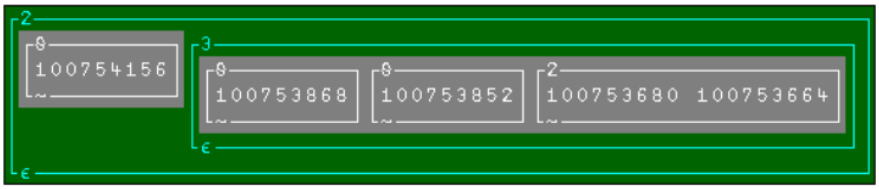
or

```
(A B).⎕WA ⇒ (A.⎕WA)(B.⎕WA)
```

or

```
⍴RArr.⎕TS↳2
```

When such a structure (`RArr`) is *dotted* with a variable name or niladic function, the variable name or function name pervades the namespace structure and the structure of the result reflects the structure of the namespace array.



A function in a namespace executes within that namespace; and normally only sees other functions and variables in that same namespace.

$R_D \leftarrow \tilde{n}_D.f_0$	⍝ f_0 pervades nested structure $\tilde{n}_D$

A nested array of namespace references, $\tilde{n}_D$ (depth D and dataType `RArr`) is *pervaded* by a niladic function in an analogous way to the way in which the primitive scalar functions *pervade* their nested arguments.

§§§ 11.2.3.2 Array.Monadic

A monadic function dotted with a space array *pervades* the space array structure in the same way as a niladic function or an arbitrary array expression does. The arguments, on the other hand, are *distributed* to spaces according to the space structure in the way that data is distributed to each space in the single name assignment $\tilde{n}_D$`.§←`S_D`Arr` where no strand expansion takes place.

$R_D \leftarrow \tilde{n}_D.f_1\ W_D$	⍝ Run f_1 in space $\tilde{n}_x$ with argument W_x

In the general case of $\tilde{n}_D$`.(...)` where `(...)` is a function expression that evaluates to a monadic function, the items of its argument array(s) are ***distributed*** to each referenced function. The structure of the argument W_D mirrors the namespace structure $\tilde{n}_D$ (and also reflects the *argument rank* of the function f_1). The structure of the result R_D also reflects the namespace structure $\tilde{n}_D$ (and the structure of the result when f_1 is applied to a typical argument – *ie* the *result rank* of f_1).

For example, a monadic scalar function, dotted with a vector of refs, may take a vector argument. The function is then applied to element I of the argument in element I of the space vector:

```
R1 R2←(N1 N2).f1 W1 W2 ⇒ ⊨(R1≡N1.f1 W1)∧(R2≡N2.f1 W2)
```

Note that variables W_D and R_D are taken to exist in the covering space, whereas f_1 is assumed to exist in all leaf spaces in $\tilde{n}_D$. If the arguments and results are to exist in the leaf spaces, then the expression $\tilde{n}_D$`.(R←`f_1` W)` could be used.

11.2.3.2.1 Examine the depth and structure of the results of

```
⎕SE.(cbbot cbtop mb popup tip NumEd).⎕NL 2
⎕SE.(cbbot cbtop mb popup tip NumEd).⎕NL 3
⎕SE.(cbbot cbtop mb popup tip NumEd).⎕NL⊂2 3
```

```
⎕SE.((cbbot cbtop mb)(popup tip NumEd)).⎕NL⊂⊂2 3
⎕SE.(cbbot cbtop mb popup tip NumEd).(⎕NL¨)⊂2 3
```

and

```
(#.F #.G).GetTextSize 'ab'
(#.F #.G).GetTextSize 'abc'
(#.F #.G).GetTextSize⊂'abc'
```

in the case where *F* and *G* are GUI *Form*s on the Root.

§§§ 11.2.3.3 Array.Dyadic

A dyadic function dotted with a space array *pervades* the space array structure in the same way as a niladic function, monadic function or an arbitrary array-expression does. The arguments, on the other hand, are *distributed* to spaces according to the space structure in the way that data is distributed to each space in the single name assignment $\tilde{n}_D.\S{\leftarrow}S_DArr$ where no strand expansion takes place. This distribution takes place for both left and right arguments.

$R_D{\leftarrow}A_D\ \tilde{n}_D.f_2\ W_D$	⍝ Run f_2 in space $\tilde{n}_x$ with arguments A_x and W_x

In the general case of $\tilde{n}_D.(\ldots)$ where $(\ldots)$ is a function expression that evaluates to a dyadic function, the items of its argument array(s) are ***distributed*** to each referenced function. In the dyadic case, there is a 3-way distribution among: left argument, reference array and right argument.

For example, a dyadic scalar function, dotted with a vector of refs, may take vector arguments. The function is then applied to element *I* of both arguments in element *I* of the space vector:

```
R1 R2←A1 A2(N1 N2).f2 W1 W2 ⇒ ⊨(R1≡A1 N1.f2 W1)∧(R2≡A2 N2.f2 W2)
```

11.2.3.3.1 Consider 2 *Form*s in a CLEAR WS.

```
'FG'⎕WC¨⊂'Form'
```

Create a *Button* and a *Label* on each *Form* via a dyadic, space-qualified use of ⎕*WC*

```
'BL'(F G).⎕WC¨⊂'Button' 'Label'
```

Set the *Caption*s on the child objects with space-qualified property assignment

```
(F G).(B L).Caption←('FB' 'FL')('GB' 'GL')
```

Change the positions of all the children using modified assignment

```
(F G).(B L).Posn×←(.5 .4)(.3 .2)
```

Set the Dragable property on all leaf objects

```
(F G).(B L).Dragable←1
```

Create 8 vanilla spaces in every leaf

```
(2⍴⊂2⍴⊂⎕AV[17+⍳8])(⍎¨↓⎕NL 9).(⍎¨↓⎕NL 9).(⎕NS¨)2⍴⊂2⍴⊂8⍴⊂''
```

Compare this with expression

```
(⍎¨↓⎕NL 9).(⍎¨↓⎕NL 9).((⎕AV[17+⍳8])⎕NS¨8⍴⊂⍬)
```

11.2.3.3.2 Experiment with dyadic ⎕*NL* in place of *plus* (+) in expressions like

```
(1 2)3 4(W(X Y)Z).+1 2(3 4)
```

using expressions like

```
'A'(W(X Y)Z).⎕NL 2 3
'Aa'(W(X Y)Z).⎕NL¨2 3
```

where the ⎕*NL* arguments have appropriate type, shape and rank.

§ 11.3 Arrays of Programs

§§ 11.3.1 Interpreting ..(...)..(...).f_1

We can now interpret an arbitrary dot-syntax expression. Having identified the dot-syntax sequence sub-expression, starting from the right we consider the rightmost token or parenthesised expression in ..(...) . This parenthesised name or expression preceded by a dot, is to be evaluated in the space preceding the dot. So the token or parenthesised expression in ..(...)... must evaluate to a scalar ref or an array of refs. If there is a dot to the left of this then the token or parenthesised expression in ..(...)..... must evaluate to a ref array,.. and so on until the left-most token or parenthesised expression is encountered. Only now can anything be evaluated. First the left-most term is evaluated in the current covering space to a space ref or array of space refs.. The next term is evaluated inside each of these spaces to give a set of sub-spaces .. and so on until the right-most term is revisited in the backward pass analysis. The right-most term can finally be evaluated because the space or spaces in which it is to be evaluated are now known. The result of this final term in the dot sequence may evaluate to a numeric, character, ref or mixed array if it is an array expression, or a function if it is a function expression, or any type of named object if it is a single token.

This is rather similar to the VB analysis of the line **ActiveSheet.Range'A1:A2'.Rows.Count** which is read left to right and where **Range'A1:A2'** returns an unnamed object reference. The APL equivalent, however, requires parentheses round the *Range* expression from Rules 4 & 5, as in

```
ActiveSheet.(Range'A1:A2').Rows.Count
```

or

```
ActiveSheet.(Range'A1:A2').Cells.(Item 1).Value2
```

or

```
Documents.(Open'C:\MyWord.doc').Activate
```

This procedural proscription in the analysis of a dotted sequence, together with the rules for expansion of terms, is sufficient to obtain the (derived) result of the dot sequence. If the result is a function then the rules for distribution of arguments must be applied in order to obtain the final (array) results of the entire (array) expression.

11.3.1.1 Create a *Calendar* within a *Group* on a *Form*.

```
'F' 'F.G' 'F.G.C'⎕WC¨'Form' 'Group' 'Calendar'
```

Write a niladic function *goo* in *F*-space which returns a ref to the *Group* space, then trace the expression

```
F.goo.C.DateToIDN 3↑⎕TS
```

Do the same for *F* and *C* and check the order of execution of *foo.goo.coo* .

11.3.1.2 Create namespaces *a b c* and *d* within *A B C* and *D* within *A B C* and *D*. Use **varChar** to zoom and view the structure of derived spaces such as

```
⊃⊃(A B C).(A B).(a b c d)
```

or

```
(3 1⍴A B C).(2 2⍴A B).(a b c (⊂⊂,d))
```

Notice that it has a 3 by 1 outer shape, then a 2 by 2 , then a 4 vector, the last element of which is doubly enclosed.

If, in particular, the final term in a dot-syntax expansion resolves to a monadic function then a function of that name (if it has a name) is assumed to exist in every leaf namespace found to the left of the final dot.

11.3.1.3 Interpret, or otherwise explain, the following lines:

```
      (N.+).×              ⍝ wrong
  #.N .+ .×
      N.(+.×)              ⍝ right
 #.N . +.×
      3 4(N.+).× 5 6       ⍝ should error
39
      3 4 N.(+.×) 5 6      ⍝ 3 4 +.× 5 6
39
```

§§ 11.3.2 Arrays of .. Arrays of defined Functions

Motivation: Physics deals with arrays of functions. The position of a particle is a 3-vector. Generally the position is a function of time – a 3-vector of functions **r**(t) = (x(t) , y(t) , z(t)). Even if the particle position is constant in one frame of reference, it is not necessarily constant in a different frame of reference. Since the laws of physics have to hold in all (inertial) frames of reference, the fundamental equations, *eg* **F** = m **a**, generally have to deal with arrays of functions.

Dyalog APL does not have notation to represent arrays of functions directly. However, there are ways in which arrays of functions can be represented. For example, functions can be represented as data via ⎕OR.

Consider the function expressions

```
      R←⌽∘⍋   ⍝ rotation 90° anticlock
      H←⊖     ⍝ reflection
```

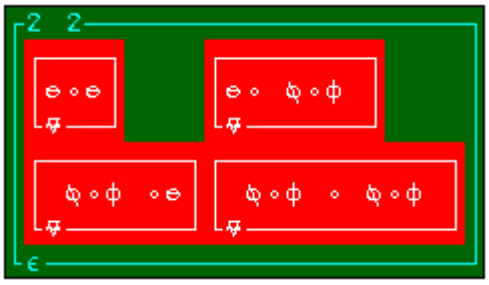
then

```
      ⎕OR¨'RH'
 ⌽∘⍋  ⊖
```

These functions could be combined into a 2 by 2 matrix *M* of (⎕ORs of) functions by snippet

```
M←2 2⍴⊂''
 :For r c :In ⍳⍴M
     ⍎'M',(⍕r),(⍕c),'←',('HR'[r]),'∘','HR'[c]
     M[r;c]←⊂⎕OR'M',(⍕r),(⍕c)
 :End
```

which in **varChar** looks like this:



This technique could be applied to canonical functions to produce arrays that can be indexed and individual functions ⎕FXed and executed with appropriate arguments.

Alternatively, an array of functions may be implemented more directly using an array of namespaces each of which contains **a different function of the same name**.

For example, if the position vector as a function of time t was given by **r**(t) = (t^{-1} , 2t , $3t^2$) then this could be implemented as

```
      S←⎕NS¨3⍴⊂''
      S[1].r←1∘÷
      S[2].r←2∘×
      S[3].r←3∘×∘(2∘(*⍨))
```

then the positions at the first 5 time points are given by the vector function `S.r` acting on each time:

```
      ↑S.r¨0,⍳4
0                0   0
1                2   3
0.5              4  12
0.3333333333     6  27
0.25             8  48
```

11.3.2.1 Make a matrix function of a scalar angle `⍵`, `Rot⍵` representing the rotation matrix in 2 dimensions:

⎛Cos ⍵ -Sin ⍵⎞
⎝Sin ⍵   Cos ⍵⎠

Use it to rotate the 2 element vector (2,3) clockwise through π/2 radians.

If Dyalog APL is to be a language suitable for scientific programming then it needs complex numbers. These can be modelled but would be better built into the interpreter as is done in APL2, SharpAPL and J.

Many other mathematical features could be built into the language, such as a monadic determinant function of a matrix for which Iverson has suggested notation `-.×⍵`, or the exponential function of a matrix defined by the power series expansion

```
      ((⍴⍵)⍴(1+⊃⍴⍵)↑1)+⊃⊃+/(+.×/¨(⍳⍺)⍴¨⊂⊂⍵)÷!⍳⍺
```

by analogy with the usual scalar definition

```
      1+⊃⊃+/(×/¨(⍳⍺)⍴¨⊂⊂⍵)÷!⍳⍺
```

where `⍺` is the number of terms in the power series and `⍵` is the square matrix to be exponentiated.

> Aside: Really big steps in mathematics are from scalar arithmetic to vector algebra and from vector algebra to tensor calculus. Linear vector spaces play a most fundamental role in mathematics, and in APL. If APL could efficiently and neatly handle arrays of functions then this would be another big step along the road of executable maths. For example, scientists habitually deal on paper with the determinant of matrices of functions, such as the Jacobian determinant, and even with the exponential of matrices of functions. With the new possibility of space-arrays of functions, some of these higher mathematical constructs start to become visible in APL.

§§ 11.3.3 Arrays of .. Arrays of defined Operators

The same technique as we used to model arrays of functions can be employed to model arrays of operators. (In 3D vector analysis, the gradient and curl operators are vector operators. See New Foundations in *Vector Vol.20 No.1* for some more discussion of operators in APL.)

Consider the simple derivative operator that returns the gradient function $^{df}/_{dx}$ of a given function f(x). $^{d}/_{dx}$ could be defined in APL as operator `∆` where

```
      f ∆ ↳ ((f x+⎕CT)-f x)÷⎕CT
```

or better as

```
      f ∆ ↳ ((f x+1E¯6)-f x-1E¯6)÷2×1E¯6
```

With this operator we can find the gradient function g(x)=$^{dy}/_{dx}$ of , for example, the function $y=x^2$ with

```
      g←2∘(*⍨)∆
```

and apply this function, which should be g(x)=2x, at the points x=1..9 to get

```
      g ⍳9
2 4 6.000000001 8 10 12 14 15.99999999 17.99999999
```

This operator (∆) may be placed in 3 different spaces, with just a small difference in each, to yield a model of a gradient vector operator that takes the partial derivatives in the x, y and z directions.

```
      D←⎕NS¨3⍴⊂''
      ⎕CS #.D[1]
    ∇ R←(f ∆)W;dW
[1]   dW←3↑⍨1E¯6      ⍝ (dx 0 0)
[2]   R←((f W+dW)-f W-dW)÷2×⍨1E¯6
    ∇
      ⎕CS #.D[2]
    ∇ R←(f ∆)W;dW
[1]   dW←3↑¯2↑⍨1E¯6   ⍝ (0 dy 0)
[2]   R←((f W+dW)-f W-dW)÷2×⍨1E¯6
    ∇
      ⎕CS #.D[3]
    ∇ R←(f ∆)W;dW
[1]   dW←¯3↑⍨1E¯6     ⍝ (0 0 dz)
[2]   R←((f W+dW)-f W-dW)÷2×⍨1E¯6
    ∇
      ⎕CS #
```

11.3.3.1 Test this vector operator on a scalar function of a vector ($x^{-1}+2y+3z^2$), ie `fn`,

```
    ∇ R←fn W
[1]   R←(÷W[1])+(2×W[2])+3×W[3]*2
    ∇
```

Show that the gradient function `(fn D.∆)` at 3 vector points (x,y,z) gives the expected answer.

```
      ↑fn D.∆ ¨(1 2 3)(2 3 4)(4 5 6)
¯1              2.000000002 18
¯0.2499999994   2.000000002 24
¯0.06249999984  1.999999995 36.00000001
```

This agrees well with the analytic solution ($-x^{-2}$, 2, 6z) at (x,y,z).

11.3.3.2 Please ask for the next module on **Dynamic Functions** – it's shorter and easier ☹!

Comment: If an operator can be dotted with an operator one might expect that `+.×` should be the equivalent of `#.+#..#.×`, or `(#.+)(#..)(#.×)`, but these latter expressions, unsurprisingly, cause a *SYNTAX ERROR* because `#..` is hard to interpret even if operators can take operator operands. Actually, no primitive operators may be space-qualified, but derived functions such as `#.(+.×)`, `#.(∘.×)` and even `#.(/)` can be. This is not a limiting factor as primitive operators (`. ∘ / \` etc) are identical in every space and so it should never matter from which space they were called. User defined operators, on the other hand, **may** be space-qualified. (Perhaps a bold version (•) of the relatively new and subtle addition `⎕AV[94+⎕IO]`, (·), should have been used instead of (.) for dot syntax.)