mcode JavaScript transpiler

```
// mcode JavaScript transpiler
// Copyright David Remba 2022, 2023, 2024
// Open Source under Apache 2.0 License

"use strict"; // nb. soft tabs 4
/* globals JSHINT: false, */

function mcode ( mcodeOptions = {} ) {  // see coreLoader below for mcode options

    let mcode_version = 'version 0.07.31.2024 EXPERIMENTAL';

    let mcodeUseCache = 0;

    let mcodeDebug = '';
    // common debug options:  '' off
    // 'I' for primitives and no runtime library loaded
    // 'deE' for expressions    'deP' for parser

    // mcodeOptions:
    /*
        debug:      see below
        log:        fn to log output
        onload:     fn to call after mcode is loaded
        help:       fn to call for a help message
        msg:        fn to call to post an IDE status change 'ready', 'error', or a 1 line message

    mcodeOptions.debug
    affects entire session or 1 mexec call
        e       exec with no debug infog
        me      exec and show input mcode           *
        eI      exec and show primitive initialization
        eC      exec and resets cursor to start of output
        de      for code generated                  *
        deL     for lexer debug                     **
        deE     for expression generator debug      **
        deP     for parser debug                    **
        S       to insert source into output code
        *   may create a lot of output
        **  may create an extreme amount of output, best to use small tests

        nb. any of these options may be combined
        nb. these are the same options for mexec
    */

    let mc = {};                            // mc is for internals, mcode is the public API
    mcode.href = '';                        // html location of mcode system, for coreLoader

// initialization for JavaScript transpiler
let init = () => {

    logn('mcode nano-transpiler '+mcode_version);

    mcodeOptions.debug = '';
    if (mcodeDebug!='') {
        trn('mcodeDebug',mcodeDebug);
        mcodeOptions.debug = mcodeDebug; }

    mc.lexerDebug = mc.exprDebug = mc.parserDebug = mc.insertSource = 0;
    mc.initDebug = mcodeOptions.debug.indexOf('I') >= 0;
    mc.jshintFixMarker = 'E030 fix';
    mc.tperr = 0;                       // transpiler error
    mcode.fn = '';                      // last run function

    let tables = () => {
    /*
    statements are evaluated top down and Left to Right
    Expressions are evaluated Right to Left
    input   output                  notes
    */
    mc.stmtsIn = `
    ∇       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
    `;

    // variable & constant replacements
    mc.vmapIn = `
    α       _a                      // left function argument
    ω       _w                      // right function argument
    δ       _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
    `;

    // function mapping from primitive symbosl to underlying language runtime for scalar data
    // used by vectorized math functions, and inner and outer product
    //
//  mcode   op ω                    α op ω                  notes
    mc.fmapIn = `
    ≈       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
∨       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'
`;

// infix or prefix (unary) builtin functions or tokens that are dyadic: α op ω
mc.bmapIn = `
=       _                       // assignment
⥊       ===                     // exact equality test
==      _                       // built-in equals
!=      _                       // built-in not equals
!==     _                       // built-in not identical
!       _                       // not (unary)
~       !                       // not (unary)                          reflexive
∨       ||                      // logical or   nb. not v 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
    `;
    // nb.  _ means no replacement for symbol

    // ≈       ==          // equality test with type conversion
    // ==      x           // built-in equality with type conversion
    // ≠       !=          // not equals with type conversion

    // mc.operatorRe = /¨|⌿|⍀|⍣|@/;   // fn operators
    mc.oprIn = `
    ¨                               // each
    ⌿                               // reduce
    ⍣                               // power
    `

    }; tables();     // define transpiler tables

    let maps = () => {

        // convert tables above into maps used by transpiler for source code replacements
        // TblFn: given table t and regexp re, apply fn to each re match in t
        let
        TblFn = ( t, re, fn ) => {
            // nb. [... x] converts a result with an iterator into an array
            let m = [...t.matchAll(re)];
// alert('m = '+jstr(m));
            return m.forEach( fn );
        },
        // make an array of replacements rp for use in: string.replace(s,...rp)  or  tbl.replaceAll(t,...rp)
        Rp = c => {
            let r = []; for (let b of c.split('  ')) r.push(b.split(' ')); return r;
        },
        // apply replacement array to a table
        Rpl = (tbl,c) => {
            // let b = Rp(c), r = tbl.replaceAll(...b);
            for (let b of c.split('  ')) { b = b.split(' '); tbl = tbl.replaceAll(...b); }
            return tbl;
        };

        mc.smap = new Map();                            // statement map
        let s = Rpl( mc.stmtsIn, 'S _S  L _L  R _R  α _a  ω _w  δ _d');
        TblFn( s, /^\s*(\S+)\s+(.+(?=\/\/))/gm,         // nb. (?= is pos lookahead, non-capturing group
            e => {
                let s = e[2].trim(),                    // statement
                lt = (s.match(/_L/g) || []).length,     // left token count
                rt = (s.match(/_R/g) || []).length;     // right token count
                mc.smap.set( e[1], {s:s, lt:lt, rt:rt} );
            } );

        mc.vmap = new Map();                            // variable & constant map
        TblFn( mc.vmapIn, /^\s*(\S*)\s+(\S+)\s+(.*)/gm,
            e => mc.vmap.set( e[1], e[2] ) );

        mc.bmap = new Map();                            // built-in function map
        mc.rmap = new Map();                            // reflexive built-in function map
        TblFn( mc.bmapIn, /^\s*(\S*)\s+(\S+)\s+(.*)/gm, // sets bmap and rmap
            e => {
                // trn('e',e.slice(1));
                mc.bmap.set( e[1], e[2] );
                if ( e[2] != '_' ) mc.bmap.set( e[2], '_' );
                if ( /reflexive/.test(e[3]) ) mc.rmap.set( e[1]+'=', '_' );
            } );

        mc.fmap = new Map();                            // function map for vector functions
        TblFn( mc.fmapIn, /^\s*(\S+)\s+(\S+)\s+(\S+).*/gm,
            (e) => { mc.fmap.set( e[1], {m:e[2],d:e[3]} ) } );

        mc.omap = new Map();                            // function map for operator functions
        TblFn( mc.oprIn, /^\s*(\S+)\s+\S+\s+(.*)/gm,
            (e) => { mc.omap.set( e[1], e[2] ) } );

        mc.pmap = new Map();                            // primitive map
        mc.cmap = new Map();                            // code map for mcode sources

        mc.mtxRpl = Rp('( [  ) ]  , ][');               // matrix notation replacements: M(x,y) to M[x][y]
```

```
        // expose language elements for syntax
        mcode.language = {};


    }; maps();         // init mcode parser maps

    mcode.guide = () => {
        let r = '', glog = m => r += m + '\n';
        // s.replaceAll(/\</g,'<').replaceAll(/\>/g,'>');
        // .replaceAll(/\//g,'&sol');

        glog(`Statements`);
        glog( mc.stmtsIn );

        glog(`  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 + ⌿ ω ) ÷ ρ ω }
`);

        glog(`
Primitive Functions`);
        {
            const t = mc.pmap.entries();
            for (const e of t) {
                glog('\t'+e[0]+' '+e[1]);
            }
        }
        glog(`
    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' ⎕ ρ ω } ; avg [ 1 2 3 ]
        step 1 = 6
        step 2 = [ 3 ]
        [ 2 ]
    ⍠       alone sets a breakpoint for the JavaScript debugger
`);

        glog(`
Variables and Constants
    symbol  JavaScript              comment
    ------- -----------             --------`);
        glog(  mc.vmapIn );

        glog(`
Vectorized Functions
    fn      fn ω                    α fn ω
    --      ----                    ------`);
        glog( mc.fmapIn );
        glog(`        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
`);

        glog(`
Operators
    symbol                          operator
    ------                          ---------`);
        glog(mc.oprIn);
        glog(`        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 ω replace ab with shared variable r
`);
```

```
        glog(`
Built-in Scalar Operations
    symbol  JavaScript              comment
    ------  ----------              -------`);
        glog(mc.bmapIn);

        glog(`        _  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
`);

        glog('[end]');
        return r;
    }

    mcode.guideX = (full=0) => {
        log(`
mcode guide, from JavaScript transpiler tables


    Statements
    symbol  JavaScript              comment
    ------  ----------              -------`);
        trn('',mc.stmtsIn);
        log(`        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 + ⌿ ω ) ÷ ρ ω }`);

        log(`


    Primitive Functions
`);
        {
            const t = mc.pmap.entries();
            for (const e of t) {
```

```
            log('\t\t'+e[0]+' '+e[1]+'\n');
        }
    }
    log(`
    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' ⎕ ρ ω } ; avg [ 1 2 3 ]
        step 1 = 6
        step 2 = [ 3 ]
        [ 2 ]

    ⍠       alone sets a breakpoint for the JavaScript debugger`);

    log(`


Variables and Constants
symbol  JavaScript              comment
------  ----------              -------`);
    trn('',mc.vmapIn);

    log(`
Vectorized Functions
fn      fn ω                    α fn ω
--      ----                    ------`);
    trn('',mc.fmapIn);
    log(`        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`);

    log(`


Built-in Scalar Operations
symbol  JavaScript              comment
------  ----------              -------`);
```

```
        trn('',mc.bmapIn);

        log(`        _  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
`);
        log(`

    Operators
    symbol                          operator
    ------                          ---------`);
        trn('',mc.oprIn);
        log(`        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 ω replace ab with shared variable r

`);

        if (full) {
            trn('smap',mc.smap);
            trn('rmap',mc.rmap);
            trn('cmap',mc.cmap);
        }
        nl();
    };        // guide for mcode from tables
    // mcode.guide(1);

    let rtl = () => {
```

```
mcode.addPrim = (a,w,d) => {    // add a primitive
    // nb. the new primitive will not be seen by parser
    // until current code (string or file) is re-evaluated, eg. in another ⍎ mexec
    mc.pmap.set(a,w);
    if (mc.initDebug) log(a+' '+w+ ( d ? ' '+d : '') + '\n'); // show primitives as they are defined
};
// used as ⍙ mcode.create in core

mcode.output = (a,w,d) => {                                 // ⎕  log output to console and quad functions
    // trn('output a',a); trn('w',w); trn('d',d);
    if ( d == null ) {                                      // trace output
        if ( a === null ) tr('',w); else tr(a,w);           // 'w'  or  'a' = w
        nl(); return w;
    } else if ( d == 'j' ) {                                // raw JSON output (convert to string)
        if (a != null) log(a+' = '); log(jstr(w)); nl(); return w;
    } else if ( d == 'log' ) {                              // raw log output (convert to string)
        if (a != null) log(a+' = '); log(w); nl(); return w;
    } else if ( d == 'nnl' ) { log(w); return w.length;     // no newline
    } else if ( d == 'src' ) {                              // ⎕.src return mcode source (only if parsed)
        if ( w == '' ) {
            let c = [...mc.cmap.keys()];                    // nb. spread expansion
            c.sort(); return c;
        } else {
            w = w.replace('_cp','⊙');                       // for context objects
            return mc.cmap.get(w) ?? null;
        }
    }
    return null;
};
mcode.addPrim( '⎕', 'mcode.output', 'M' );

mcode.nyi = (a,w,d) => {                    // ⊢  not yet implemented stand-in or diagnostic
    if ( 1 || mcode.mexecDebugLevel ) {
        tr('mcode.nyi a',a); tr('w',w); trn('d',d); }
    return w;
};
mcode.addPrim( '⊢', 'mcode.nyi' );          // no-op, not yet implemented
mcode.vf = mcode.nyi;                       // vector function, will be overloaded in core

// stand-ins for testing
if (mcodeOptions.debug.indexOf('d') >= 0) {
    logn('adding standins');
    mcode.create    = mcode.nyi;            mcode.addPrim('⍙','mcode.create');
    mcode.shape     = mcode.nyi;            mcode.addPrim('⍴','mcode.shape');
    mcode.typeof    = mcode.nyi;            mcode.addPrim('⍷','mcode.typeof');
    mcode.each      = mcode.nyi;            mcode.addPrim('¨','mcode.each');
    mcode.reduce    = mcode.nyi;            mcode.addPrim('⌿','mcode.reduce');
    mcode.power     = mcode.nyi;            mcode.addPrim('⍣','mcode.power');
    mcode.iota      = mcode.nyi;            mcode.addPrim('⍳','mcode.iota');
    mcode.read      = mcode.nyi;            mcode.addPrim('⍃','mcode.read');
    mcode.system    = mcode.nyi;            mcode.addPrim('⍠','mcode.system');
    mcode.format0   = mcode.nyi;
}

mcode.help = (a,w,d) => {                   // intrinsic help function, gets overloaded
    // tr('help a',a); tr('w',w); trn('d',d);
    if (mcodeOptions.help) mcodeOptions.help();
    return '';
};

// set language regexs for syntax highlighting
mcode.setLang = () => {
    // let esc = '?+*!^~|&'.split(''),          // escape needed for regexp
    let getmap = (s,m) => {
        let r = '',c;
        for (let k of mc[s].keys()) {
            // mcode.trn('k',k);
            if (r!='') r += ' ';
            // if (k=='||') k = '\\|\\|';
            r += k; }
        // for (c of esc) r = r.replaceAll(c,'\\'+c);
        mcode.lang[m] = r;
    }
    mcode.lang = {}
    getmap('smap','stmts');  // these names shold match the highlighting classes in IDE
    getmap('pmap','prims');
    getmap('vmap','vars');
    getmap('fmap','fns');
    getmap('omap','oprs');
    getmap('bmap','builtins');

    // js
```

```
            mcode.lang.js = 'let for of if else return while continue break throw length push pop \\\( \\\)';
            // mcode.lang.builtins = '//|=>|' + mcode.lang.builtins;
            // mcode.trn('setLang',mcode.lang);
        };

        // mcode.guide();                              // debug
    }; rtl();           // mcode runtime library

}; init();

// lexer: operates in 2 passes: statements, and expressions
let lexer = () => {
        let c, cn, cp, cpp, i, t, r, lsc, ms, ls, il, ila, st, stm, ex,
            ln, lnc, lna, rc, unit,
        d = 0, // debug flag
    dbg = (p = '') => {
        if (!d) return;
        // create temp object and show contents
        if (p) log( p+' ' );                // prefix
        tr('', jstr([cp,c,cn]) );           // character checking
        tr('', {ls:ls,lsc:lsc,il:il});      // indent processing checking
        tr('st',st);                        // single statement flags
        tr('stm',stm);                      // multi statement flags
        tr('lnc',jstr(lnc));                // line accumulator
        // tr('t',t);                       // token accumulator
        // tr('r',jstr(r));                 // output accumulator
        nl();
    },
    reset = () => {
        // c                                current char scanned
        // d                                debug flag
        r = [];                             // result lexed token list
        // lsc                              leading space count
        il = 0;                             // indent level
        ln = -1;                            // line nr
        lnc = '';                           // line contents
        lna = [];                           // line number array
        // i                                // index into input string
            ms = 0;                             // indent size from first indent seen
            rc = 0;                             // result code
        ila = [];                           // indent level array
        // ex                               // expression mode if true
        unit = '';                          // code unit, eg. function
        stm = { bq: 0, cmt: 0, ar: 0 };     // reset multiline state flags
        mc.src = '';                        // reset source code accumulator
    },
        resetLine = () => {
            cn = '',            // next char (lookahead)
            cp = '',            // prev char
            cpp = '',           // prev prev char
            t = '',             // current token
            lnc = '',           // current line contents
            // reset state  nb. other keys are set to 0
            st = {};            // reset single line state flags
            lsc = 0;            // leading ws count
            ls = ex ? 0 : 1;    // leading space flag
        },
        term = () => {
        // if (d) trn('  term t',jstr(t));
            t = t.trim();
                if ( t.length > 0 ) {
                    r.push(t);
            t = '';
            }
        },
        isws = (c) => c == ' ' || c == '\t',
    chkMultiline = (ex) => {
        lnc += c;                                   // accumulate line content

        if ( c=='\n' ) ln++;                        // count lf always for src code

        // handle /* ... */ comment, multiline array, and unescaped backquote
        // dbg('M');
        if ( st.qs || st.qd || st.re ) return 0;

        // stopping
        if ( stm.bq && c == '`' && cp!='\\') { t += c; if (ex) term(); stm.bq = 0; return 1; }
        if ( stm.cmt && c == '*' && cn == '/') {
            t += '*/'; if (ex) term(); i++; stm.cmt = 0; return 1; }
        if ( stm.ar && c == ']' ) { t += c; if (ex) term(); stm.ar = 0; return 1; }

        // accumulating
        if ( stm.bq || stm.cmt ) { t += c; return 1; }
        if ( stm.ar ) { t += c; return 1; }
        if ( st.cmt && c != '\n' ) { t += c; return 1; }    // if one-line cmt accum only

        // starting
        if ( c == '`' && cp != '\\' ) { if (ex) term(); t += c; stm.bq = 1; return 1; }
        if ( c == '/' && cn == '*' ) {
            if (ex) term(); t += '/'; stm.cmt = 1; return 1; }
        // start multiline matrix
        if ( !ex && isws(cp) && c == '[' && cn != ']' && isws(cn) ) {
            t += c; stm.ar = 1;  return 1; }

        return 0;
    },
        chk = (x,k) => {
            rc = 0;
            if ( !(k in st) ) st[k] = 0;        // init
        // if (d) trn('chk',{x:x,k:k});

        // quotes in quotes bypass
        if (st['qs'] && x=='"') return 0;
        if (st['qd'] && x=="'") return 0;

        let isEsc = cp == '\\' && cpp != '\\';

        // regexp must have preceding ws and not // or /* */ comment
        if ( x == '/' && c == x ) {
            if (!st[k] ) {
                // regexp start requirments:
                if ( !isws(cp) ) return 0;              // preceding space
                if ( cn == '/' || st.cmt || cp == '*' ) // not // or not already cmt or not /*
                    return 0;
                if ( isws(cn) ) return 0;               // not / ws
                st[k]++;                                // is regexp
            } else if ( cp != '\\' ) st[k]--;           // end unescaped regexp
            // trn('rf',st[k]);
        // } else if ( c == x && cp != '\\') {             // unescaped char
        } else if ( c == x && !isEsc ) {                // unescaped char
            st[k] ^= true;
            // ending delimiter, and adj for trailing space
            if ( !st[k] ) { t += c; if (ls) lsc--; }
```

```
                rc = 1;
            }
            // starting and inside protected string
            if ( st[k] ) { t += c; rc = 1; }
        // dbg('  '+k+' ');
            return rc;
        },
    chkProtected = () => {
        // dbg('');
        if ( stm.cmt || stm.bq )    return 0;
        if ( chk("'",'qs') )        return 1;
        if ( chk('"','qd') )        return 1;
        if ( chk('/','re') )        return 1;
        return 0;
    },
    idErr = (s, ln, unit) => {
        return (!!unit ? ' in "'+unit+'"' : '') +
            (ln ? ' at line '+(ln+1) : '') + ':\n' + s;
    },
    chkErrors = ( s, unit='' ) => {
        let s0 = idErr(s, ln, unit);
        if ( st.qs || st.qd )   throw 'unclosed string'+s0;
        if ( st.re )            throw 'unclosed regexp'+s0;
    },
    chkErrorsMulti = ( s, unit='' ) => {
        let s0 = idErr(s, ln, unit);
        if ( stm.cmt )          throw 'unclosed comment block'+s0;
        if ( stm.bq )           throw 'unclosed backquote'+s0;
    },
    setIndent = () => {
        // let d = 1;  // local debug
        // if (d) logn('  indent');
        if ( ls ) {
            if (c=='\t') throw 'leading spaces required, not tabs for indenting\n'+
                    'leading tab seen at line '+ln+' in '+unit+'\n';
            if ( isws(c) ) lsc++;
            else if (c!='\n') {
                if (ms==0) ms = lsc;        // set indent size
                if (ms) il = Math.floor( lsc/ms );
                if (d) trn('  il ',jstr({il:il,lsc:lsc,ms:ms}));
                ls = 0;
            }
        }
    },
    chkCmt = () => {
        // handle // 1 line comment
        if (c == '/' && cn == '/') {
            term(); r.push('//'); lnc += '/'; st.cmt = 1; i++; return 1; }
        return 0;
    },
    chkToken = () => {
        if ( !isws(cp) && isws(c) ) {                           // break on ws
            term(); return 1;
        } else if ( c == ',' && ( isws(cn) || cn == '\n' ) ) {  // comma sep in matrix
            term();
        }
        return 0;
    },
    lexDump = (r,ila,lna) => {
        let ln = 0, e, i,
        startLine = () => {
            let n = ila[ln]||0, p = n > 0 ? ' '.repeat(n*4) : ''
            log( ln+' '+n+' '+(lna[ln]||0)+': '+ p );
            ln++;
        };
        logn('ln ila lna');
        for (i=0; i<r.length; i++) {
            e = r[i];
            if ( i == 0 ) startLine();
            if ( e != '\n' ) log(jstr(e)+' ');
            else { log('EOL \n');
                if (i+1<r.length) startLine(); }
        }
        nl();
    },
    srcDump = () => {
        log('src dump:\n');
        log(mc.src);
        log('\n');
    },
    accumStmtTerm = () => {
        // nb. statement terms are not dynamic, unlike primitives which are dynamic
        let isTerm = c && mc.smap.get(c);
        // nb. : and no-space is NOT a term since it is also ternary x ? y : z
        if ( c == ':' && (!isws(cn) && cn!='\n') ) isTerm = 0;
                if (isTerm) {
                    term(); r.push(c);
                } else {
                    t += c;                                 // accumulate term
                }
    },
    saveCode = () => {
        if ( mc.src != '' && mc.src != '\n' ) {
            // if (d) { tr('saveCode: unit',unit); trn('mc.src',jstr(mc.src)); }
            mc.cmap.set(unit==''?'lambda':unit,mc.src); mc.src = '';
            if (unit!='') mcode.lastUnit = unit;
        }
    },
    doEOL = () => {
        if ( c == '\n' ) {
            dbg('EOL');
            if (/^▽/.test(lnc)) saveCode();         // save prev fn, if any
                mc.src += lnc;                          // save src
            let m = lnc.match(/^▽\S*\s+(\S+)/);     // set unit from '▽ unit'
            if ( m != null ) {
                unit = m[1] ?? '';
                // if (d) trn('unit',unit);
            }
            chkErrors(lnc, unit);
                    lna.push(ln); ila.push(il); term(); resetLine(); r.push('\n');
            // ln++ is done in chkMultiline, not here, to account for `...\n...`
            return 1;
        }
        return 0;
    },
    lex = (s, mode, unitp='') => {
        d = mc.lexerDebug;                          // debug flag
        ex = mode == 'expr';
        if (ex) unit = unitp;
        if (d) trn('lex '+mode+' s',s);
        if ( typeof s !== 'string' ) return ex ? [] : { r:[], ila:[], lna:[] };
        reset();
        resetLine();
        for (i=0; i<=s.length; i++) {
            // d = mc.lexerDebug && ln > 29 && !ex;     // selective debug
            c = s[i] || ''; cn = s[i+1] || ''; cp = s[i-1]||''; cpp = s[i-2]||'';
            dbg(mode);

            if (chkMultiline(ex))           continue;
            if (!ex) setIndent();
            if (doEOL())                    continue;
            if (chkProtected())             continue;
            if (chkCmt())                   continue;

            if (ex) {                       // expression
                if ( chkToken() )           continue;
                           t += c;                     // accumulate expr term
            } else accumStmtTerm();
               }
               term();
        dbg('end');
        if (!ex) {                              // statements:
            // zvzv NIU
            // while (il>0) { r.push('\n'); ila.push(il); il--; }  // close pending indents
            // r.push(''); ila.push(0); lna.push(ln);              // for lookahead in parser
            if (d) { lexDump(r,ila,lna); srcDump(); }
            chkErrors(lnc, unit);
            chkErrorsMulti(lnc, unit);
            saveCode();                         // save last fn or expr
        } else {
            chkErrors(s, unit);
        }
            if (d) trn('lexer '+mode+' r',jstr(r)); if (d) nl();
        return ex ? r : { r:r, ila:ila, lna:lna };       // return array only for expr
    };
    mc.lex = lex;                               // lexer, may throw error
    mc.lexDump = lexDump;
}; lexer();

// expression builder: constructs a tree, then does depth-first-visitation
let expr = () => {
    let level, t,
        d,                  // debug flag
        unit = '',          // code unit
    reset = () => {
        level = 1;          // expr tree or emit levels   nb. start at 1 for parsePgm
        t = 0;              // tree term counter
        d = mc.exprDebug;   // debug flag
    },
    di = (a,b) => {     // debug with indent
        if (!d) return;  trn(' '.repeat(level*4)+a,b);
    },

    // build infix tree as  [ L op mod R ]  from expression token stack (es)
    //     ( ) creates subtrees using recursion
    //     arrays are re-assembled as subarray args
    //     { } creates lambda fn, as subtrees using recursion as  [ null ← mod R ]
    //
    peek = st => {
        return (st.slice(-1) ?? [])[0] ?? null; },  // peek at next item to be popped from array
    isIdentifier = e => /^[a-zA-Z\$_∆⊙]/.test(e),   // is fn or a.fn or this or ctx
    isBuiltin = (op,mod) => {
        let rc = 1;
        di('isBuiltin',{op:op,mod:mod});
        if ( op==null || op instanceof Array ) rc = 0;
        // if builtin function or variable/constant or keyword
        else if ( typeof op != 'string' ) rc = 0;
        else if ( mod!=null ) rc = 0;               // built-ins do not have modifiers
        else if ( !mc.bmap.get(op) && !mc.rmap.get(op) && !/\w_$|\w\($/.test(op) ) rc = 0;
        di('isBuiltin',{op:op,rc:rc});
        return rc;
    },
    // quote if does not have ⊙ (local context) and not numeric
    isData = x => (!isNaN(x)) || /^[`'"]/.test(x),
    quoteArg = x => ( !/\./.test(x) && isNaN(x) && /^[^`'"]/.test(x) ) ? '\''+x+'\'' : x,
    splitOpMod = s => {
        if ( typeof s != 'string') return null;
        let m = s.match(/(\S+?)(\.\S*)/),           // split op .mod (not greedy)
            op = s, mod = null;
        if (m) {
            if ( /^⊙/.test(op) ) op = m[0];         // call to local method
            else { op = m[1]; mod = m[2]??null;
                if (mod!=null) mod = mod.slice(1); }
        }
        di('splitOpMod',{op:op,mod:mod});
        return [op,mod];
    },
    isFn = e => {
        if ( typeof e != 'string') return 0;
        di('isFn e',e);
        if ( /^['"\`]/.test(e) ) return 0;           // string literal
        let [op,mod] = splitOpMod(e), r;
        r = isBuiltin(op,mod) || !!mc.pmap.get(op) || !!mc.fmap.get(op);
        di('isFn',r);
        return r;
    },
    // isOperator = e => mc.operatorRe.test(e),     // fn operators
    isOperator = e => !!mc.omap.get(e),                // fn operators
    exTree = (es) => {  // nb. recursive
        let
        L = null,       // left term
        R = null,       // right term
        op = null,      // operation
        mod = null,     // modifier
        V = null,       // assembled node  L op mod R   nb. mod may be omitted
        ac,             // array contents accumulator
        collectDataVector = E => {                       // sets output in ac
            // nb. since comma sep is not reqd, each element is NOT an mcode expr
            if ( E != ']' ) return 0;                    // not a vector
            ac = ''; if (d) di('  collectDataVector E', jstr(E));
            while (es.length > 0) {
                E = es.pop();                            // di('  E',E);
                if ( E == ']' ) throw 'vector error: use M ← for matrices, unit = '+unit;
                if ( E == '[' ) break;                   // done
                if ( /,|\n/.test(E) ) continue;          // if (d) tr('E',E);
                let sep = !/,$/.test(E) ? ', ' : ' ';
                ac = E + (ac != '' ? sep : '') + ac;     // insert element to vector
            }
            di('  ac',ac);                     // result in accumulator
            ac = '[ '+ac+' ]';
            return 1;
        },
        isGroup = ( t, p ) => {
            let rc;
            if ( t == '(' ) rc = '({'.indexOf(p) >= 0;
            else            rc = ')}'.indexOf(p) >= 0;
            // else if ( t == ')' ) rc = ')}'.indexOf(p) >= 0;
            // if (d) trn('  isGroup',{t:t,p:p,rc:rc});
            return rc;
        };

        // exTree
        di('exTree es',es);
        level++;
        if ( ! ( es instanceof Array ) ) return es;             // singleton
        while (es.length > 0) {

            // process R
            R = es.pop() ?? null;                               di('R',jstr(R));
            if ( collectDataVector(R) ) R = ac;                 // data vector
            else if ( isGroup(')',R) ) R = exTree(es);          // eval R subgroup
            else if ( R == '{' ) { R = null; break; }           // closing
            else if ( R.indexOf('/*') == 0 ) continue;          // /* */ comment, start again

            // get op and modifier
            op = es.pop() ?? null; mod = null;
            if ( op instanceof Array ) {
                if (d) trn('op',op); throw 'expr error: op is array'; }
            if ( op != null ) {
                if (!isNaN(op)) throw `expr error: op is a number: ${op}, expected primitive or function

mcode reminders:
    use triplets as in  α fn ω  for most operations, eg.  1 ÷ 3
    group triplets with ( ) if needed, eg.  avg ← { ( 0 + ⌿ ω ) ÷ ρ ω }
    use spaces around most elements, eg.  avg [ 1 2 3 ]
`;
                if ( (!isIdentifier(op)) && /\./.test(op) )     // not name and has .
                    [op,mod] = splitOpMod(op);
            }
            di('op mod',{op:op,mod:mod});

            // process op
            if (op == '}') op = exTree(es);                     // eval op lambda
            else if (op == '{') {                               // put fn on stack
                R = [null,'←',null,R]; break; }
            else if (op == '(') { R = [ R ]; break; }           // end subexpr
            else if (op == '}') break;                          // start lambda fn
            else if (op == ')')                                 // derived fn
                throw new Error('unexpected )');                // (function of functions) NYI
            else if (op==null)  break;                          // no op, only R present

            // process L
            L = es.pop() ?? null;                               di('L',jstr(L));
            if ( collectDataVector(L) ) L = ac;                 // data vector
            else if ( isGroup('(',L) ) {                        // end of group
                es.push(L); L = null; }
            else if (L== ')') L = exTree(es);                   // start of group
            // else if (L=='}' || isOperator(op)) {             // L is lambda or op is fn-opr
            else if (isOperator(op)) {                          // L is fn-operator
                // operators are functions that operate on other functions  see operatorRe
                // input: L0 leftFn operator R
                // stack: [ L, operator, [ modifier, leftFn ], R ]
                if (L=='}') L = exTree(es);                     // eval Left Lambda
                else if (isData(L)) throw 'left of operator cannot be data, got '+L+' in unit '+unit;
                mod = [ mod, L ]; L = null;
                // check next L for data
                let s = peek(es);                               di('opr',{mod:mod,s:s});
                if (!isFn(s) && !isGroup('(',s)) {              // s is not a fn and not grouping
                    let L0 = es.pop() ?? null;                  di('opr L0',jstr(L0));
                    if ( collectDataVector(L0) ) L0 = ac;       // data vector
                    L = L0;                                     // setup data for signature
                }
            }
            else if (L=='}') {                                  // L is lambda
                L = exTree(es);                                 di('lambda',jstr(L));
            }
            else if ( isFn(L) ) {                               // op or prim
                di('push back','');
                es.push(L); L = null; }                         // push back and continue

            // assemble group
            V = [L,op,mod,R];                                   di('V',V);
            es.push(V);                                         di('es',es);
            if ( t++ > 128 ) throw new Error('error: exTree overflow');
        }
        level--;
        di('result R',jstr(R));     if (d && level==1) log('\n');
        return R;
    },
    // emit code using recursive depth first search of exTree result
    // tree nodes are  L op R
    emit = (es) => {       // nb. is recursive
        let
        L = null,               // left term
        R = null,               // right term
        op = null,              // operation
        mod = null,             // modifier
        pn = null,              // primitive function name
        pre = '',               // function prefix  eg. await
        r = '',                 // result string
        cvtDword = a => {
            let s = a;
            s = s.replace(/([^_])(_)(?!_)/g,'$1 '); // consume last X_ as Xb  matches solo _
            // di('cvtDword',[a,s]);
            return s;
        },
        matrixIndexing = s => {
            // di('matrixIndexing 0',s);
            // matrix notation replacements: M[x,y] to M[x][y]
            if ( /^\(|^\[/.test(s) )        return s;       // expr or literal
            if ( /\[\]|=>/.test(s) )        return s;       // empty ar or fn
            if ( !/^\S+\[\S+\,/.test(s) )   return s;       // not in proper form
            // re with capture groups
            // s = s.replaceAll(/((\s|[^,]+)\[)([^,])(,)([^,]\])/g,'$1$3][$5');
            s = s.replaceAll(/((\s|[^,]+)\[)([^,]{1,3})(,)([^,]{1,3}\])/g,'$1$3][$5');
            // 1 to 3 symbol indices only
            di('matrixIndexing',s);
            return s;
        },
        Vsub = a => {                                           // variable substitutions
            if (typeof a != 'string') return a;                 // di('Vsub',a);
            if ( '\'"`'.indexOf(a[0]) >= 0 ) return a;          // should not process string literal contents
            let s = cvtDword(a);
            s = matrixIndexing(s);
            for (const b of mc.vmap) s = s.replaceAll(...b);    di('Vsub',[a,s]);
            return s;
        },
        Bsub = a => {                                                   // built-in substitutions
            if ( typeof a != 'string' ) return a;
            let s = mc.bmap.get(a);
            if ( s == '_' || s == undefined ) s = a;                    // no change
            s = cvtDword(s);                                            di('Bsub',[a,s]);
            return s;
        },
        getFn = (L,op,mod) => {
            let pn = null, vf = null, bf = null, f = null;
            di('getFn',{L:L,op:op,mod:mod});
            pn = mc.pmap.get(op) ?? null;                       // primitive lookup
            if (!pn) vf = mc.fmap.get(op) ?? null;              // vector fn lookup
            if (!vf) bf = mc.bmap.get(op) ?? null;              // built-in lookup
            di('  getFn  ',{pn:pn,vf:vf,bf:bf});
            if (pn) f = pn;                                     // call prim
            else if (vf) f = L!=null ? vf.d : vf.m;             // call vector function
            else if (bf) f = '(a,w)=>a'+(bf=='_'?op:bf)+'w';    // call built-in
            if (mod) mod = Vsub( quoteArg(mod) );
            if (f) f = '['+f+','+mod+']';
            di('  getFn f',f);
            return f;
        },
        getVecExpr = (L,op,mod,R) => {
            // get operator expr or vectorized expr if any
            di('getVecExpr',{pn:pn,L:L,op:op,R:R});
            di('  mod',mod);
            let r = '', f = null;
            if (pn) {
                if (isOperator(op)) {
                    // calls to operators:
                    // op(α,ω,[op_mod,f0])  where fN = [opN,modN]  op_mod is operator's mod
                    let op_mod = null;
                    if (mod) {
                        if (mod[1][1]=='←') {                   // lambda  nb. lambdas have no mod
                            f = '['+emit(mod[1])+',null]';      // emit lambda
                        } else {
                            let op0 = null, mod0 = null;
                            [op0,mod0] = splitOpMod(mod[1]);    di('opr',{op0:op0,mod0:mod0});
                            f = getFn(L,op0,mod0);              // lookup vec fn for operator
                        }
                        op_mod = Vsub( quoteArg(mod[0]) );      // prep operator's modifier
                        if (!f) f = '['+Vsub(mod[1])+',null]';  // fn is named fn
                    }
                    r = pn+'( '+L+', '+R+', ['+op_mod+','+f+'] )'; }    // non-operator primitive
                return r; }                                      // return '' if prim but not operator
            f = getFn(L,op,mod);                                 // lookup vectorized fn
            if (f) {
                // calls to vectorized function handler:
                // op(α,ω,[f0,f1])  where fN = [opN,modN]
                let f1 = getFn(L,mod,null);                      // setup for inner product  eg. +.×
                r = 'mcode.vf( '+L+','+R+',['+f+','+f1+'] )'; } // called as vf(α,ω,δ) where δ is fn array
            di('  getVecExpr r',r);
            return r;
        },
        OPsub = R0 => {                                          // operation substitutions
            // returns string r, but may also set 'pre' to 'await' for async ops
            let r = '';
            if (typeof op != 'string') return '';                // not expected
            for (const b of mc.vmap) op = op.replaceAll(...b);  // vmap subs in op  ∆ ⊙
            di('OPsub',{L:L,op:op,mod:mod,R:R});

            if ( op == '□' && (mod == null || mod == 'j') ) {                    // log idiom
                let Rq = '\''+R0+'\'', f = mc.pmap.get('□'), mq = mod ? "'"+mod+"'":mod;
                // if no L arg and 1 char symbol x or ⊙.x then construct 'R' □ R
                if ( L == null && /^\w+$|^(_cp\.)?.$/.test(R) ) r = f+'('+Rq+','+R+','+mq+')';
                else r = f+'('+L+','+R+','+mq+')';
                return r;
            }

            if ( op == '←' ) {                                   // create lambda fn
                if ( L !== null ) r = L + ' = (_a,_w,_d) => ' + R; // named func expr, nb. L sb declared already
                else r = '(_a,_w,_d) => ' + R; }                 // anonymous func

            else if ( op == '⍙' ) {                              // ⍙ create idioms
                if ( mod=='n' )                                  // ⍙.n  new class
                    r = 'new '+R+(L!=null?('(...'+L+')'):'()');      // spread for ctor call
                else if ( mod=='v' )                             // ⍙.v  let varlist
                    r = 'let '+R0;                               // use raw R in R0
                else if ( /^[^'"`]/.test(R) ) {                  // ⍙ quote R for ⍙.'[[]]' etc.
                    mod = quoteArg(mod);
                    r = pn+'('+L+',\''+R+'\','+mod+')'; } }

            else if ( op == '⍷' && L!=null && /^[^'"`]/.test(R0) ) // for ⍷ quote R when L not null (nb. or ⍷ ω )
                r = pn+'('+L+',\''+R0+'\',null)';
```

```
            else if ( op == 'α' )                               // call α
                r = '_a('+L+','+R+')';
            else if ( op == 'ω' )                               // call ω
                r = '_w('+L+','+R+')';
            else if ( /^⊙/.test(op) )                           // call ⊙.fn
                r = op.replace('⊙','_cp')+'('+L+','+R+')';
            else if ( /^⍃|^⍄/.test(op) && L==null)              // add await in async fn
                pre = 'await ';
            else if ( op == ',' )                               // pass comma
                r = L+', '+R;
            else if ( op == '⍕' && mod != null )                // special conversions, others handled by vf, see
fmapIn
                r = 'mcode.format0('+L+','+R+','+'\''+mod+'\')';

            di('OPsub r',jstr(r));
            return r;
        },
        wrapAsg     = (op,r) => /=|‗$/.test(op) ? r : '('+r+')',
        wrapLambda  = (op,r) => /=>/.test(op) ? '('+r+')' : r,
        emitFnCall = (R0) => {
            pn = mc.pmap.get(op) ?? null;
            r = OPsub(R0);                                      // returns '' if no idiom  also sets 'pre'
            if ( r == '' && isBuiltin(op,mod) ) {               // infix builtin with no mod
                op = Bsub(op);
                r = (L?L+' ':'')+op+' '+R; }                    // build L op R
            if ( r == '' ) r = getVecExpr(L,op,mod,R);          // general vector fn expression
            if ( r == '' ) {
                if ( pn == null ) {                             // not a known prim
                    if ( (!/‗/.test(op)) &&                     // not a keyword
                    (!/=>/.test(op)) && !isIdentifier(op)) {    // not a lambda and not identifier
                        log('warning: primitive '+op+' is unknown in '+unit+'\n');
                        pn = 'mcode.nyi';
                        mod = '\''+op+'\'';                     // call nyi with mod = unknown op
                    }
                    else pn = op;                               // named function call
                }
                pn = wrapLambda(op,pn);                         // for lambda
                mod = quoteArg(mod);
                r = pre+pn+'('+L+','+R+','+mod+')';             // build call with any quoted args
            }
            let isOpen = /\w\($/.test(op);                      // op is direct call  ( was given
            di('isOpen',isOpen);
            r = r + (isOpen ? ' )':'');                             // add closing )
            di('emitFnCall',r);
            return r;
        },
        emitWord = es => {
            pn = mc.pmap.get(es) ?? null;
            if ( es == '⍠' ) return 'debugger'
            di('emitWord',[es,pn]);
            if (pn) throw new Error('missing right data for primitive '+es);
            return Vsub(es);
        };

        // emit
        di('emit es',jstr(es));
        if ( ! ( es instanceof Array ) ) return emitWord(es);
        level++;

        while (es.length > 0) {

            // gather and process a group of [ L op mod R ]

            let isRleaf = 1, isLleaf = 1;                           // track R L node type

            // gather R
            R = es.pop() || null;                                   di('R',jstr(R));
            if ( R instanceof Array ) {                             // recurse for R expression
                R = emit(R); isRleaf = 0; }

            // prepare op and mod
            mod = es.pop() ?? null;
            op = es.pop() ?? null;
            di('op mod',[op,mod]);
            if ( op instanceof Array ) {
                op = emit(op); di('r op',op); }                     // recurse to build lambda  op is { ... }

            // gather L
            L = es.pop() || null;                                   di('L',jstr(L));
            if ( L instanceof Array ) {                             // recurse for L expression
                L = emit(L); isLleaf = 0; }
            // checks
            if ( R == ',' ) R = null;                           // no R data
            if ( L == ',' ) L = null;                           di('L op R',[L,op,R]);
            if ( /^\/\*/.test(R) ) { di('cmt',R); return op; }  // inline comment

            let R0 = R;                                         // save raw R

            // () wrapping
            if ( !isRleaf ) R = wrapAsg(op,R);                  di('wrapAsg R',R);

            // substitutions
            di('leafs',{isLleaf:isLleaf,isRleaf:isRleaf});
            if (isLleaf) L = Vsub(L);                           // only do Vsubs on leaf nodes
            if (isRleaf) R = Vsub(R);
            mod = Vsub(mod);

            // if (op=='/.') trn('','XXX'); op = '/';           // divide idiom  zvzv

            if (op != null) r = emitFnCall(R0);                 // fn call
            else r = !isRleaf ? Vsub(R) : R;                    // expr
        }
        level--;                                                // done with expression
        di('result r',r);
        return r;
    };
    mc.expr = (s, u) => {                                       // parse an expression
        if ( s == '' ) return '';
        unit = u; reset();
        return emit( exTree( mc.lex( s, 'expr', unit )));       // nb. may throw an error
    };
}; expr();

// js code check
let codeCheck = () => {
    let d0 = 0,             // full debug info
        reminders = `
mcode reminders:
    use a space between most elements, eg. avg ← { ( 0 + ⌿ ω ) ÷ ρ ω }
    use no spaces for JSON or JavaScript

`,
        jshintsReporter = (code,d=0) => {
        // returns 0 if no errors, 1 if errors found
        // URL: https://cdnjs.cloudflare.com/ajax/libs/jshint/2.13.6/jshint.min.js
        let jsh = JSHINT,   // nb. loaded by page
        // see: https://github.com/jshint/jshint/blob/2.1.4/src/shared/messages.js
        suppress = 'W027 W030 W032 W040 W051 W093 W098 W087 W117 W118 E006 E021 E041 E054 E058'.split(' '),
        // E030
        // see https://jshint.com/docs/options/
        options = { esversion: 11, strict:'implied', nocomma: true,
            undef: true, unused:true, asi:true, latedef:true, // shadow:true,
            browser:true, devel:true, nonstandard:true,
            predef:['mcode','fetch','_cp','_'],
            };
            // nb. asi: true  means no semicolon check
        if (!jsh) { log('warning: cannot access jshint\n'); return; }
        let cf, codeFix = (c) => {
            // needed due to a bug in jshints
            // parses class method expressions incorrectly in version 2.13.6
            // eg.
            // class Foo {
            //      m1 = (_a,_w,_d) => {     // is valid, but jshint reports errors
            let d = '';
            for (let b of c.split('\n')) {
                if (b.indexOf(mc.jshintFixMarker) > 0)
                    b = b.replace(/(\S+) = \(.*\) =>/,'$1(_a,_w,_d)');
                d += b + '\n';
            }
            return d;
        };
        cf = codeFix(code);
        jsh(cf,options);
        let rpt = jsh.errors, line = 1, cs, fe=0;
        if (d0) trn('rpt',jstr(rpt));
        let
        showError = (line,rpt) => {
            let rc = 0;
            for (let b of rpt) {
                if (b.line == line) {
                    log('     error: '+b.reason+' '+b.code+'\n');
                    if (!d && !d0) { log('     stopping report\n'); rc=1; break; }
                }
```

```
            }
            return rc;
        },
        removeWarnings = rpt => {
            let r = [], v = '_ _a _w _d'.split(' ');
            for (let b of rpt) {
                if (d0) trn('b',b);
                if ( ! suppress.includes(b.code) ) {
                    r.push(b); if (!fe) fe=b.line; } }
            return r;
        };
        if (!d0) rpt = removeWarnings(rpt);
        if (rpt.length == 0) {
            if (d0) log('no errors or warnings from jshints\n');
            return 0;
        }
        log('problem found:\n');
        cs = code.split('\n');
        // if (cs.length>30)
        for (const m of cs) {
            if (line>fe-10) {
                log(line.toString().padStart(4,' ')+' '+m+'\n');
                if (showError(line,rpt)) break;
            }
            line++;
        }
        // log(reminders);
        return 1;
    };
    mc.codeCheck = jshintsReporter;
}; codeCheck();

// parser & code block builder: non-recursive statement processor
let parser = () => {
    let r, k, L, sop, R, S, sm, ln, il, ila, lna, isa, st, sop2, src, unit,
        sc, d,
    top = 'top {}',
    dbg = (p = '') => {     // debug info reporter
        let b1 = { il:il, ln:ln };
        tr('',b1);
        // log( p + tr('', b0, tr('',r,'') ));
        // tr( ' r', jstr(r) ); nl();
    },
    reset = () => {
        r = [];                 // result
        // d                    // debug flag
        // k                    // index for lookahead of stmt token
        // L                    // left arg
        sop = '';               // statement operation
        // R                    // right arg
        // S                    // next statement
        // sm                   // stmt template from statement map
        ln = 0;                 // line nr
        il = 0;                 // indent level
        // st                   // statements from lexer
        ila = [];               // indent level array
        lna = [];               // line number array
        src = [];               // mcode source before lexing
        isa = {};               // initialization statement array
        sop2 = ''               // previous stmt operation; used for : after ∇ etc
        unit = top;             // code unit
        sc = 0;                 // statement count (for return heuristic)
    },
    idErr = (s, ln) => {
        return (!!unit ? ' in "'+unit+'"' : '') +
            (ln ? ' at line '+(ln+1) : '') + ':\n' + s + '\n';
    },
    rptError = ( err, s = '' ) => {
        let s0 = idErr(s, ln);
        // log(err+s0);
        throw new Error( err+s0 );
    },
    getSrc = ln => {
        let m = lna[ln]; // trn('getSrc ln',ln); trn('m',m);
        if (m==undefined) return '';
        return src[m]||'';
    },
    handleError = (e,module='') => {
        let g = {};                    // handle throw '' or throw new Error('')
        if ('string' == typeof e) g = { stack: e }; else g = e;
        if (module != '') module += ': ';
        let s = getSrc(ln);
        if (s!='') s = 'src: "'+s+'"\n';
        g.stack = s+module+g.stack;
        mc.tperr = 1;
        throw new Error(g.stack);
    },
    peek = (st) => st[0] ?? null,                  // peek at next item to be shifted from array
    collectMatrix = ( sop ) => {
        if (d) { trn('collectMatrix sop',jstr(sop)); }
        try {
            let es = mc.lex( sop, 'expr', unit ),
                r = '', t = '', e = '', ex = '', isMatrix = 0, es0 = es[0] ?? '';
            if (d) { trn('  es',jstr(es)); trn('  es0',jstr(es0)); }
            if (( es0 ).startsWith('[[')) {             // already in matrix form
                r = es[0];                              if (d) trn('  r',r);
                return r; }
            // if ( /^\[\S/.test(es0)) {                   // already in vector form  zvzv NIU
            //     r = es[0];                              if (d) trn('  r',r);
            //     return r; }
            while (es.length) {
                t = es.shift();                         if (d) trn('  t',jstr(t));
                if ( t == '[' ) r += t;
                else if ( /\/\/|#|⍝/.test(t) ) { es.shift(); continue; }    // 1 line comment
                else if ( /\/\*/.test(t) ) { continue; }                    // comment block
                else if ( t == ',' || t == ']' || t == '\n' ) {
                    if (d) tr('  e',jstr(e));
                    ex = mc.expr( e, unit );
                    if (d) { trn('  ex',ex); }
                    r += ex;
                    if ( t == '\n' ) { r += '],['; isMatrix = 1; sc++; }
                    else if ( t == ',' && peek(es) != '\n' ) r += ',';
                    else if ( t == ']' )  r += ']';
                    e = '';
                }
                else e += t + ' ';
            }
            if (isMatrix) r = '['+r+']';
            if (d) trn('  r',r);
            return r;
        } catch (e) { handleError(e,'mcode collectMatrix'); }
    },
    genCtor = ( R ) => {
        let r = '';
        for (let a of R.split(',')) { r += 'this.'+a+'='+a+'??0; '; }
        return r;
    },
    buildStmt = ( L, sop, R, sop2 ) => {
        // let d = 0;      // local debug
        let sm = mc.smap.get(sop), lexpr = '', rexpr = '', is_mcode = 1, r;
        sm = sm ? sm.s : '';
        if ( d ) { tr('  buildStmt  sop',sop); tr('L',L); tr('R',R);
            tr('sop2',sop2); trn('sm',sm); }
        if ( sop2 == '←' && sop.startsWith('[') ) return collectMatrix(sop);
        if ( sop == '⍔' ) {                             // return stmt with value check
            if ( R==null ) log('warning: return statement has no value in '+unit+'\n');
        } else if ( sop == '∇' && R ) {
            // is_mcode = 0;                                       // not mcode
            if ( R.startsWith('.a') ) {                         // make async func
                R = R.slice(2); sm = '_R = async (_a,_w,_d) =>'; }
            if (il > 0) sm = 'let '+sm;                         // fn inside fn
            unit = R;                                           // set unit to current function
            isa[ln] =  'mcode.fn = "'+unit+'"; '; // set runtime fn name for trace
        } else if ( sop == '⍢' && R ) {
            let mod = '';
            if ( R.startsWith('.') ) { mod = R[1]; R = R.slice(3); }
            if ( mod=='s' )                                     // simple ctor
                sm = 'constructor(' + R + ') { ' + genCtor(R) + ' }';
            else if ( mod=='c' ) sm = 'constructor('+R+')';     // general ctor
            else if ( mod=='t' ) sm = genCtor(R);               // ctor this
            else {                                              // method or class decl
                if (il > 0) sm = R + ' = (_a,_w,_d) => /* '+mc.jshintFixMarker+' */';
                else { sm = 'class '+R; if (L) sm += ' extends '+L; } }
        } else if ( sop == '⌺' ) {                              // ⌺ switch / case
            R = R || '';
            if ( R.startsWith('.s') ) { R = R.slice(2); sm = 'switch (_R)'; }
            if ( R.startsWith('.d') ) sm = 'default:';
        } else if ( sop == '⍂' ) {                              // pass thru
            is_mcode = 0;                                       // not mcode
            let s = R;
            if (s.indexOf('.err') == 0) s = '/* jshint ignore:start */';
            else if (s.indexOf('.nerr')==0) s = '/* jshint ignore:end */';
            for (const b of mc.vmap) s = s.replaceAll(...b);    // apply vmap subs
            rexpr = s;
        }

        if (is_mcode) {
            try {
                if ( sm == '' ) {
                    sm = sop; if (R) sm += ' ' + R;      if (d) trn('    is expr, sm',sm);
                    sm = mc.expr( sm, unit );
                    if ( sm == null ) { sm = sop;        if (d) trn('    not m expr, sm',sm); }
                } else {
                    if (d) log('    mcode\n');
                    lexpr = mc.expr( L, unit ),
                    rexpr = mc.expr( R, unit );
                }
            } catch (e) {
                handleError(e,'mcode expr');
            }
        }
        if (d) { trn('    lexpr',lexpr); trn('    rexpr',rexpr); trn('    sm',sm); }
        // : stmt syntax subs
        if ( sop == ':' ) {
            sm = sm.replace(': ',''); // nb. remove :    ∇ fn  ⌼ for  ⍢ class
            // insert statement for var initialization in JS
            if ( sop2 == '∇' ) { isa[ln] += 'let '+rexpr+';'; return ''; }
        }
        // line substitutions
        if ( typeof sm != 'string' ) sm = '';
        const rl = [ ['_L',lexpr], ['_R',rexpr], ['_S',''] ];
        for (const ri of rl) sm = sm.replace( ...ri );
        r = sm;
        if (r!='') { if (d) trn('    sc',sc); sc++; }
        if (sop=='⍔') r += ';';                     // append semicolor for return stmt
        if (d) trn('    r',jstr(r));
        return r;
    },
    specialCommands = s => {
        if ( s == '?\n' ) s = 'mcode.help()';       // help command
        s = s.replace(/<html>/,'// <html>');        // make comment for any file html line
        return s;
    },
    parsePgm = (sp) => {
        reset();
```

```
let rs, pgm = specialCommands(sp);
src = sp.split('\n');                       // code of mcode input
try {
    if (mc.parserDebug) logn('parsePgm: "'+pgm+'"');
    rs = mc.lex( pgm, 'stmts', unit );      // lexical analysis of statements
    if (mc.parserDebug) mc.lexDump(rs.r,rs.ila,rs.lna);
} catch (e) { handleError(e,'mcode lexer'); }
if ( !rs ) return null;                     // error
st = rs.r, ila = rs.ila, lna = rs.lna;      // results of lexer, st is eaten
let eol = () => {
    r.push('\n');
    ln++;
    if (d) trn('  eol r',jstr(r));
};
if (d) trn('src',jstr(src));

// parsePgm
while (st.length) {
    sop2 = sop;     sop = st.shift();   il = ila[ln];
    if (d) { tr('\nsop',jstr(sop)); tr('sop2',jstr(sop2)); tr('ln',ln); trn('il',il); }
    if ( sop == '\n' && sop2 == '\n' ) { eol(); continue; }
    if ( sop == '\n' ) { eol(); continue; }
    if ( sop == ';' ) continue;
    if ( sop == '//' ) { if (peek(st)!='\n') st.shift(); continue; }
    if ( sop.startsWith('/*') ) continue;

    L = null, R = null;
    k = mc.smap.get(sop) || 0;              if (d) trn('  sop k',k);
    if ( k ) {                              // mcode statement
        if ( k.lt ) L = sop2 || null;       // get L

        // get : R
        if ( k.rt || (sop==':' && sop2=='∇')) R = st.shift() || null;
        if ( sop == '∇̈' && peek(st)==':' ) { st.shift(); L = st.shift() || null; }

        if ( R == '\n' ) { R = null; st.unshift('\n'); }    // missing R arg
        let u = buildStmt( L, sop, R, sop2 );
        if (u!='') r.push(u);
    } else {                                // not a statement sop
        // soloExpr is false if the next sop will need this token
        k = mc.smap.get( peek(st) ) || 0;   if (d) trn('  peek k',k);
```

```
                let soloExpr = !(k && k.lt);        if (d) trn('  soloExpr',soloExpr);
                if ( soloExpr ) {
                    let Rt = peek(st);              // take next arg as possible R
                    if (d) trn('  Rt',jstr(Rt));
                    if ( Rt && ( Rt != '//' && Rt != '\n' && !k ) ) R = st.shift();
                    if (d) trn('  R',jstr(R));
                    if ( R == '\n' ) R = null;
                    if ( sop != '' ) {              // build solo stmt if given
                        if (sop.indexOf('⍢ ')==0) { // mark class method for jshint error fix
                            st.unshift(mc.jshintFixMarker); st.unshift('//'); }
                        let u = buildStmt( null, sop, R, sop2 );
                        if ( !/‗$/.test(sop) ) u += ';';      // semicolon inserted after solo expr
                        r.push(u); }
                }
            }
            if (d) { trn('r',jstr(r)); trn('  st',jstr(st)); }
        }
        // ila.push(0);                                // for lookahead     zvzv NIU
        return r;
    },
    buildBlocks = ( gc ) => {
        // construct output code blocks using input array 'gc' and indent array 'ila'
        let
            exprFn = 0,             // output 1 line expression function
            t,                      // current token
            ln = 0,                 // line
            bl = 0,                 // block level
            r = '',                 // result
            sf = mc.insertSource,   // source flag
            d = mc.parserDebug,     // local debug override
        dbg = () => {
            tr('  ln',ln);
            tr('il',[ila[ln],ila[ln+1]]);
            trn('r',jstr(r));
        },
        computeFirstIndent = ( st ) => {
            let i = 0, lnc = '', ln = 0, t;
            for (t of st) {
                if ( t != '\n' ) lnc += t;
                else  {
                    if ( lnc != '' ) { i = ila[ln]; break; }
                    lnc = ''; ln++;
                }
            }
            return i;
        },
        removeBlankLines = () => {
            let tp = '', ln = 0, gc0 = [], f = [], isa0 = {};
            for (t of gc) {
                if (t == '\n') {
                    f.push(ln);                     // store isa line
                    if (tp == '\n') { ila.splice(ln,1); lna.splice(ln,1); } // remove line
                    else { gc0.push('\n'); ln++; }
                } else gc0.push(t);
                tp = t;
            }
            gc = gc0;                               // store generated code
            for (t in isa) isa0[f[t]] = isa[t];     // re-index init stmt array
            isa = isa0;                             // store init stmt array
        },
        insertSrc = () => {
            // insert source mcode as comment
            if (!sf) return;
            let s = getSrc(ln);
            if (s=='') return;
                // src line nr NIU
                // lns = (ln+1).toFixed().padStart(4)
                // if (/^\s*\/\/\/ notebook/.test(s)) {
                //     r += '// X '+s.replaceAll('\\','Z');
                //     trn('s',jstr(s));
                // }
            if (/^\s*\/\//.test(s)) r += '    ' + s + '\n';          // cmt
            else if (/^\s*\/\*|\`/.test(s))                  // multiline
                r += '//  ' + s.split('\n')[0] + ' ...\n';
            else r += '//  '+s+'\n';                        // code
        },
        openBlock = () => {
            // start block if not top block
            let sb = 0+( ( ila[ln+1] > ila[ln] ) && (ila[ln] >= bil));
            if ( sb ) {                                 // start block
                // if ( !unit ) throw new Error('indent not allowed outside of function after:\n'+r);
                // r = r.replace(/(_?;\s*$)/,' ');         // remove any trailing ; before { // zvzv NIU
```

```
                r += '{ ';                             // trn('r',jstr(r)); // debug
                bl++;
            }
        },
        // zvzv NIU
        // closeBlock = () => {
        //     if ( ila[ln] < (ila[ln-1]||0) ) {          // check if end block needed
        //         if ( ila[ln] >= bil ) {                // if not top block
        //             while (bc-ila[ln] >= 1) {          // while bc > ila
        //                 let bcs = ' '.repeat(bc*4);    // indent spacing
        //                 r += bcs+'}\n'; bc--; }        // close block
        //         }
        //     }
        // },
        closeBlock = () => {
            // if ( ila[ln] <= bil ) return 0;             // if not top block  zvzv NIU
            let bc = ila[ln+1]||0, i=0;
            while (ila[ln]-bc >= 1 && i < 100) {           // while bc > ila
                r += '} '; bc++; bl--; i++; }              // close block
        },
        sol = (il=ila[ln]) => {
            // dbg();
            let p = il >= 0 ? ' '.repeat((1+il)*4) : '';    // regenerate indentation
            // if (d)  r += '/* '+ln+' '+ila[ln]+' '+sb+' '+bc+' */ '; // debug
            // if (d)  r += '/* '+ln+' '+ila[ln]+' */ '; // debug
            r += p;
        },
        removeTrLF = () => {                               // remove trailing \n s in generated code
            let i = 0; while (gc.slice(-1) == '\n' && i<100) { gc.splice(-1,1); i++; }
        },
        closeCodeBlocks = () => {                          // close any open code block levels
            let i = 0; while (bl && i<100) { r += '\n'; sol(bl); r += '}'; bl--; i++; }
        },
        bil = computeFirstIndent(gc);
        removeTrLF();
        exprFn = sc==1 && gc[0]!='\n' && gc[0]!='debugger;';
        if (d) { tr('\nbuildBlocks  bil',bil); tr('sc',sc); trn('exprFn',exprFn); }
        if (exprFn) gc.unshift('return');                  // add return value
        removeBlankLines();
        if (d) {
            // trn('  src',jstr(src));
            // trn('  gc',jstr(gc));
            // trn('  ila',ila); trn('  lna',lna);
            mc.lexDump(gc,ila,lna);
            trn('isa',jstr(isa)); }
        insertSrc();
        sol();
        for (t of gc) {
            if ( t != '\n' ) r += t + ' ';              // append token
            else {
                openBlock();
                closeBlock();
                if (isa[ln]) { r += isa[ln]; }          // insert init statements
                if ( !exprFn ) r += '\n';               // no \n for 1 line fns
                ln++;
                insertSrc();
                sol();
            }
        }
        closeCodeBlocks();
        r = r.replace(/\*\/;/g,'*/');       // remove any trailing comment semicolon
        // r = r.replace(/\};/,'}');        // remove any trailing block semicolon  zvzv NIU
        if (d) { trn('buildBlocks r',jstr(r)); logn(r) };
        return r;
    },
    chkBlockErrors = ( r, m ) => {      // zvzv NIU - doesn't account for embedded { } in ` `
        il = 0;
        il += (r.match(/\{/g) || []).length;
        il -= (r.match(/\}/g) || []).length;
        if ( il != 0 ) rptError( 'unbalanced { } blocks', 'input:\n'+m+'\noutput:\n'+r);
    };
    mc.parse = ( m ) => {
        let r = null;
        d = 1 && mc.parserDebug;
        mc.tperr = 0;   // reset
        r = parsePgm( m );
        if (r) r = buildBlocks( r );
        // if (r) chkBlockErrors( r,m );    // NIU
        return r;
    };
    let parserTests = () => {
        log('parser tests:\n');
        const tc = '';
        trn('tc',tc);
        let r = mc.parse(tc);
        trn('r',r.r);
    };
}; parser();

// mexec: parses then executes mcode source
let mexec = () => {
    let level = 0,              // exec level
        pgm = null,             // last program parsed
        cerr = 0;               // code check error
    mcode.cp = {};              // user data object accessed as ⊙
    mcode.mexecDebugLevel = 0;  // set from mexec options for this level
    mcode.lastUnit = '';        // last seen named code unit for IDE & debugging
    let
    pragmaStop = s => {         //  /// stop  stops all processing at that point
        let n = s.search(/^\/\/\/ stop/m);
        if ( n >= 0 ) { s = s.slice(0,n); log('parse stop encountered\n'); }
        return s;
    };
    mcode.mexec = (a,w,opts='') => {   // ⌽ execute mcode      errfn ⌽.'medr' 'mcodeString'
        let r = null, d, m;
        if (opts == '') opts = mcodeOptions.debug;
        opts = opts==''?'e':opts; opts = opts ?? 'e';   // default: e execute
        m = opts.indexOf('m') >= 0;                     // if 'm' then show mcode input
        d = opts.indexOf('d') >= 0;                     // if 'd' then debug on
        mcode.mexecDebugLevel = d;
        // if (d) trn('mexec',[opts,jstr(w.slice(0,50))]);

        if ( w === undefined ) { w = a; a = null; }     // setup for js calls with no errfn
        w = w ?? ''; w = pragmaStop( w.toString() );
        mc.lexerDebug   = d && opts.indexOf('L') >= 0;  // debug suboptions
        mc.exprDebug    = d && opts.indexOf('E') >= 0;
        mc.parserDebug  = d && opts.indexOf('P') >= 0;
        mc.insertSource = opts.indexOf('S') >= 0;
        try {
            level++;
            if (d) log('mexec level = '+level+' opts = '+opts+' input =\n'+w+'\n');
            else if (m) log('mexec: "'+w+'"\n');
            pgm = null;
            pgm = mc.parse(w);
            if (pgm) {
                cerr = mc.codeCheck(pgm,d);
                if (cerr) log('    attempting JavaScript evaluation and execution:\n');

                // let newFn = new Function( pgm );
                let newFn = new Function( '_cp', pgm );
                if (d) log('mexec newFn =\n'+newFn+'\n');
                // r = newFn();                            // do eval  nb. done in global scope
                r = newFn(mcode.cp);                 // do eval  nb. done in global scope

                if ( opts.indexOf('r') >= 0 ) r = pgm;      // if 'r' then return parse result
                else r = r ?? null; // ' at level '+level;
            }
            mc.lexerDebug = mc.exprDebug =  mc.parserDebug = mc.insertSource = 0;
            level--;
        } catch (e) {                                       // only called for synchronous fn errors
            mcode.msg('error');
            let err = e, lv;
            if ( a && a instanceof Function ) a(err,pgm);
            else {
                if (!mc.tperr) logn('error: at runtime, level '+level);
                logn(err);
                let n = 1000;
                if ( pgm ) {
                    log( '    during JavaScript evaluation of:\n'+pgm.slice(0,n)+'\n' );
                    if (pgm.length>n) log('...\n');
                    if (mcode.fn)       logn('    last run function was: '+mcode.fn);
                    if (mcode.lastUnit) logn('    last parsed function was: '+mcode.lastUnit);
                    // let c = mc.cmap.get('');
                    // if (c && c!='\n') log('mcode src:\n'+c);
                } else {
                    log('    during mexec of: "\n'+w.slice(0,n)+'"\n' );
                    if (w.length>n) log('...\n');
                }
                //log('\n');
                r = null;
                lv = level; level--; pgm = null;
                // log('error: occured at mexec level '+lv + '\n');
                throw 'stop';
            }
        }
        if (d) trn('mexec r',jstr(r));
        return r;
    };
    mcode.addPrim( '⍎', 'mcode.mexec','M' );                // execute mcode
    let mexecTests = () => {
        log('mexec test\n');
        // mcode.guide();
        const tc = '';
        trn('tc',tc);
        let r = mcode.mexec(null,tc,null);
        trn('r',r);
    };
}; mexec();

// core, futures, file loader, calls mexec
let coreLoader = () => {

    mcode.desktop = +(typeof Neutralino != 'undefined');

    // await mcode.busy  for async operations
    mcode.busy = null;                      // promise: resolves when mcode is done
    // nb. ok for caller to await with no promise
    mcode.done = () => 0;                   // resolver function for .busy
    mcode.setBusy = () => {                 // call in non-async fn before starting operation
        if (!mcode.busy) mcode.busy = new Promise(rs=>mcode.done=rs);
        return mcode.busy; };

    // set when mcode needs input, eg. notebooks or console input
    mcode.consoleInput = null;              // await mcode.consoleInput for input
    mcode.haveInput = () => 0;              // provider calls mcode.haveInput(data)
    mcode.getInput = () => {                // setup input before awaiting
        mcode_ide.editor.focus();
        if (!mcode.consoleInput) mcode.consoleInput = new Promise(rs=>mcode.haveInput=rs);
        return mcode.consoleInput;
    };

    // NIU
    // mcode.isBusy = f => { let r=mcode.busy; mcode.busy=null; return r ?? f; };
    // if busy promise then return it, otherwise return f
    // in async func: data = await_ mcode.isBusy();
    // example using timer with await mcode.isBusy in an async function :
    // async wait = () => { mcode.setBusy(); setTimeout(()=>mcode.done(-1),200); trn('timeout', await
mcode.isBusy()); }; wait();

    // defaults for ⍎ mexec from IDE, can be set in core
    mcode.shellOpts = '';

    // show a msg of a few words, usually 'error' or 'ready' in IDE status banner
    mcode.msg = (m) => { if (mcodeOptions.msg) mcodeOptions.msg(m); };

    // NIU
    // mcode.mexecAsync = (a,w,d) => mcode.isBusy( mcode.mexec(a, w, d) );
    // mcode.mexecAsync = (a,w,d) => {
        // mcode.setBusy();
        // let r = mcode.mexec(a, w, d);
        // if (mcode.busy) { log('not busy\n'); mcode.done(r); }
    //     return r;
    // };
    // usage: data = await mcode.mexecAsync(a,w,d)

    mcode.errorFn = e => mcode.log('async error: '+e+'\n');
    // default error handler for async errors

    window.addEventListener('unhandledrejection', function(event) {
            mcode.errorFn(event.reason);
            // alert('error: unhandledrejection:\n'+event.reason);
    });

    mcode.getURL = async (a,w) => {   // get URL as data by JavaScript
        // a is callback fn or null for promise return  w is URL or file
        // nb. data is not attached to document
        // usage:   callback:   mcode.getURL( () => 0, 'url' );
        //          async:      await mcode.getURL( null,'url' );
        let r = await fetch(w,{method:"POST",cache:"no-cache"});
        if (a instanceof Function) {
            if (!r.ok) a(r.status,'');
            else r.text().then( data => a(r.status,data) ); // used with callback fn
        } else {
            if (r.ok) return r.text();              // used with async/await or .then
        }
        // nb  .then style:
```

```
    // let p = fetch(w,{method:"POST",cache:"no-cache"}).then(rsp=> {
    //     if (!rsp.ok) a(rsp.status,'');
    //     else rsp.text().then( data => a && a(rsp.status,data) );
    // } );
    return null;
};

mcode.loadURL = (a,w) => {      // load URL by browser as a Document element
    // a is callback fn  w is URL
    // usage:   callback load:   mcode.loadURL( () => 0, 'url' );
    //          async load:      await mcode.loadURL( null,'url' );
    let head, s, p = null;
    if (/\.js/.test(w)) {
        head = document.getElementsByTagName('head')[0];
        s = document.createElement('script');
        s.src = w;    // nb. try ... NFN, e.stack is undef (and using fetch/eval)
        s.type = 'text/javascript';
    } else if (/\.css/.test(w)) {
        head = document.getElementsByTagName('head')[0];
        s = document.createElement('link');
        s.href = w;
        s.type = 'text/css';
        s.rel = 'stylesheet';
    }
    // else {
    //     throw 'unknown URI type';
    // }
    if (!s) throw new Error('loadURL failed');
    if (a instanceof Function) s.onload = a;    // callback fn
    else {
        p = new Promise( (resolve, reject) => {
        s.onload = () => resolve(s);            // resolve with script, not event
        s.onerror = reject;
        } );
    }
    head.append(s);
    return p;
    // nb  to process file list:
    // if ( uriList.length ) s.onload = () => loadURI( uriList, cb );
    // else if (cb) s.onload = cb;
```

```
        // s.onload = uriList.length ? () => loadURI( uriList, cb ) : cb;  // nb. recurse to load more if list
else do cb
        // if (s && a instanceof Function) s.onload = a;    // callback fn
    };

    mcode.serverAuth = null;        // authorization for server to write files

    let completion = async () => {

        let c,r;
        await mcode.loadURL( null,'https://cdnjs.cloudflare.com/ajax/libs/jshint/2.13.6/jshint.min.js' );

        try {
            if (!mcodeUseCache) {
                logn('reading source');
                c = await mcode.getURL( null, mcode.href+'core.mc.txt' );
                r = mcode.mexec(null, c, mcodeOptions.debug);
                trn('',r);
                if (mcodeDebug == '') {
                    c = await mcode.getURL( null, mcode.href+'rtl.mc.txt' );
                    r = mcode.mexec(null, c, mcodeOptions.debug);
                    trn('',r);
                } else logn('runtime library not loaded');
                await mcode.busy;           // await for any IDE promises
                // logn('core done');
            } else {
                await mcode.loadURL( null, mcode.href+'lib/mcode_cache.js' );
            }
            mcode.lastUnit = '';   // clear trace
            mcode.msg('ready');
            if (mcodeOptions.onload) mcodeOptions.onload();
        } catch(e) {
            mcode.logn('error: core '+e)
            mcode.msg('error');
        }
    };
    completion();

}; coreLoader();

// debugging - hoisted functions  nb. can be used before declaration
```

```
function log(msg) {
    if ( mcodeOptions.log ) mcodeOptions.log(msg);      // use IDE logger
    else {
        let el = document.getElementById('log');        // use current page log textarea
        // if ( msg == '' ) msg = '\n';
        if (el) el.value += msg;
    }
    return '';
}
function logn(m) { log(m); log('\n'); }
function tr( m, v ) {
    let ll = 50, mapId = 'M { ', precision = 5, level = 0, max = 1000,
    num = (v,p=precision) => {
        if ( isNaN(v) ) return '∞';
        if (typeof v != 'number' || Number.isInteger(v) ) return v;
        if (v=='Infinity') return '∞';
        return v.toFixed(p);
    },
    mao2 = v => {
        if (v instanceof Function) return v.toString();
        if (level>2) return jstr(v);
        return entry(v);
    },
    mao = (e,v,t) => {                           // map, array, object
        let k, b, c,
        isMap = e == mapId,
        pr2 = (k,b) => {
            if (e!='[') c += k;
            if (isMap) c += ' ';
            c += e=='{'?':':'';
            c += mao2(b) + ' ';
            if (isMap) c += ' ';
            if ( c.length > ll ) { t += c + '\n  '; c = ''; }
// logn('  pr2 c = '+c);
// logn('  pr2 t = '+t);
        };
        t = e + ' ';
        {
            c = '';
            if ( isMap ) for (const [k, b] of v ) pr2(k,b);
            else {
                // logn(jstr({v:v,k:Object.entries(v)}));
                if (Object.entries(v).length > max) return Array.isArray(v) ? '[...]' : '{...}';
                for (const [k, b] of Object.entries(v) ) pr2(k,b);
            }
            t += c + (e == '[' ? ']' : '}');
        }
// logn('  mao t = '+t);
        return t;
    },
    matrix = (m,v) => {     // return matrix rep or null
        if ( !(v instanceof Array) ) return 0;              // is matrix?
        if ( v.length==0 ) return 0;                        // is empty?
        let b;
        for (b of v) if (!(b instanceof Array)) return 0;   // each elem has array?
        let vf = v.flat();
        if (vf.length > max) return '[[...]]';
        let rc=1, w=1, f=0, r='[ ', p=4, i=0, c=v.length, t=/\S/.test(m), rl;
        for (b of vf) {
            if (typeof b !== 'number') { rc=0; break; }                     // all numeric?
            let w0 = (b+'').replace('.','').length; if (w0>w) w = w0;       // get max nr of digits
            if (!Number.isInteger(b)) f=1;      // is fractional
        }
        // logn(jstr({rc:rc,f:f,w:w,c:c}));
        if (!rc) return null;
        if (t) r = m + ' ← [ '; rl = r.length; w++;
        for (b of vf) {
            let p = rl > 0 ? ' '.repeat(rl) : ''
            if ((i%c) == 0 && i) r += '\n'+p; i++;
            // r += f ? ' '+b.toFixed(p) : (b+'').padStart(w);
            r += f ? ' '+num(b,p) : (b+'').padStart(w);
            // r += ' ('+i+') ';
            if (i<vf.length) r += ',';
        }
        // if (t)
        r += '  ]';
        return r;
    },
    entry = v => {                          // recursive from mao2
        level++;
        let t = '';
        if ( v === '' )                     t = '';
        else if ( typeof v == 'number' )    t = num(v);
        else if ( v === undefined )         t = 'U undefined';
        else if ( v === null )              t = null;
        else if ( v === window )            t = '{ window global }';
        else if ( v instanceof Promise )    { t = 'P ' }
        else if ( v instanceof Date )       { t = 'D ' + v.toLocaleString() + ' ' }
        // nb. JSON returns Date.toISOString()
        else if ( v instanceof Function )   { t = 'F ' + v.toString() + ' ' }
        else if ( v instanceof RegExp )     { t = 'R ' + v.toString() + ' ' }
        else if ( v instanceof Map )        t = mao( mapId,v,t ); // ' = M ' + jstr(v);
        else if ( v instanceof Array )      t = mao( '[',v,t );
        else if ( v instanceof Object )     t = mao( '{',v,t );
        else t = v;
        // { for (const [k, a] of Object.entries(v)) t += '\n  ' + k + ':' + a; }
        // else if ( !isNaN( parseFloat(v) ) )       { t = '' + v.toString(); }
        // else                   { t = '"' + v.toString() + '"'; }
        // else                                t = v.toString();
        level--;
        return t;
    }, r;

    r = matrix(m,v);
    if (!r) {
        r = entry(v);
        if ( /\S/.test(m) ) r = m+' = '+r;
        r += '  ';
    }

    console.log( r );
    log( r );
    return v;
}   // trace with data interpretation/formatting
function jstr(b) {
    try { return JSON.stringify(b);
    } catch(e) { log('JSON.stringify failed\n'); }
}
function nl() { log('\n') }
function trn(m,v) { tr(m,v); nl(); return v; }

// add to API for other modules
mcode.log = log, mcode.logn = logn, mcode.tr = tr;
mcode.jstr = jstr, mcode.nl = nl, mcode.trn = trn;

// log('transpiler loaded\n');

function trTest() {
    logn('trace test:');
    let M = new Map([['a',0],['b',1]]);
    trn('M',M);
    let U = [];
    trn('U',U);
    let V = [[0,1],[2,3]];
    tr('V',V);
    let W = [[0,1],2,3];
    trn('W',W);
    let N = 0 / 0;
    trn('N',N);
    logn('\ndone');
}
// trTest();

}  // mcode

// end
```