mcode guide

Statements

```
    ∇       R = (α,ω,δ) =>          //  ∇ functionname : localVarList  eg. ∇ f : a,b,c
                                    //  ∇.a for async fn
    ⍛       return R                //  nb. top level has a return value
    →       if (L) S                //  L implies S  S is expr or stmt or block
    ◇       else                    //  must follow → if
    ⍟       for (L of R) S          //  var ⍟ expr  : expr or stmt or block
    :       : S                     //  L ⍟ R : S       S is stmt or block
    ⍇       while (L) S             //  expr ⍇ expr     S is stmt or block
    ⍗       break ;                 //  only valid in loop
    ⍐       continue ;              //  only valid in loop
    ⍁       case R :                //  ⍁.s switch R  ⍁.d default R
    ⍂       R                       //  pass R to underlying language (useful for OOP etc)
    ⍝       //                      //  single line comment
    ←       L = S                   //  matrix and 1 line function expression assignment
    ;       ;                       //  stmt separator
    ⍢       class R                 //  ⍢ class : baseClass
                                    //      ⍢ name              method
                                    //      ⍢.s varList         simple constructor method
                                    //      ⍢.c varlist         general constructor method
                                    //          ⍢.t varlist     initialization
                                    //      ⍢.m class           mixin class
    ⍫       delete R ;              //  valid only for object.R in JavaScript, not vars
```

L is left, R is right, S is statement after : pr next { block }
use ← to assign a matrix containing code expressions, eg.

```
    M ← [   1, t0 × π,
            t1 × π, 1 ]
```

use ← to assign a named 1-line function expression, eg.

```
    avg ← { ( 0 + ≠ ⍵ ) ÷ ⍴ ⍵ }
```

Primitive Functions

```
    ⍞ mcode.output
    ⊢ mcode.nyi
    ⍎ mcode.mexec
    ⍙ mcode.create
    ∈ mcode.typeof
    ⍞ mcode.system
```

```
      ⍴ mcode.shape
      ↓ mcode.push
      ⍪ mcode.concatenate
      ⍳ mcode.iota
      ⌷ mcode.select
      ¨ mcode.each
      ⌿ mcode.reduce
      ⍣ mcode.power
     ?= mcode.assert
      ↑ mcode.pop
      » mcode.insert
      « mcode.remove
      ≡ mcode.match
      ≠ mcode.replace
      ⊃ mcode.split
      ⊂ mcode.join
      ∈ mcode.memberof
      ⍋ mcode.sort
      ⍒ mcode.sortDown
      ⍞ mcode.read
      ⍞ mcode.write
      ⍤ mcode.dom

   primitives and functions are called as  α fn ω
   expressions execute right to left as    α fn2 fn1 ω  same as  α fn2 ( fn1 ω )
   fn can have a modifier as               α fn.mod ω

   α ⍞ ω  is useful to trace function execution, eg.
   avg ← { ⍞ ( 'step 1' ⍞ 0 + ⌿ ω ) ÷ 'step 2' ⍞ ⍴ ω } ⋄ avg [ 1 2 3 ]
       step 1 = 6
       step 2 = [ 3 ]
       [ 2 ]

   ⍰       alone sets a breakpoint for the JavaScript debugger


Variables and Constants
   symbol  JavaScript              comment
   ------  ----------              -------

   α       _a                      // left function argument
   ω       _w                      // right function argument
   ð       _d                      // modifier or 3rd function argument
```

```
    θ       null                    // null value
    π       Math.PI                 // pi
    ė       Math.E                  // Euler e
    ⊙       _cp                     // shared object  same as  mcode.cp
    Δ       this                    // this  for object oriented programming


Vectorized Functions
    fn      fn ω                    α fn ω
    --      ----                    ------

    ≈       mcode.nyi               (a,w)=>a==w             // no-op, equality test
    ≠       mcode.nyi               (a,w)=>a!=w             // no-op, inequality test
    ⊭       (a,w)=>Math.trunc(w)    (a,w)=>+w.toFixed(a)    // truncate, special rounding
    ~       (a,w)=>+!w              mcode.rmset             // logical not, remove from set
    +       (a,w)=>+w               (a,w)=>(+a)+(+w)        // use +.  cvt, add  α +. ω
    -       (a,w)=>-w               (a,w)=>a-w              // negate, subtract  α -. ω
    ×       (a,w)=>Math.sign(w)     (a,w)=>a*w              // sign of number, multiply
    ÷       (a,w)=>1.0/w            (a,w)=>a/w              // reciprocal, divide
    /       (a,w)=>1.0/w            (a,w)=>a/w              // reciprocal, divide
    |       (a,w)=>Math.abs(w)      (a,w)=>a%w              // absolute value, α modulus ω
    ¦       (a,w)=>Math.round(w)    'norm'                  // round, norm of vec ω
    ⌈       (a,w)=>Math.ceil(w)     (a,w)=>Math.max(a,w)
    ⌊       (a,w)=>Math.floor(w)    (a,w)=>Math.min(a,w)
    ⊛       (a,w)=>Math.log(w)      (a,w)=>Math.log(w)/Math.log(a)  // log(base=α,x=ω) = y
    ?       (a,w)=>Math.random(w)   (a,w)=>Math.round(a*Math.random()) // uniform dist
    √       (a,w)=>Math.sqrt(w)     (a,w)=>Math.pow(w,1.0/a) // α √ ω
    **      (a,w)=>Math.exp(w)      Math.pow                // α **. ω
    !       mcode.factorial         mcode.binomial          // use !.
    sinθ    (a,w)=>Math.sin(w)      (a,w)=>a*Math.sin(w)
    cosθ    (a,w)=>Math.cos(w)      (a,w)=>a*Math.cos(w)
    tanθ    (a,w)=>Math.tan(w)      (a,w)=>a*Math.tan(w)
    asinθ   (a,w)=>Math.asin(w)     (a,w)=>a*Math.asin(w)
    acosθ   (a,w)=>Math.acos(w)     (a,w)=>a*Math.acos(w)
    atanθ   (a,w)=>Math.atan(w)     (a,w)=>a*Math.atan(w)
    atan2θ  'atan2'                 'atan2'
    ^       mcode.nyi               (a,w)=>a&&w             // logical AND   not binary AND
    v       mcode.nyi               (a,w)=>a||w             // logical OR    not binary OR
    <       mcode.nyi               (a,w)=>+(a<w)           // less than
    >       mcode.nyi               (a,w)=>+(a>w)           // greater than
    ≤       mcode.nyi               (a,w)=>+(a<=w)          // less than or equals
    ≥       mcode.nyi               (a,w)=>+(a>=w)          // greater than or equals
    ∘       mcode.nyi               'outer'
```

```
        these are applied to each vector and matrix element
        +.× inner product uses rules of linear algebra
        use dot after ~ + - * / | ** to specify vectorized function    eg. 1 +. [ 2 3 ]
            nb. if no dot then the built-in scalar op is used
            eg. 'a' + [ 2 3 ]  + is string concat


Operators
    symbol                          operator
    ------                          --------

    ¨                               // each
    ⌿                               // reduce
    ⍣                               // power

        operators are functions that operate on functions,  syntax is  L function operator R

        0 ⌈ ⌿ [ 2.1 3.7 ]           // ⌈ max reduction of list      is 3.70000
            nb. Left arg 0 selects the function ⌈ max, not function ⌈ ceiling

        ⌈ [ 2.1 3.7 ]               // ceiling over list            is [ 3 4 ]
            nb. Each operator ¨ could be used, but ⌈ is already vectorized

        'ab' ≠.xy ¨ [ 'ab cd' 'ab ef' ]                             is [ xy cd xy ef ]

        ⊙.r = 'new'
        'ab' ≠.⊙.r ¨ [ 'ab cd' 'ab ef' ]                            is [ new cd new ef ]
            nb. for each element in ⍵ replace ab with shared variable r


Built-in Scalar Operations
    symbol  JavaScript              comment
    ------  ----------              -------

    =       _                       // assignment
    ≒       ===                     // exact equality test
    ==      _                       // built-in equals
    !=      _                       // built-in not equals
    !==     _                       // built-in not identical
    !       _                       // not (unary)
    ~       !                       // not (unary)                          reflexive
    ∨       ||                      // logical or   nb. not ∨ is ∨          reflexive
```

```
    ^       &&                      // logical and  nb. shift 6 as exp      reflexive
    +       _                       // addition                             reflexive
    -       _                       // subtraction (unary)                  reflexive
    *       _                       // multiplication                       reflexive
    **      _                       // exponentiation                       reflexive
    %       _                       // modulus                              reflexive
    /       _                       // divide                               reflexive
    ??      _                       // nullish (undefined or null)
    <       _                       // less than
    >       _                       // greater than
    ≤       <=                      // less than or equals
    ≥       >=                      // greater than or equals

        _  means no change to symbol in this table
        reflexive  means do operation, then do assignment    eg. i += 2  means i=i+2
        ++ and -- are reflexive increment and decrement      eg. i++     means i=i+1
        = is general assignment  ≒ is exact equality test like ===
        regexp: use /s for space, since / means divide

[end]
```