Sylvester's algorithm is expressed in APL with tolerance as left argument:

```
     ∇Z←T F X
[1]    ⍝(X≤T)/'→0,0⍴Z←⍳0'
[2]    Z←Z, T F X-÷Z←⌈÷X
[3]  ∇
```

Sylvester's example is:

```
      1E¯16 F 335÷336
2 3 7 48
```

In direct definition, this leads to a useful paradigm for writing recursive functions in APL:

```
      F:Z,αFω-÷Z←⌈÷ω : ω≤α ⍳0

      (○1) = +/÷1E¯16 F ○1
1
```

Roger Hui (in a personal communication) translated this into the *purely functional* form in J, using @. for *agenda*:

```
  f=. i.@8: ' (>.@% @ ] , [f ]->.&.%@]) @.<:

  le_16 f 335%336
2 3 7 48
```

The initial result of the function must be the identity element for the primary function, which for catenation is an empty array of the appropriate shape—in the case of Sylvester's algorithm this is an empty vector.

An example using recursion

A good way to introduce recursion is by one of the oldest of all algorithms: the calculation of pi by approximating inscribed (and circumscribed) polygons.[44] The symbol pi (π) was chosen by William Jones (1706) because pi is the length of the perimeter of a circle of unit diameter. An inscribed hexagon has 6 sides each of length 0.5, which gives 3 as the first approximation.[45]

Doubling the sides of the hexagon gives a better approximation, and further doublings give still closer values. The secret is, therefore, to compute the length of a new chord from the length of an old one, which is not difficult to do once the theorem of Pythagoras is known. *CH* gives the new chord as a function of the old one.

```
     ∇Z←CH X
[1]    Z←(0.5×1-(1-X*2)*0.5)*0.5
[2]  ∇
```

For a circle of unit diameter, the first approximation is given by the perimeter of a hexagon whose sides are each equal to the radius, i.e., the approximation to pi is 3.

After 8 doublings (8 applications of *CH*), pi is given by:

```
      6×(2×2×2×2×2×2×2×2)×CH
           CH CH CH CH CH CH CH CH 0.5
3.14159
```

We have a notation (exponentiation) that allows us to abbreviate this to:

```
      6×(2*8)×CH CH CH CH CH
           CH CH CH CH 0.5
3.14159
```

With APL we can use recursion to effect successive applications of the function *CH*:

```
     ∇Z←N C X
[1]    ⍝(N=0)/'→0, 0⍴Z← X'
[2]    Z←(N-1) C CH X
[3]  ∇

     ∇Z←PI N
[1]    Z←6×(2*N)× N C 0.5
[2]  ∇
```

In direct definition these functions can be given more concisely:

```
      CH:(.5×1-(1-ω*2)*.5)*.5
      C:(α-1)C CHω: α=0 : ω

      PI:6×(2*ω)× ω C 0.5

      PI 8
3.14159
```

Because Iverson's J includes primitives for square root (%:), halve (-:), and square (*:), and a conjunction (dyadic operator) for raising a function to a power (^:), we have the following formulation:

```
     ch=. '%: -: 1- %: 1- *: y.' : ''

     6*(2^8)*ch ch ch ch ch ch ch ch 0.5
3.14159

     6*(2^8)*(ch^:8) 0.5
3.14159
```