```
to class x y ()
to number x y :: nprint ()
to vector x y :: substr ()
to atom x y (CODE 29)
to string x y :: substr ()
to arec x y ()
to float x y :: fprint ()
to falseclass x y (isnew)
to isnew (CODE 5)
☞false←falseclass.
☞(TITLE USER DO SIZE CODE SELF AREC GLOB MESS RETN CLAS
 length eval or and mod chars error
 ☞.,/;:-[]?"s⇑#()↜)←={}*+⇒<> go goto turn next contents end □ ≤≟≠≥ min max)

'↑h75.↑f1.
DONT EDIT ABOVE HERE!--These classes and atoms mentioned early to guarantee addresses for mach
**ine code.
```

HEREWITH A SOMEWHAT WHIMSICAL ANNOTATED VERSION OF SYSDEFS. ANNOTATIONS ARE IN ITALIC
** IT IS HOPED THAT THIS WILL PROVIDE SOME ELUCIDATION OF THE CODE ESCAPES, OBSCURITIES W
**NO DOUBT PERSIST . THE ANNOTATIONS ARE INTENDED TO BE BUT DIMLY LIGHTED MARKERS ON T
** TO TRUE ILLUMINATION.

TO PRINT ALLDEFS -----

Get ALLDEFS into Bravo (with font ST8.AL, already FONT.0 for those who have run any of the STBRAVO*.CM command files) and translate onto another file (e.g. ALLDEFS.TR). Get ALLDEFS.EC from MAXC and type it. You must have all the .EP files listed therein. Then you may type

GEARS/D ALLDEFS/G ALLDEFS.TR

↑h90.↑f2.
BOOTSTRAPPING MAGIC.

```
↑h75.↑f1.
        to isnew (null instance⇒(☞instance←allocate permsize.
                instance[0]←class. ⇑true)
        ⇑false).


↑h60.↑f0.'

to print (↜..)
        '↑h75.↑f1.
        :x.Print its address in octal.
        Printing goes to the same place as CODE 20. This is used primarily
        for bootstrapping. All system classes will print themselves.
        ↑h60.↑f0.'

to read (CODE 2)
        '↑h75;H;Head keyboard input into a vector. This is almost identical
        in function to the SMALLTALK read routine, except that DOIT is
        signalled by <CR> at zero-th parenthesis level, and single-quote
        strings are ignored. It is only available in Nova versions.
        ↑h60.↑f0.'
```

```
'↑h90.↑f2.
MESSAGE HANDLING
↑h60.↑f0.'

to : (CODE 18)
        '↑h75.↑f1.
        to : name

        ◁☞ ⇒(:☞name nil ⇒(⇑name←caller message quotefetch)
                                (⇑caller message quotefetch)

        Fetch the next thing in the message stream unevaluated
        and bind it to the name if one is there.

        ◁# ⇒(:☞name nil ⇒(⇑name←caller message referencefetch)
                                (⇑caller message referencefetch)

        Fetch the reference to next thing in the message stream
        and bind it to the name if one is there.

                (:☞name nil ⇒(⇑name← caller message evalfetch)
                        ⇑caller message evalfetch)

        Fetch the next thing in the message stream evaluated
        and bind it to the name if one is there.
        ↑h60.↑f0.'

to ◁ (CODE 17)
        '↑h75.↑f1.
        :☞token. token=caller.message.code[caller.message.pc]→
                (caller.message.pc←caller.message.pc+1. ⇑true) ⇑false.
        That is, if a match for the token is found in the message, then
        gobble it up and return true, else return false.
        ↑h60.↑f0.'

to ⍰ ( CODE 39 )

to ⍇ ( CODE 36 )

to ⇑ (CODE 13)
        '↑h75.↑f1.
        :x. then do a return, and apply x to any further message. Note
        that in (... ⇑x+3. ☞y←y-2), the assignment to y will never
        happen, since ⇑ causes a return.
        ↑h60.↑f0.'

to ☞ (CODE 9)
        '↑h75.↑f1.
        ⇑:☞. That is, get the next thing in the message stream unevalled
        and active return it (which causes it to be applied to the message).
        ↑h60.↑f0.'

to # (:#)
        '↑h75.↑f1.
        Returns a REFERENCE to its argument's binding.
        ↑h60.↑f0.'


'↑h90.↑f2.
CONTROL CLASSES
↑h60.↑f0.'

to repeat token (:#token. CODE 1)
        '↑h75.↑f1.
        repeat (token eval) Not a true apply to eval, and therefore token
        MUST be a vector.
        ↑h60.↑f0.'

to done x (◁with⇒(:x. CODE 25) CODE 25)
```

```
        '↑h75.↑f1.
        done causes a pop out of the nearest enclosing repeat, for, or do.
        "done with val" will cause the repeat to have value val
        ↑h60.↑f0.'

to again (CODE 6)
        '↑h75.↑f1.
        repeat (☞active←active caller. eq active. class #repeat⇒(done)).
        That is, redo the most recent repeat, for, or do loop.
        ↑h60.↑f0.'

to if exp (:exp⇒(☜then⇒(:exp. ☜else⇒(⍟. exp)exp)error ☞(no then))
                ☜then⇒(⍟.  ☜else⇒(:exp) false)error ☞(no then))
        '↑h75.↑f1.
        The ALGOL "if ... then ... else ..."
        ↑h60.↑f0.'

to for token step stop var start exp (
        ☞var ← ⍟.
        ☞start ← (☜←⇒(:) 1).
        ☞stop ← (☜to⇒(:) start).
        ☞step ← (☜by⇒(:) 1).
        ☜do. :#exp. CODE 24)
        '↑h75.↑f1.
        An Algol-like "for".
        Note the default values if "←","to","by",etc., are omitted.
        CODE 24 means  --repeat(exp eval).      This implies "done" and "again"
        will work, which is correct.
        ↑h60.↑f0.'

to do token step stop var start exp (
        ☞step←☞start←1. :stop. :#exp. CODE 24)


'↑h90.↑f2.
INITIALIZING SYSTEM CLASSES


        ↑h75.↑f1.
        Here are the main kludges which remain from the time when we
        really didn't understand classes very well, but wanted a working
        SMALLTALK. PUT and GET are two of the principle actions of class
        class. The new verson of SMALLTALK will have class as a class
        with these actions intensional.
        ↑h60.↑f0.'

to PUT x y z (:#x. :y. :z. CODE 12)
        '↑h75.↑f1.
        The first argument MUST be an atom which is bound to a class
        table.  The third argument is installed in the value side of that table
        corresponding to the name (atom) which was the second argument.
        ↑h60.↑f0.'

to GET x y (:#x. :y. CODE 28)
        '↑h75.↑f1.
        If "x" is a class table then the binding of the atom in "y" will be
        fetched.
        ↑h60.↑f0.'

to leech field bits : ptr (CODE 27
        isnew⇒(:ptr))
        '↑h75.↑f1.
        Lets you subscript any instance
        a[0] gives you the class, a[1] gives the first field, etc.
        a[2] gives you the pointer; a[2][] returns the BITS in an integer
        a[2]←foo will dereference count previous contents, but a[2][←]foo
        will not.
        ↑h60.↑f0.'
```

```
PUT USER ☞TITLE ☞USER
PUT falseclass ☞TITLE ☞false

PUT atom ☞DO ☞(CODE 29
        '↑h75.↑f1.
        ◀←⇒(:x.  ⇑x -- Lookup SELF and replace its value by x.)
        ◀eval⇒(⇑ -- Lookup the binding of SELF)
        ◀=⇒(⇑SELF=:)
        ◀chars⇒(⇑ -- printname of SELF (a string))
        ↑h60.↑f0.'

        ◀is⇒(ISIT eval)
        ◀print⇒(disp←SELF chars) )
        '↑h75.↑f1.
        Done this way (PUT used rather than using "to") because we
        wanted to know where the system classes are. Hence the initial
        "to atom x y ()" , for example, in "Bootstrapping Magic" followed
        by the behavior here.
        ↑h60.↑f0.'

to ev (repeat (cr read eval print))

PUT falseclass ☞DO ☞(CODE 11
        '↑h75.↑f1.
        ◀⇒⇒ (:☞.)
        ◀or⇒ (⇑:)
        ◀and⇒ (:.)
        ◀<⇒ (:.)
        ◀=⇒ (:.)
        ◀>⇒ (:.)
        ↑h60.↑f0.'

        ◀◀is⇒(◀false⇒(⇑true) ◀?⇒ (⇑☞false) ⇕. ⇑SELF)
        ◀print⇒(☞false print) )

PUT vector ☞DO ☞(CODE 3 ⇒(⇑substr SELF x GLOB MESS)
        '↑h75.↑f1.
        isnew⇒(Allocate vector of length :.
                Fill vector with nils.)
        ◀[⇒(:x. ◀].
                (◀←⇒(:y. ⇑y -- store y into xth element. )
                ⇑ xth element) )
        ◀length⇒(⇑ length of string or vector)
        ◀eval⇒(☞pc←0. repeat
                (null SELF[☞pc←pc+1]⇒(done)
                ☞val←SELF[pc] eval)
                ⇑val) sort of...
        ↑h60.↑f0.'
        ◀is⇒(ISIT eval)
        ◀+⇒(:y is vector⇒(☞x←SELF[1 to SELF length+y length].
                ⇑x[SELF length+1 to x length]←y[1 to y length])
                error ☞(vector not found))
        ◀map⇒(:y. for x to SELF length
                (evapply SELF[x] to y))
        ◀print⇒(disp←40. for x to SELF length
                (disp←32. SELF[x] print). disp←41)
        )

PUT string ☞DO ☞(CODE 3 ⇒(⇑substr SELF x GLOB MESS)

        '↑h75.↑f1.
        isnew⇒(Allocate string of length :.
                Fill string with 0377s.)
        ◀[⇒(:x. ◀].
                (◀←⇒(:y. ⇑y -- store y into xth element. )
                ⇑ xth element) )
        ◀length⇒(⇑ length of string or vector)↑h60.↑f0.
```

```
        ◁is⇒(ISIT eval)
        ◁=⇒(:y is string⇒(SELF length=y length⇒(
                for x to SELF length (SELF[x]=y[x]⇒() ⇑false)) ⇑false)
                ⇑false)
        ◁+⇒(:y is string⇒(☞x←SELF[1 to SELF length+y length].
                ⇑x[SELF length+1 to x length]←y[1 to y length])
                error ☞(string not found))
        ◁print⇒(0 = ☞x ← SELF[1 to 9999] find first 39⇒
                (disp ← 39. disp ← SELF. disp ← 39)
                SELF[1 to x-1] print. SELF[x+1 to SELF length] print)
        )

PUT number ☞DO ☞(CODE 4
        '↑h75.↑f1.
        ◁+⇒(⇑val+:)
        ◁-⇒(⇑val-:)
        ◁*⇒(⇑val*:)
        ◁/⇒(⇑val/:)
        ◁<⇒(⇑val<:)
        ◁=⇒(⇑val=:)
        ◁>⇒(⇑val>:)
        ◁≤⇒(⇑val≤:)
        ◁≠⇒(⇑val≠:)
        ◁≥⇒(⇑val≥:)
        ◁max⇒(☞x←:. SELF>x⇒(⇑SELF) ⇑x)
        ◁min⇒(☞x←:. SELF<x⇒(⇑SELF) ⇑x)
        ◁□⇒(◁+⇒(⇑val OR :)
                ◁-⇒(⇑val XOR :)
                ◁*⇒(⇑val AND :)
                ◁/⇒(⇑val LSHIFT :))
        ↑h60.↑f0.'

        ◁is⇒(ISIT eval)
        ◁print⇒(SELF>0⇒(nprint SELF)
                SELF=0⇒(disp←060)
                SELF=0100000⇒(disp←base8 SELF)
                disp←025. nprint 0-SELF) )
        '↑h75.↑f1.
        For floating point stuff see FLOAT
        ↑h60.↑f0.'

to - x (:x*¯1)
        '↑h75.↑f1.
        An often used abbreviation, has to work for float as well.
        ↑h60.↑f0.'

to base8 i x s (:x. ☞s←string 7. for i to 7
        (s[8-i] ← 060 + x ⍰7. ☞x ← x ⍁¯3). ⇑s)
        '↑h75.↑f1.
        Returns a string containing the octal representation (unsigned) of its
        integer argument.
        ↑h60.↑f0.'


☞ISIT ← ☞(◁?⇒(⇑TITLE) ⇑TITLE=⍟).

to nil x (#x)
        '↑h75.↑f1.
        nil is an "unbound pointer", which is used to fill vectors and
        tables.
        ↑h60.↑f0.'

to null x (:x. 1 CODE 37)
        '↑h75.↑f1.
        Null returns true if its message is "nil", otherwise false.
        ↑h60.↑f0.'

to eq x (CODE 15)
```

```
to disp x i (
        ◁←⇒(:x is string⇒(for i to x length (TTY←x[i])) TTY←x)
        ◁clear⇒() ◁sub⇒(:x eval))
        '↑h75.↑f1.
        This disp is used for bootstrapping. Later in these definitions
        (READER)it will be restored to an instance of "display frame."
        ↑h60.↑f0.'

to TTY (0 CODE 20)
        '↑h75.↑f1.
        TTY←<integer> will print an ascii  on the Nova tty. On altos, TTY
        prints in little error window at bottom of screen.
        ↑h60.↑f0.'

to dsoff (mem 272←0)
        '↑h75.↑f1.
        Turns display off by storing 0 in display control block ptr. Speeds
        up Alto Smalltalk by factor of 2.
        ↑h60.↑f0.'

to dson (mem 0420 ← 072)
        '↑h75.↑f1.
        Turns display back on by refreshing display control block
        pointer.
        ↑h60.↑f0.'

to apply x y  (:#x. ◁to⇒(:y. ◁in⇒(:GLOB. CODE 10) CODE 10)
                ◁in⇒(:GLOB. CODE 10) CODE 10)
to evapply x y (:x. ◁to⇒(:y. ◁in⇒(:GLOB. CODE 10) CODE 10)
                ◁in⇒(:GLOB. CODE 10) CODE 10)
        '↑h75.↑f1.
        Causes its argument to be applied to the message stream of the
        caller, or, in the case of apply foo to <vector>, to that vector.
        Note that only the message is changed, and that the caller is not
        bypassed in any global symbol lookup unless the in-clause is used to
        specify another context.
        ↑h60.↑f0.'

to cr (disp←13).  to sp (disp←32)

☞true←☞true
☞eval←☞eval
to is ( ◁?⇒(⇑☞untyped) ⁸. ⇑false)
        '↑h75.↑f1.
        These are used to handle messages to classes which can't answer
        questions invoking "is", "eval", etc.
        ↑h60.↑f0.'

to t nprint substr (ev). t
        '↑h75.↑f1.
        prevent -to- from making these global.
        ↑h60.↑f0.'

to nprint digit n (:n=0⇒()
        ☞digit←n mod 10. nprint n/10. disp←060+digit)
PUT number ☞nprint #nprint.
        '↑h75.↑f1.
        Prints (non-neg) integers in decimal     with leading zeroes suppressed!
        ↑h60.↑f0.'

to substr op byte s lb ub s2 lb2 ub2 (
        :#s. :lb. :ub. :MESS. ☞GLOB←ub.  '↑f1.tee hee↑h60.↑f0.
'
        :ub. (◁]⇒() error ☞(missing right bracket))
        ☞byte ← ☞lb2 ← ☞ub2 ← 1.
        ◁find⇒ (☞op ← (◁first⇒(1) ◁last⇒(2) 1)
                                + (◁non⇒(2) 0). :byte. CODE 40)
        ◁←⇒  (◁all⇒ (:byte. ☞op←0. CODE 40)
                :#s2. ☞op←5.
```

```
                ☜[⇒ (:lb2. ☜to. :ub2. ☜]. CODE 40)
                        ☞ub2←9999. CODE 40)
        ☞op ← 6. ☞ub2 ← ub+1-lb.
        ☞s2 ← (s is string⇒(string ub2) vector ub2). CODE 40).
PUT string ☞substr #substr.
PUT vector ☞substr #substr.
done
        '↑h75.↑f1.
        substr takes care of copying, moving and searching within strings
        and vectors. It first gets its father (string/vector) and the lower
        bound, and then proceeds to fetch the rest of the message from
        above. Some examples:
                ☞(a b c d e)[2 to 3] -> (b c)
                ☞(a b c d e)[1 to 5] find ☞c  -> 3
                ☞(a b c d e)[1 to 5] find ☞x  -> 0
        See vecmod for more examples. String syntax is identical.
        ↑h60.↑f0.'

to vecmod new end old posn ndel nins ins (☞end←10000.
        :old. :posn. :ndel. :ins.
        ☞nins←(ins is vector⇒(ins length-1) null ins⇒(0) 1).
        ☞new ← old[1 to old length+nins-ndel].
        (ins is vector⇒(new[posn to end] ← ins[1 to nins]) new[posn]←ins).
        new[posn+nins to end] ← old[posn+ndel to end].
        ⇑new)
        '↑h75.↑f1.
        Vecmod makes a copy of old vector with ndel elements deleted
        beginning at posn. If ins is a vector, its elements are inserted  at
        the same place. It is the heart of edit.
        ↑h60.↑f0.'

to addto func v w (:#func. :w. ☞v←GET func ☞DO. null v⇒(error ☞(no code))
        PUT func ☞DO vecmod v v length 0 w)
        '↑h75.↑f1.
        Addto appends code to a class definition.
        ↑h60.↑f0.'


to fill t i l str (
  ☞l ← :str length.
  ☞t ← disp ← kbd.
   (t = 10⇒
      (☞t ← disp ← kbd)).
  str[☞i ← 1] ← t.
  repeat
   (i = l⇒(done)
    10 = str[☞i ← i + 1] ← disp ← kbd⇒(done)).
  ⇑str)


to stream in : i s l(
  CODE 22
  '↑h75.↑f1.
  CODE 22 is equivalent to...
  ☜←⇒
   (
     (i = l⇒
       (☞s ← s[1 to ☞l ← 2 * l]))
    ⇑s[☞i ← i + 1] ← :)
  ☜next⇒
   (i = l⇒(⇑0)
    ⇑s[☞i ← i + 1])
  ☜contents⇒
   (⇑s[1 to i])
  ↑h60.↑f0.'
  ☜reset⇒
   (☞i ← 0)
  isnew⇒
   (☞s ←
```

```
      (◁of⇒(:)
       string 10).
      ☞i ←
      (◁from⇒((:)
         - 1)
       0).
      ☞l ←
      (◁to⇒(:)
       s length))
    ◁is⇒
     (ISIT eval)
    ◁end⇒
     (⇑i = l)
    ◁print⇒
     (
      (i > 0⇒
        (s[1 to i] print)).
      disp ← 1.
      l < i + 1⇒()
      s[i + 1 to l] print))

to { set (☞set←stream of vector 10. repeat(
        ◁}⇒(⇑set contents)
        set ← :))          '↑f1.uses stream to accumulate a vector↑f0.'

to indisp disp (:disp. ⇑disp eval)  '↑f1.redefines disp and evals a vector↑f0.'

to stringof x (☞x←:. ⇑indisp stream (x print. disp contents))
        '↑h75.↑f1.
        uses stream and indisp to give you the print-string of anything.
        ↑h60.↑f0.'


to obset i input : vec size end : each (
        ◁add⇒((size=☞end←end+1⇒(☞vec←vec[1 to ☞size←size+10]))
                vec[end]←:)
        ◁←⇒(0=vec[1 to end] find first :input⇒
                (SELF add input))
        ◁delete⇒(0=☞i←vec[1 to end] find first :input⇒(⇑false)
                vec[i to end]←vec[i+1 to end+1]. ☞end←end-1)
        ◁unadd⇒(☞input←vec[end]. vec[end]←nil.
                ☞end←end-1. ⇑input)
        ◁vec⇒(⇑vec[1 to end])
        ◁map⇒(☞i←0. ☞input←:. repeat
                (end<☞i←i+1⇒(done) input eval)⇑false)
        ◁print⇒(SELF map ☞(each print. sp))
        ◁is⇒(ISIT eval)
        isnew⇒(☞end←0. ☞vec←vector ☞size←4)
        )


'↑h90.↑f2.
PRETTY-PRINT
↑h60.↑f0.

        ↑h75.↑f1.
        This prints the code; classprint makes the header.
        ↑h60.↑f0.'

to show func t (
        :#func. ☞t←GET func ☞DO.
        null t ⇒ (⇑☞(no code)) pshow t 0.)
to pshow ptr dent i t :: x tabin index (:ptr :dent.
        (ptr length>4⇒(tabin dent)) disp←40.
        for i to ptr length-1
                (☞t ← ptr[i].
                t is vector ⇒(pshow t dent+3.
                        i=ptr length-1⇒()
                        ☞. = ☞x←ptr[i+1]⇒()
```

```
                    x is vector⇒()
                    tabin dent)
               i=1 ⇒(t print)
               0<☞x←index ☞(. , 's [ ] ⇒) t⇒
                    (x=1⇒(t print. ptr[i+1] is vector⇒() tabin dent)
                                                    t print)
               0=index ☞(: ☞ # ⇑ [ ⇙ ⇒ □) ptr[i-1]⇒(disp←32. t print)
               t print)
        disp←41)
to t each tabin index (ev)
t
to each (⇑vec[i])       'shorthand for mapping with obsets'
PUT obset ☞each #each.

to tabin n :: x (:n. disp←13. repeat
     (n > 6⇒
       (disp ← x[6].
        ☞n ← n - 6)
      done)
    disp ← x[n + 1])
(PUT tabin ☞x {string 0 32 fill string 2 fill string 3
       fill string 4 fill string 5 fill string 6}).
            'leave these blanks'
PUT pshow ☞tabin #tabin.
to index op byte s lb ub s2 lb2 ub2 (
       :s. :byte. ☞op←☞lb←☞s2←☞lb2←☞ub2←1. ☞ub←9999. CODE 40)
       '↑h75.↑f1.
       A piece of substr which runs faster.
       ↑h60.↑f0.'
PUT pshow ☞index #index.
done

'↑h90.↑f2.
FLOATING POINT↑h60.↑f0.
'

PUT float ☞DO ☞(0 CODE 42  'this does + - * / < = > ≤ ≠ ≥'
     ⇙ipart⇒(1 CODE 42)
     ⇙fpart⇒(2 CODE 42)
     ⇙ipow⇒
      (:x = 0⇒(⇑1.0)
       x = 1⇒()
       x > 1⇒
        (1 = x mod 2⇒
          (⇑SELF *(SELF * SELF)
           ipow x / 2)
         ⇑(SELF * SELF)
         ipow x / 2)
        ⇑1.0 / SELF ipow 0-x)
     ⇙epart⇒
      (SELF < :x⇒(⇑0)
       SELF < x * x⇒(⇑1)
       ⇑
        (☞y ← 2 * SELF epart x * x)
       +
        (SELF / x ipow y)
       epart x)
     ⇙is⇒(ISIT eval)
     ⇙print⇒
      (SELF = 0.0⇒(disp ← 48. disp←46. disp←48)
       SELF < 0.0⇒
        (disp ← 025.
         fprint - SELF)
       fprint SELF)
        )
to t fprint (ev)
t
to fprint n p q s : : fuzz z (
   '↑f1.Normalize to [1,10)↑f0.'
```

```
    (:n < 1⇒
      (☞p ← -(10.0 / n)
       epart 10.0)
     ☞p ← n epart 10.0)
     ☞n ← fuzz + n / 10.0 ipow p.
     (n≥10.0⇒(☞p←p+1. ☞n←n/10.0 'ugly fix for now'))
   '↑f1.Scientific or decimal↑f0.'
    (☞s ← fuzz*2.
     ¯4<p<6⇒(☞q ← 0.
       p < 0⇒
         (disp←z[1 to 1-p])
       ☞s ← s * 10.0 ipow p)
     ☞q←p. ☞p←0)
   '↑f1.Now print (s suppresses trailing zeros)↑f0.'
   do 9
    (disp ← 48 + n ipart.
     ☞p ← p - 1.
     ☞n ← 10.0 * n fpart.
     p < 0⇒
       (
         (p = ¯1⇒(disp ← 46))
        n < ☞s ← 10.0 * s⇒(done)))
    (p = ¯1⇒(disp ← 48))
   q ≠ 0⇒(disp←0145.
     q print))
PUT fprint ☞fuzz 5.0 * 10.0 ipow ¯9.
PUT fprint ☞z fill string 4
0.00
PUT float ☞fprint #fprint.
done

'↑h90.↑f2.
TEXT DISPLAY ROUTINES
↑h60.↑f0.

↑h75.↑f1.
Display frames are declared with five parameters. They are a left x, a width, a top y, a heigh
**t, and a string. Hence --
        ☞yourframe←dispframe 16 256 16 256 string 400.
-- gets you an area on the upper left portion of the display that starts at x,y 16,16 and is 2
**56 bits(raster units) wide and 256 bits high. The string (buf) serves as the text buffer, an
**d is altered by ← and scrolling.

There are actually two entities associated with display frames--frames and windows. Currently
**both are given the same dimensions upon declaration (see isnew).

The four instance variables defining the window are "winx", "winwd", "winy", and "winht". The
** boundaries of this rectangle are intersected with the physical display. The window actuall
**y used by the machine language will reduce the size of the window, if necessary, to be confi
**ned by the physical display. Clipping and scrolling are done on the basis of window boundar
**ies. If a character is in the window it will be displayed. If a string or character cause
**overflow of the bottom of the window, scrolling will occur.

The four instance variables defining the frame are "frmx", "frmwd", "frmy", and "frmht". Thi
**s rectangle may be smaller or larger than its associated window as well as the physical disp
**lay. Frame boundaries are the basis for word-wraparound. (Presently, if frmy+ frmht will
**cause overflow of the window bottom[winx+winht], frmht will get changed to a height consonan
**t with the bottom of the window. This has been done to manage scrolling, but may get change
**d as we get a better handle on the meaning of frames and windows.).

"Buf" is the string buffer associated with any given instance of dispframe. This is the strin
**g that is picked on the way to microcode scan conversion. When scrolling occurs, the first
**line of characters, according to frame boundaries, is stripped out and the remainder of the
**buffer mapped back into itself. If a "←" message would overflow this buffer, then scrolling
** will occur until the input fits.

"Last" is a "buf" subscript, pointing to the current last character in the buffer. That is, t
**he last character resulting from a "←".
```

"Lstln" also points into the buffer at the character that begins the last line of text in the
**frame. It is a starting point for scan conversion in the "←" call.

"Mark" is set by dread (see below) and points to the character in the buffer which represents
** the last prompt output by SMALLTALK; reading begins there. Mark is updated by scrolling, so
** that it tracks the characters. One could detect scrolling by watching mark.

"Charx" and "chary" reflect right x and top y of the character pointed to by "last".

The "reply" variable in the instance may be helpful in controlling things. When the reply is
**0, it means everything should be OK. That is, there was intersection between the window and
**display and intersection between the window and the frame. When reply is 1, there was no int
**ersection between the window and the display. A 2 reply means no intersection between window
** and frame. A 3 reply means window height less than font height or a character to wide for a
** frame -- hence no room for scan conversion of even one line of text. A 4 means that the fra
**me height has been increased in order to accomodate the input. A 5 means the bottom of the w
**indow (i.e. window x + window height) has been overflowed --hence that scrolling took place.
** A 6 means that both 4 and 5 are true.

"justify" is a toggle for right justifying the contents of a dispframe. The default is 0 an
**d means no justification. Setting it to 1 causes justification on frame boundaries.

The "font" variable allows for the association of a font other than the default font with the
**display frame. To get a different font into core say ☞something ← file <fontfilename> conte
**nts. Then you can say disp's (☞font←something) or you can declare the font at the same time a
**s the tdispframe is declared as e.g.
☞yourframe ← dispframe 3 40 3 40 string 20 font something.
↑h60.↑f0.'

```
(to dispframe input
        : winx winwd winy winht frmx frmwd frmy frmht
        last mark lstln charx chary reply justify buf font editor
        : sub frame dread reread defont (
◁ ← ⇒(0 CODE 51)
        '↑h75.↑f1.
        :s. s is number ⇒ (append this ascii char)
                s is string ⇒(append string)
                error.
        ↑h60.↑f0.'

◁'s⇒(⇑°eval)
        '↑h75.↑f1.
        Allows access to instance variables. For example,
                yourframe's (☞winx←32)
        will alter the value of window x in the instance of dispframe
        called ☞yourframe☞.
        ↑h60.↑f0.'

◁hasmouse⇒(frmx<mx<frmx+frmwd⇒(⇑frmy<my<frmy+frmht)⇑false)
        '↑h75.↑f1.
        Tells you if the mouse is within a frame.
        ↑h60.↑f0.'

◁show⇒(4 CODE 51 3 CODE 51)

◁display⇒(SELF show frame black)
        '↑h75.↑f1.
        Show clears the intersection of window and frame (see fclear,
        below) and displays buf from the beginning through last. A handy
        way to clean up a cluttered world.
        ↑h60.↑f0.'

◁fclear⇒(4 CODE 51)
        '↑h75.↑f1.
        Fclear clears the intersection of the window and frame. Hence if
        the frame is defined as smaller than the window, only the frame
        area will be cleared. If the frame is defined as larger than the
        window, only the window area will be cleared, since that space
```

```
        is in fact your "window" on that frame.
        ↑h60.↑f0.'

⍁hide⇒(4 CODE 51 frame white)

⍁put⇒(:input. ⍁at. ☞winx←☞frmx←:. ☞winy←☞frmy←☞chary←:.
        ☞last←0. ☞lstln←1. SELF←input. ⇑charx-winx)
        '↑h75.↑f1.
        For them as would rather do it themselves.
        ↑h60.↑f0.'

⍁wclear⇒(5 CODE 51)
        '↑h75.↑f1.
        Wclear clears the intersection of a window and the physical
        display.
        ↑h60.↑f0.'

⍁scroll⇒(2 CODE 51)
        '↑h75.↑f1.
        Scroll removes the top line of text from the frame's string buffer,
        and moves the text up one line.
        ↑h60.↑f0.'

⍁clear⇒(1 CODE 51)
        '↑h75.↑f1.
        Clear does an fclear and sets the "last" pointer into the string buffer
        to 0 and "lstln" to 1. It has the effect of cleaning out the string
        buffer as well as clearing the frame area.
        ↑h60.↑f0.'

⍁mfindc ⇒(7 CODE 51)
        '↑h75.↑f1.
        Find character.
        Takes two arguments -- x and y (typically msex and msey).
        Returns vector:
                vec[1] = subscript of char in string
                vec[2] = left x of char
                vec[3] = width of char
                vec[4] = topy of char
        If vec[1] is -1 x,y is after the end of the string.
        If vec[2] is -2 x,y is not in the window.
        Sample call:
                ☞myvec←yourframe mfindc mx my.
        ↑h60.↑f0.'

⍁mfindw ⇒(8 CODE 51)
        '↑h75.↑f1.
        Find word.
        Takes two arguments -- x and y (typically msex and msey).
        Returns vector:
                vec[1] = subscript of first char in word
                vec[2] = left x of word
                vec[3] = width of word
                vec[4] = topy of word
        If vec[1] is -1 x,y is after the end of the string.
        If vec[2] is -2 x,y is not in the window.
        Sample call:
                ☞myvec←yourframe mfindw mx my.
        ↑h60.↑f0.'

⍁mfindt ⇒(6 CODE 51)
        '↑h75.↑f1.
        Find token.
        Takes two arguments -- x and y (typically msex and msey).
        Returns vector:
                vec[1] = token count, ala Smalltalk token Spaces and
                        carriage returns are considered as delimiters,but
                        multiple delimiters do not bump the count. Text
                        delimited by single quotes is counted as one
```

```
                    token, and embedded text (i.e. more than one
                    quote in sequence will not cause the token count
                    to be bumped (allows for embedding strings
                    within strings).
                vec[2] = left x of word
                vec[3] = width of word
                vec[4] = topy of word

        If vec[1] is -1 x,y is after the end of the string or not in frame.
        If vec[2] is -2 x,y is not in the window.
        A sample call--
                ☞variable←yourframe mfindt mx my.
        ↑h60.↑f0.'

⍁read⇒(⇑dread)
        '↑h75.↑f1.
        Makes a code vector out of keyboard input. See dread below.
        ↑h60.↑f0.'

⍁reread⇒(⇑reread :)
        '↑h75.↑f1.
        Used by redo and fix.  Goes back n(its argument), prompts and does
        a read from there. See reread below.
        ↑h60.↑f0.'

⍁sub⇒(☞input ← sub :. SELF show. ⇑input)
        '↑h75.↑f1.
        Evals its argument in a sub-window.  Used by fix and shift-esc.
         See sub below.
        ↑h60.↑f0.'

⍁knows⇒(ev)
        '↑h75.↑f1.
        Whilst at the KEYBOARD, one can say
                "yourframe knows(DOIT)"
        and get a copy of the evaluator in the context of that instance of
        dispframe.  Allows access to instance variables without going
        through the 's path.
        ↑h60.↑f0.'

⍁frame ⇒ (apply frame)
        '↑h75.↑f1.
        Draws a border of the given color around the frame.  E.g.,
                yourframe frame -1.
        ↑h60.↑f0.'

⍁is ⇒(ISIT eval)

isnew ⇒ (☞winx←:frmx. ☞winwd←:frmwd. ☞chary←☞winy←:frmy.
        :frmht.☞winht←682-winy. :buf. ☞lstln←1.
        ☞mark←☞last←☞charx←☞reply←☞justify←0.
        ☞font ← (⍁font⇒(:input is string⇒(input) defont)defont)
        ⍁noframe⇒() frame black) ))

dispframe knows
to dread t flag (
        disp←20. ☞flag←false. ☞mark←last.
        (null #DRIBBLE⇒() DRIBBLE flush).
        repeat (050>disp←☞t←kbd⇒(
                t=010⇒(last<mark⇒(disp←buf[last+1])
                '↑h75.↑f1.
                Backspace only up to prompt.
                ↑h60.↑f0.'
                        buf[last+1]=047⇒(☞flag←flag is false))
                        '↑h75.↑f1.
                        Backspace out of string flips flag.
                        ↑h60.↑f0.'
                t=036⇒(flag⇒() done)
                '↑h75.↑f1.
```

```
                DOIT checks if in a string.
                ↑h60.↑f0.'
                t=047⇒(☞flag←flag is false)
                '↑h75.↑f1.
                Flag is true if in a string
                ↑h60.↑f0.'
                t=05⇒(sub ☞(ev). ☞last←last-1. disp show)
                '↑h75.↑f1.
                Shift-Esc make sub-eval.
                ↑h60.↑f0.'
                t=04⇒(disp←010. ☞done print. disp←036. ⇑☞(done))
                ))
        disp←13. ⇑read of stream of buf from mark+1 to last)
to sub disp (
        ☞disp←dispframe frmx+48 frmwd-64 frmy+14 frmht-28 string 300
        font font. disp clear. (:)eval)
        '↑h75.↑f1.
        Opens a sub-frame, and evals its argument in that context.
        ↑h60.↑f0.'

to frame a (☞a ← turtle at frmx - 1 frmy - 1.
        a'swidth ← 2 . a'sink←(⇐white⇒(¯1) ⇐black⇒(¯3) ⇐color⇒(:) ¯3)
        do 2 (a turn 90 go frmwd + 2 turn 90 go frmht + 2)       )
        '↑h75.↑f1.
        Draws a double line around the frame.
        ↑h60.↑f0.'


to reread n i p reader ((null :n⇒(☞n←1))
        ☞p←mark. ☞last←mark-2. disp show.
         for i to n
                (☞p←buf[1 to p-1] find last 20.
                p<1⇒(done))
        i<n+1⇒(error ☞(no code))
        ⇑read of stream of buf from p+1 to last)
        '↑h75.↑f1.
        Counts back n prompts (n is integer arg) and then does a read from
        there. Also erases          the line just typed.
        ↑h60.↑f0.'

done

to dclear (CODE 52)
        '↑h75.↑f1.
        This function takes five parameters --
        x width y height value, and "clears" the display rectangle thus
        defined to the "value" given. A 0 value, for example, puts all
        zeros into  the rectangle.
        ↑h60.↑f0.'


to dcomp (CODE 53)
        '↑h75.↑f1.
        Just like dclear only complement rectangle.
        ↑h60.↑f0.'


to dmove (CODE 54)
        '↑h75.↑f1.
        This function takes seven parameters -- source x width source y
        height destination x destination y mode. It takes the source
        rectangle (x and width mod 16'd as in dclear) and moves it to the
        destination x and y. Clipping will occur on display boundaries. The
        source will remain intact unless  it overlaps with the destination, in
        which case the overlapping portion of the destination wins. The
        mode is either store or or. Store (clearing destination area before
        scanning in source) is indicated with a mode 0. Or (which lays the
        source bits on top of whatever is in the destination rectangle) is
        indicated by mode -anything but zero-.
```

```
        ↑h60.↑f0.'

to dmovec (CODE 55)
        '↑h75.↑f1.
        Dmovec takes the same parameters as dmove, but in addition clears
        the non-intersecting source material. It is the general case of what
        happens on the display screen during a scroll, i.e. scrolling could be
        accomplished by saying disp's (dmovec winx winwd
        winy+fontheight winht-fontheight winx winy 0). A sample call --
                dmovec 0 256 0 256 256 256 0.
        This will move whatever is in the upper left hand corner of the
        display to x,y   256,256 -- and then erase the source area.
        ↑h60.↑f0.'


to redo (⇑(disp reread :) eval)
        '↑h75.↑f1.
        Causes re-evaluation of the input typed n prompts before this.
        Setting last←mark-2 makes the redo statement and its prompt
        disappear with a disp show.
        ↑h60.↑f0.'

to fix vec (☞vec←disp reread :.
        (disp sub ☞(veced vec)) eval)
        '↑h75.↑f1.
        Like redo, except that the previous input is given to the editor in a
        subwindow.  When editing is done, the resulting code is evalled
        before returning.
        ↑h60.↑f0.'


'↑h90.↑f2.
TURTLES
↑h60.↑f0.'

to turtle var : pen ink width dir x xf y yf frame : f (
        CODE 21
        '↑h75.↑f1.
                CODE 21 is equivalent to:
                ◁go⇒(draw a line of length :)
                ◁turn⇒(turn right : (degrees))
                ◁goto⇒(draw a line to :(x), :(y))
        ↑h60.↑f0.'
        ◁'s⇒(☞var ← ⁸. ◁←⇒(⇑var ← :)
                        ⇑var eval)
        ◁pendn⇒(☞pen ← 1. ⇑SELF)
        ◁penup⇒(☞pen ← 0. ⇑SELF)
        ◁black⇒(☞ink ← ¯3. ⇑SELF)
        ◁white⇒(☞ink ← ¯1. ⇑SELF)
        ◁xor⇒(☞ink ← ¯2. ⇑SELF)
        ◁is⇒(ISIT eval)
        ◁home⇒(☞x ← frame 's frmwd/2.
             ☞y ← frame 's frmht/2.
             ☞xf ← ☞yf ← 0. ☞dir←270. ⇑SELF)
        ◁erase⇒(frame fclear. ⇑SELF)
        ◁up⇒(☞dir ← 270. ⇑SELF)
        isnew⇒(☞ink ← ¯3. ☞pen ← ☞width ← 1.
                ☞frame ← (◁frame⇒(:) f).
                ◁at⇒(:x. :y. ☞xf ← ☞yf ← 0. ☞dir←270)
                SELF home)
        )
PUT turtle ☞f dispframe 0 512 0 684 string 1 noframe.
☞☺ ← turtle.


'↑h90.↑f2.
THE TRUTH ABOUT FILES AND DIRECTORIES
```

↑h75.↑f1.
a file is found in a directory ("dirinst") by its file name ("fname"), and has a one "page", 5
**12 character string ("sadr"). "rvec" is an optional vector of disk addresses used for rando
**m page access.

☞fi ←
<directory> file <string> old -- finds an old file named <string> in <directory> or returns f
**alse if does not exist or a disk error occurs.

☞fi ←
<directory> file <string> new -- creates a new file or returns false if it already exists. i
**f neither old or new is specified, an existing file named <string> will be found or a new fi
**le created. if <directory> is not specified, the current default directory is used.

<directory> file <string> delete -- deletes a file from a directory and deallocates its pages.
** do not delete the system directory (SYSDIR.) or bittable (SYS.STAT.), or any directories y
**ou create.

<directory> file <string> rename <string> -- renames file named by first string in <directory>
** with second string. currently not implemented for directory files.

<directory> file <string> load -- loads a previously "saved" memory image (Swat format), there
**by destroying your current state.

<directory> file <string> save -- saves your Smalltalk memory.

"leader" and "curadr" are the alto disk addresses of page 0 and the current page of the file,
**respectively. "bytec" is a character index into "sadr".

"dirty" = 1 if any label block integers ("nextp" thru "sn2") have been changed; = -1 if "sadr"
** has been changed; = 0 if the current page is clean. the user need not worry about this unl
**ess (s)he deals directly with the label or "sadr". it might be noted here that multiple inst
**ances of the same file do not know of each others activities or "sadr"s.

"status" is normally 0, -1 if end occurred with the last "set"; a positive number (machine lan
**guage pointer to offending disk command block (dcb)) signals a disk error.

the next 8 integers are the alto disk label block. "nextp" and "backp" are the forward and ba
**ckward alto address pointers. "lnused" is currently unused. "numch" is number of characters
**on the current page, numch must be 512, except on the last page. "pagen" is the current page
** number. page numbers are non-negative integers, and the format demands that the difference
**in consecutive page numbers is 1. normal file access starts at page 1, although all files po
**ssess page 0 (the "leader" page). "version" numbers > 1 are not implemented. "sn1" and "sn2"
** are the unique 2-word serial number for the file.

the class function "ncheck" checks that file names contain alphabetic or "legal" characters or
** digits, and end with a period.
↑h60.↑f0.'

```
(to file : dirinst fname sadr rvec leader curadr bytec dirty status nextp
        backp lnused numch pagen version sn1 sn2 : ncheck x (

        ⇐⇒ (17 CODE 50)
                '↑h75.↑f1.
                fi←<integer>, <string>, or <file> --
                :x is string⇒ (for i to x length (SELF←x[i]))
                x is file⇒ (repeat (x end⇒ (done) SELF←x next))
                (numch<☞bytec←bytec+1⇒
                  (SELF set to write (pagen+bytec/512) bytec mod 512))
                sadr[bytec]←x ⍇ 0377
                ↑h60.↑f0.'

        ⇐next⇒ ((⇐word⇒ (⇐⇒ (7)
                        '↑h75.↑f1.
                        fi next word←<integer> -- write integer.
                        possibly increment pointer to word boundary.
                        (0=bytec ⍇ 1⇒ () ☞bytec←bytec+1)
                        SELF ← :x/256, SELF ← x mod 256.
```

```
                ↑h60.↑f0.'

                 6)
                '↑h75.↑f1.
                fi next word -- read an integer
                (0=bytec ⍰ 1⇒ () ☞bytec←bytec+1)
                ⇑(SELF next*256) + SELF next
                ↑h60.↑f0.'

          ⍁into⇒ (16)
                '↑h75.↑f1.
                fi next into <string> -- read a string
                for i to :x length(x[i]←SELF next).⇑x
                ↑h60.↑f0.'

          25) CODE 50)
                '↑h75.↑f1.
                fi next -- read a character
                (numch<☞bytec←bytec+1⇒
                 (SELF set to read (pagen+bytec/512)
                   bytec mod 512⇒ () ⇑0))  ⇑sadr[bytec]
                ↑h60.↑f0.'

⍁set⇒ (⍁to. (⍁end⇒(13)
                '↑h75.↑f1.
                fi set to end -- set file pointer to end of file.
                SELF set to read 037777 0
                ↑h60.↑f0.'

          ⍁write⇒(5)
                '↑h75.↑f1.
                fi set to write <integer> <integer> -- set file
                pointer to :spage :schar. if current page is dirty,
                or "reset", "set to end" or page change
                occurs, flush current page. read pages until
                pagen=spage. allocate new pages after end if
                necessary (-1 512 is treated as start of next
                page, i.e. pagen+1 0). ☞bytec←schar
                ↑h60.↑f0.'

          ⍁read. 4) CODE 50)
                '↑h75.↑f1.
                same as "write" except stop at end
                ↑h60.↑f0.'

⍁skipnext⇒ (18 CODE 50)
          '↑h75.↑f1.
          fi skipnext <integer> -- set character pointer relative to
          current position. (useful for skipping rather than reading,
          or for reading and backing up, but "end" may not work if
          "bytec" points off the current page) ☞bytec← bytec + :.
          ↑h60.↑f0.'

⍁end⇒ (10 CODE 50)
          '↑h75.↑f1.
          fi end -- return false if end of file has not occurred.
          nextp=0⇒ (bytec<numch⇒(⇑false))⇑false
          ↑h60.↑f0.'

⍁'s⇒ (⇑⁸ eval)

⍁flush⇒ (12 CODE 50)
          '↑h75.↑f1.
          fi flush -- dirty=0⇒ () write current page
          ↑h60.↑f0.'

⍁writeseq⇒ (22 CODE 50)
          '↑h75.↑f1.
          transfer words from memory to a file
```

```
        :adr. :count. for i←adr to adr+count-1
        (SELF next word ← mem i)
        ↑h60.↑f0.'

⪡readseq⪢ (21 CODE 50)
        '↑h75.↑f1.
        ...from a file to memory...(mem i ← SELF next word)
        ↑h60.↑f0.'

⪡is⪢ (ISIT eval)

⪡remove⪢ (dirinst forget SELF)
        '↑h75.↑f1.
        remove file from filesopen list of directory
        ↑h60.↑f0.'

⪡close⪢ (dirinst 's bitinst flush.
         SELF flush. SELF remove. ⇑☞closed)
        '↑h75.↑f1.
        fi close or ☞fi←fi close (if fi is global) -- flush bittable
        and current page, remove instance from filesopen list of
        directory
        ↑h60.↑f0.'

⪡shorten⪢ (⪡to. ⪡here⪢ (SELF shorten pagen bytec) 14 CODE 50)
        '↑h75.↑f1.
        fi shorten to <integer> <integer> -- shorten a file SELF set
        to read :spage :schar. ☞x←nextp. ☞nextp←0.
        ☞numch←schar. ☞dirty←1. deallocate x and successors
        ↑h60.↑f0.'

⪡print⪢ (disp ← fname)
        '↑h75.↑f1.
        file prints its name
        ↑h60.↑f0.'

⪡reset⪢ (11 CODE 50)
        '↑h75.↑f1.
        fi reset -- reposition to beginning of file
        SELF set 1 0
        ↑h60.↑f0.'

⪡intostring⪢(SELF set to end.
        ☞x ← string bytec + 512 * pagen - 1.
        SELF reset. ⇑SELF next into x)

⪡random⪢ (SELF set to end. ☞rvec ← vector pagen.
          for x to rvec length (SELF set x 0. rvec[x] ← curadr))
        '↑h75.↑f1.
        fi random -- initialize a random access vector to be used
        in fi set... new pages appended to the file will not be
        randomly accessed
        ↑h60.↑f0.'

⪡pages⪢ (20 CODE 50)
        '↑h75.↑f1.
        fi pages <integer> ... <integer> -- out of the same great
        tradition as "mem" comes the power to do potentially
        catastrophic direct disk i/o (not for the faint-hearted).
         :coreaddress. :diskaddress. :diskcommand. :startpage.
        :numberofpages. :coreincrement. if -1 = coreaddress,
        copy "sadr" to a buffer before the i/o call. diskaddress
        (=-1 yields "curadr") and diskcommand are the alto disk
        address and command. startpage is relevant if label checking
        is performed. numberofpages is the number of disk pages
        to process. coreincrement is usually 0 (for writing in same
        buffer) or 256 for using consecutive pages of core. use
        label block from instance of "fi". copy label block from
        instance. perform i/o call. copy "curadr" and label block
```

```
        into instance. if -1=coreaddress copy buffer to ☞sadr☞.
        ↑h60.↑f0.'

isnew⇒ (☞fname←ncheck :. fname is false⇒
            (error ☞(bad file name) ⇑nil)
        (null ☞dirinst ← #curdir⇒
            (☞dirinst ← directory 's defdir.  dirinst open)).
            '↑h75.↑f1.
            set directory instance for file. if curdir is nil
            because file was not called from the context of a
            directory instance, use the default directory
            ↑h60.↑f0.'

        ☟exists⇒ (24 CODE 50. ⇑fname)
            '↑h75.↑f1.
            return false if file name does not occur in the
            directory
            ↑h60.↑f0.'

        ☟delete⇒ (15 CODE 50. ⇑☞deleted)
            '↑h75.↑f1.
            delete a file (see intro)
            ↑h60.↑f0.'

        ☞sadr ← (☟using⇒ (:) string 512).
            '↑h75.↑f1.
            set up file string buffer
            ↑h60.↑f0.'

        ☟rename⇒ (☞x ← ncheck :. x is false⇒
                    (error ☞(bad new name)⇑nil)
                file x exists⇒ (error ☞(name already in use))
                2 CODE 50.  ☞fname ← x.  23 CODE 50.
                SELF set 0 12.  SELF ← fname length.
                SELF ← fname.  SELF flush.  ⇑fname)
            '↑h75.↑f1.
            check that the new name is not already in use.
            lookup the original file and change its name in its
            directory, and in its leader page
            ↑h60.↑f0.'

        ☟load⇒ (2 CODE 50. 8 CODE 50)
            '↑h75.↑f1.
            lookup an old file and load (overlay) a Swat
            memory image; return via save.
            ↑h60.↑f0.'

        (☟old⇒ (2)
         ☟new⇒ (dirinst 's filinst is file⇒ (3) 19)
         1) CODE 50.
            '↑h75.↑f1.
            find an old file or add a new entry (updating
            create/write/read date and time, and file name
            (as a Bcpl string) in its leader page. special
            handling for new directories). machine code
            may return false
            ↑h60.↑f0.'

        ☟save⇒ (SELF set to write 256 0.  SELF reset.
            dirinst close.  9 CODE 50)

            '↑h75.↑f1.
            allocate a file, close the directory (other files
            e.g. DRIBBLE, and directories should be already
            closed), and write out the memory image as a
            Swat file.  when arriving here from a "load",
            return false; otherwise return the file instance.
            ↑h60.↑f0.'
```

```
                <intostring⇒(⇑SELF intostring)
                dirinst remember SELF) ))
                        '↑h75.↑f1.
                        finally, file puts itself into the filesopen list of its
                        directory
                        ↑h60.↑f0.'
file 's(ev)
to ncheck str i x :: legal (☞str←:.
        (str is string⇒(str length < 255⇒() ⇑false) ⇑false)
        for i to str length
                (☞x ← str[i].
                0140 < x < 0173⇒ ('lowercase')
                 057 < x <  072⇒ ('digit')
                  0 < legal[1 to 6] find x⇒ ('legal')
                0100 < x < 0133⇒ ('uppercase')
                ⇑false)
        x=056⇒(⇑str) ⇑str+ ☞.chars)
'↑h75.↑f1.
check that the file name is a proper length string containing only lower/upper case letters, d
**igits, or legal characters. if name does not end with a period, append one.
↑h60.↑f0.'

PUT ncheck ☞legal fill string 6
+-()⇑⇒.
done

'↑h75.↑f1.
a directory is found in a directory ("dirinst"), has a bittable file ("bitinst") for allocatin
**g new pages, a file of file entries ("filinst" -- file names, disk addresses etc.), and a li
**st of currently open files ("filesopen" which is an "obset"). each file must ask its directo
**ry for the bittable when page allocation is necessary, and the system directory (via its loc
**al directory) for the disk number.

☞di ←
 <directory> directory <string> old/new

currently, <directory> and old or new must be specified.

"dirname" is the system directory name and "bitname" is the bittable name. "curdir" is a class
** variable bound to the last directory instance "opened", and provides information "who calle
**d you" (i.e. CALLER) to a file or directory. "defdir" is a default directory, initially set
**to dp0, which is invoked when "curdir" fails to be a directory, i.e. file was not called in
**the context of a directory, but globally
↑h60.↑f0.'

(to directory name exp : dirinst bitinst filinst filesopen : dirname bitname
        curdir defdir (

        <file⇒ (SELF open. ⇑apply file)
                '↑h75.↑f1.
                di file <string>... -- open directory. create file instance
                (see file intro)
                ↑h60.↑f0.'

        <open⇒ (☞curdir ← SELF.  filinst is file⇒ ()
                bitinst⇒(☞filinst←file filinst old. ☞bitinst←file bitinst old)
                 ☞filinst ← (<new⇒(file filinst new) file filinst old).
                 ☞bitinst ← dirinst 's bitinst.  dirinst remember SELF)
                '↑h75.↑f1.
                di open -- (normally not user-called since access to the
                directory always reopens it) initialize directory file and
                bittable instances: a "subdirectory" uses the bittable of
                its system directory and puts itself into that filesopen list.
                ↑h60.↑f0.'

        <is⇒ (ISIT eval)

        <remember⇒ (filesopen ← :)
```

```
forget⇒ (filesopen delete :)
        '↑h75.↑f1.
        add or delete file instances in filesopen.
        ↑h60.↑f0.'

print⇒ (disp←0133. filesopen print. disp←0135)
        '↑h75.↑f1.
        di or di print. --print the filesopen list.
        ↑h60.↑f0.'

map⇒ (SELF open. ☞exp ← :. filinst reset.
        repeat (filinst end⇒ (cr. done)
         1024 > ☞name← filinst next word⇒
                (name < 2⇒ () filinst skipnext 2*name-1)
        filinst skipnext 10.
        ☞name ← filinst next into string filinst next.
        exp eval))
        '↑h75.↑f1.
        di map <expression> -- evaluate an expression for each
        file name
        ↑h60.↑f0.'

list⇒ (SELF map ☞(disp←name. sp))
        '↑h75.↑f1.
        di list -- print the entry names contained in filinst
        ↑h60.↑f0.'

flush⇒(filesopen map ☞(each flush))

close⇒ ((filinst is file⇒ (SELF flush. ☞filesopen ← obset.
        ☞filinst ← filinst 's fname.
        dirinst is directory⇒ (dirinst forget SELF. ☞bitinst ← false)
        ☞bitinst ← bitinst 's fname)). ⇑☞closed)
        '↑h75.↑f1.
        di close (e.g. dp0 close) or ☞di←di close (to release
        instance) --close a directory by closing all files and
        directories in its filesopen list and deleting it from the
        filesopen list of its directory. this is currently one way
        to regain space by closing unwanted file instances, and
        to change disk packs.
        ↑h60.↑f0.'

use⇒ (☞defdir ← SELF)
        '↑h75.↑f1.
        di use -- change the default directory.
        ↑h60.↑f0.'

's⇒ (⇑⁸ eval)

free⇒ (SELF open. ☞exp ← 0.
        bitinst set to 1  609*dirinst ⍁¯1.
        do 609  (255 = ☞name ← bitinst next⇒ (☞exp ← exp + 8)
                name > 0⇒ (
                repeat (☞exp ← exp + name ⍁1.
                        0 = ☞name ← name ⍁¯1⇒(done)))).
        ⇑4872 - exp)
        '↑h75.↑f1.
        di free -- return number of free pages from the
        bittable associated with this directory.
        ↑h60.↑f0.'

directory⇒ (SELF open. ⇑apply directory)
        '↑h75.↑f1.
        di directory <string>... -- open directory. create directory
        instance
        ↑h60.↑f0.'

isnew⇒ (☞filesopen ← obset.
        ↖device ⇒(☞dirinst ← :.  ☞filinst ← dirname.
```

```
                    ☞bitinst ← (1<dirinst⇒(bitname + ☞two chars) bitname))
                ☞dirinst ← curdir. ☞filinst ← :. ☞bitinst ← false.
                ◅new⇒(SELF open new) ◅old. SELF open)))
                '↑h75.↑f1.
                store the directory file name in filinst; "subdirectories" use
                "curdir" as their directory, are flagged by "false bitinst" and
                then opened. for system directories, dirinst is a number
                indicating physical device location and size, and bitinst
                contains the file name for the appropriate bittable. the
                directory is not immediately opened.
                ↑h60.↑f0.'

directory 's(ev)
☞dirname ← fill string 7
SYSDIR.
☞bitname ← fill string 9
SYS.STAT.
done
        '↑h75.↑f1.
        names of the system directory and bittable
        ↑h60.↑f0.'

☞dp0 ← directory device 0.
☞dp1 ← directory device 1.
☞dp2 ← directory device 2.
☞dp3 ← directory device 3.
dp0 use
        '↑h75.↑f1.
        create some system directory instances which are initially closed.
        dp0 is the same (default) directory that the Operating System
        believes in.
        dp1 can similarly access another Model 31 disk or a Model 44.

        dp2 and dp3 provide a mechanism for writing on the other halves of
        a Model 44, with some preliminary initialization; files written
        through dp2 (dp3) can be read by dp0 (dp1) also. you should only
        overwrite or delete such files via the directory that created them --
        one useful exception is that dp0 or the Operating System (e.g.
        subsystems like CONCAT and ETHERMCA) can overwrite files in dp2
        as long as the files remain the same length (e.g. Swat files).
        ↑h60.↑f0.'

to error adr ptr arec :: c sub (
        (0=☞adr←mem 0102⇒(◅knows⇒(ev ⇑) dson. :ptr))
        ☞arec←leech AREC.
        disp sub ☞((0=adr⇒(ptr print)
                mem 0102←0. disp←0377[] mem adr.
                for adr←adr+1 to adr+(mem adr)[]¯9 (
                        ☞ptr←mem adr.
                        disp←ptr[]¯8. disp←ptr[] 0377))
        cr c ev))
error knows
to c class code l1 l2 adr vadr i cpc (
        null arec[5]⇒(.) ☞arec←leech arec[5]. ☞class←arec[0].
        (GET class 'TITLE) print. ☞': print.
        ☞adr ← ☞vadr ← arec[1][]
        ☞l1 ← leech nil. ☞l2 ← leech l1.          'Fasten seat belts'
        repeat (l2[1][] vadr.
                05350=0177770[]1[0][]⇒done) 'finds start of code vector'
                ☞vadr ← vadr-1)
        ☞code ← l2[1]. l2[1][] ¯1.
        ☞cpc ← adr-vadr+1.
        for i to code length
                (i<cpc-5⇒(disp←056) i>cpc+5⇒(disp←056)
                sp. (i=cpc⇒(disp←031))
                code[i] is vector⇒(☞' () print) code[i] print).
        )
to sub disp (☞disp ← GET USER ☞disp. (:) eval)
done
```

```
to kbck (1 CODE 20)
        '↑h75.↑f1.
        Returns true if the keyboard has been hit.
        ↑h60.↑f0.'

to button n (⇑:n=mouse 7)
        '↑h75.↑f1.
        Returns true if that pattern is being held down
        ↑h60.↑f0.'


'↑h90.↑f2.
THE SMALLTALK EDITOR ---
↑h60.↑f0.'


to edit func t (:#func.
        ☞t←GET func ☞DO.
        null t ⇒ (⇑☞(no code))
        ⇦title⇒ (evapply veced classprint func header to ☞(eval) in GLOB)
        PUT func ☞DO veced t.
        ⇑☞edited)
        '↑h75.↑f1.
        Edit picks up a code vector, makes sure it is not empty and calls
        veced to edit the code body.  If you say edit foo title, veced will
        edit the header as well, and the changed form will be evalled upon
        exit to redefine the function, title and all.

        Veced can be used on any vector, and is used by FIX as well as
        EDIT.  It creates two new windows within the default DISP which
        exists when it is called.  One is used for a menu of commands,
        the other becomes the new default window DISP. The new default
        is passed to an intermediary; and the newly edited vector is
        returned.
        ↑h60.↑f0.'

(to veced back newdisp  menu x :: menuwidth menulen menustr
ed edpush edtarget gettwo bugin getvec  (
        ⇦knows⇒(ev)
        ☞back←false.
        disp fclear.
        disp's (☞menu←dispframe frmx+frmwd-menuwidth menuwidth
                        frmy (frmht>139⇒(frmht) 140) string 70
                        font font.
                menu ← menustr.
                mem 0425 ← frmy + 103.
                ☞newdisp ← dispframe frmx frmwd-menuwidth+2
                        frmy frmht string buf length
                        font font noframe)
        :x. ☞x ← indisp newdisp (ed x).
        menu hide. disp display.
        ⇑x)  )


veced knows

☞menuwidth ← 64.
☞menustr←string 0.
☞menulen ← 10.
do menulen (☞x←fill string 9.
        ☞menustr←menustr+x[1 to x[1 to 9]find 13]).
 Add
 Insert
 Replace
 Delete
 Move
 Up
 Push
```

```
Enter
Leave
Exit

to ed ptr l n nrun command temp i nv n1 fnth hfnth (
        ☞command ← 0.
        :ptr.
        ☞fnth ← (leech disp'sfont)[2]□
        ☞hfnth ← fnth/2.
        repeat(
                ☞l←ptr length.
                back⇒(done with ptr)
                mem 0424 ← menu 's frmx + 48.
                menu show. disp clear
                ☞nv←0.
                for n to l-1
                        (ptr[n] is vector⇒(disp←044. sp
                                ☞nv←nv+1. ☞n1←n)
                        ptr[n] print. disp←32)
                cr cr.
                ☞command ← edcomp bugin menu menulen both.
                mem 0424 ← disp 's (frmx + frmwd/2).

                ☞(
                (☞ptr←vecmod ptr l 0 read)
                (☞ptr←vecmod ptr edcomp edtarget both 0 read)
                (gettwo. ☞ptr←vecmod ptr n nrun read)
                (gettwo. ☞ptr←vecmod ptr n nrun nil)
                (gettwo. ☞temp ← ptr[n to n+nrun]
                        temp[nrun + 1] ← nil.
                        ☞i←edcomp edtarget both.
                        ☞ptr←vecmod ptr n nrun nil.
                        (i>n ⇒ (☞i←i-nrun))
                        ☞ptr←vecmod ptr i 0 temp)
                (getvec⇒(☞ptr←vecmod ptr n 1 ptr[n]) again)
                (gettwo. edpush)
                (getvec⇒(ptr[n]←ed ptr[n]) again)
                (done with ptr)
                (☞back←true. done with ptr)
                        ) [command] eval.
                )
        )
        '↑h75.↑f1.
        The heart of ED is a vector, containing as its elements code
        vectors. The giant vector is indexed to get the particular piece of
        program, and it is sent the message EVAL. Note that the order of
        the segments in ED1 should match the order of the atom names in
        MENUVEC.
        ↑h60.↑f0.'


to edpush ins (☞ins←vector 2.
        ins[1]← ptr[n to n+nrun]. ins[1][nrun+1]←nil.
        ☞ptr←vecmod ptr n nrun ins)

to gettwo t1 n2 (☞n←edcomp edtarget top.
            ☞n2←edcomp edtarget bot.
        ☞nrun ← 1+n2-n.
        nrun<1⇒(☞n←n2. ☞nrun←2-nrun))

to bugin someframe max index(
        ☞someframe ← :.
        ☞max ← :.
        repeat (button 0 ⇒ (repeat (
                        button 7 ⇒(disp sub ☞(ev))
                        button 0 ⇒()
                        done)
                done)
                )
```

```
        ☞index←someframe mfindt mx my.
        0 < index[1] ≤ max ⇒
                (⇑index)
        '↑h75.↑f1.
        returns token index, if within range, else
        ↑h60.↑f0.'
        again
        '↑h75.↑f1.
        causes an exit out of this command by restarting ed's
        repeat
        ↑h60.↑f0.'
        )

to edtarget (⇑ bugin disp 1)

to getvec (nv=1⇒(☞n←n1. ⇑true)
        ⇑ptr[☞n←edcomp edtarget both] is vector)

to edcomp compvec y hth (:compvec.
        ☞y←compvec[4].
        ☞hth←(≪both⇒(fnth)≪top⇒(hfnth)
                ≪bot⇒(☞y←y+hfnth. hfnth))
        dcomp compvec[2] compvec[3] y hth
        ⇑compvec[1]
        )
done

'↑h90.↑f2.
BOOTSTRAPPING REVISITED
↑h60.↑f0.'

to classprint fn a b i j k flags clsv clsm arecv arecm instv instm code (
        :#fn. ☞code ← GET fn ☞DO. null code⇒(☞(no code))
        ☞a←leech #fn. ☞b←vector 1. ☞b←leech b. ☞clsm←☞arecm←☞instm←0.
        ☞k←a[1]□. ☞clsv←vector k. ☞arecv←vector k. ☞instv←vector k.
        '↑h75.↑f1.
        Pull symbols out of class table
        ↑h60.↑f0.'
        for i←4 to 4+2*k by 2
        '↑h75.↑f1.
        k is no. dbl entries -1, here
        ↑h60.↑f0.'
                (☞k←a[i]□.
                k=(0-1)⇒(again). ☞flags ← k⍁¯14.
                '↑h75.↑f1.
                0=class, 2=arec, 3=inst
                ↑h60.↑f0.'
                flags=0⇒(0=☞(DO TITLE SIZE) [1 to 3] find a[i]⇒
                        (clsv[☞clsm←clsm+1] ← a[i]))
                b[2]□ ← k□⍁3777. ☞j←a[i+1]□.
                (flags=2⇒(arecv[j-6] ← b[2]. arecm<j-6⇒(☞arecm←j-6))
                        instv[j+1] ← b[2]. instm<j+1⇒(☞instm←j+1))
                )

        '↑h75.↑f1.
        Now make up input form.
        ↑h60.↑f0.'
        ☞a ← vector 6+arecm+instm+clsm.
        a[1] ← ☞to. a[2] ← GET fn ☞TITLE.
        a[3 to ☞j←2+arecm] ← arecv.
        (0<instm+clsm⇒ (a[☞j←j+1]←☞:. a[j+1 to ☞j←j+instm] ← instv.
                0<clsm⇒ (a[☞j←j+1]←☞:. a[j+1 to ☞j←j+clsm] ← clsv)))
        ≪header⇒(a[j+1]←code. ⇑a)
        for i to j (a[i] print. disp←32)
        showpretty⇒(pshow code 3) code print)

to show showpretty (☞showpretty←true. showev ⍟)

to showev shAtom shVal (:shAtom. cr.
```

```
        (shAtom is atom⇒
                (☞shVal ← shAtom eval.
                (null GET shVal ☞DO⇒
                        (☞☞ print. shAtom print. ☞← print.
                        (shVal is vector⇒ (☞☞ print)
                         shVal is atom⇒(☞☞print)
                                 null shVal⇒(☞nil print))
                        shVal print. ☞. print)
                classprint shVal))
        shAtom print)
        disp←30.)
'↑h75.↑f1.
*****Keyboard translation*****
↑h60.↑f0.'

to kbd (⇑kmap[TTY])
☞kmap ← string 0377.
        for i←0200 to 0377(kmap[i] ← 0177) '↑f1.ILLEGAL GETS ↑f0.'
        for i←001 to 0177(kmap[i] ← i) '↑f1.↑f1.REGULAR ASCII'S↑f0.'

                        '↑f1.CHAR -- KEYBOARD↑f0.'
kmap[0341]←kmap[0301]←kmap[0274]←kmap[0254]←01.
                                        '↑f1.≤ -- ↑↑A or ↑↑, or ↑↑SHF,↑f0.'
kmap[0342]←kmap[0302]←kmap[0236]←kmap[0237]←kmap[037]←kmap[036]←02.
                                '↑f1._ -- ↑↑B or any TOP BLANK KEY.↑f0.'

kmap[0343]←kmap[0303]←kmap[0272]←03. '↑f1.≤ -- ↑↑C or ↑↑SHF;↑f0.'
kmap[0344]←kmap[0304]←04. 'f1.↑↑D  'done' ↑f0.'
kmap[0345]←kmap[0305]←kmap[023]←05. '↑f1.E -- ↑↑E or ↑↑SHF ESC↑f0.'
kmap[0346]←kmap[0306]←kmap[0262]←06. '↑f1.@ -- ↑↑F or ↑↑2↑f0.'
kmap[0347]←kmap[0307]←kmap[0273]←07. '↑f1.' -- ↑↑G or ↑↑;↑f0.'
kmap[0353]←kmap[0313]←kmap[0245]←013. '↑f1.K -- ↑↑K or ↑↑SHF5↑f0.'
kmap[0356]←kmap[0316]←kmap[0275]←016. '↑f1.* -- ↑↑N or ↑↑=↑f0.'
kmap[0357]←kmap[0317]←kmap[0242]←017. '↑f1." -- ↑↑O or ↑↑SHF'↑f0.'
kmap[0360]←kmap[0320]←kmap[0271]←020. '↑f1.( -- ↑↑P or ↑↑9↑f0.'
kmap[0361]←kmap[0321]←kmap[0261]←021. '↑f1.! -- ↑↑Q or ↑↑1↑f0.'
kmap[0362]←kmap[0322]←kmap[0300]←022. '↑f1.° -- ↑↑R or ↑↑SHF2↑f0.'
kmap[0363]←kmap[0323]←023.          '↑f1.S -- ↑↑s↑f0.'
kmap[0364]←kmap[0324]←024.          '↑f1.T -- ↑↑T↑f0.'
kmap[0365]←kmap[0325]←kmap[0255]←kmap[0140]←025. '↑f1.' -- ↑↑U or ↑↑-↑f0.'
kmap[0366]←kmap[0326]←kmap[0265]←026. '↑f1.% -- ↑↑V or ↑↑5↑f0.'
kmap[0367]←kmap[0327]←kmap[0376]←027. '↑f1.~ -- ↑↑W or ↑↑SHF6↑f0.'
kmap[0370]←kmap[0330]←kmap[0246]←030. '↑f1.^ -- ↑↑X or ↑↑SHF7↑f0.'
kmap[0371]←kmap[0331]←kmap[0277]←031. '↑f1.» -- ↑↑Y or ↑↑SHF/↑f0.'
kmap[0372]←kmap[0332]←kmap[0276]←kmap[0256]←032.
                                        '↑f1.≥ -- ↑↑Z or ↑↑. or ↑↑SHF.↑f0.'
kmap[0333]←kmap[0264]←033. '↑f1.$ -- ↑↑[ or ↑↑4↑f0.'
kmap[0334]←kmap[0267]←034. '↑f1.& -- ↑↑\ or ↑↑7↑f0.'
kmap[0335]←kmap[0375]←035. '↑f1.§ -- ↑↑]↑f0.'
kmap[0337]←kmap[0336]←kmap[0222]←kmap[0212]←kmap[022]←kmap[012]←036.
                '↑f1.▮ -- LF or ↑↑ ← or ↑↑SHF←↑f0.'
kmap[0247]←0174.          '↑f1.' -- SHF\ or ↑↑'↑f0.'
kmap[0257]←0176.          '↑f1.? -- SHF6 or ↑↑/↑f0.'
kmap[0263]←043.          '↑f1.# -- SHF3 or ↑↑3↑f0.'
kmap[0270]←052.          '↑f1.* -- SHF8 or ↑↑8↑f0.'
kmap[0220]←kmap[0210]←kmap[020]←010. '↑f1.ALL BS'S↑f0.'
kmap[0225]←kmap[0215]←kmap[025]←015. '↑f1.ALL CR'S↑f0.'
kmap[0240]←kmap[0230]←kmap[030]←040. '↑f1.ALL SP'S↑f0.'

to filout disp flist i showpretty (☞showpretty ← ☞pretty.
        dsoff (:disp is string⇒ (☞disp←file disp⇒ () error ☞(file error)))
        (☞add⇒(disp set to end))
        (null :flist⇒(defs map ☞(showev each. cr))
          for i to flist length-1 (showev flist[i]. cr))
        disp shorten to here. disp close. dson.)

        '↑f1.Filout basically does a show in a context where the display is
        replaced by a file. filout pretty <file> or <string = file name> add
        <vector>.  if "pretty" is used, the text representation is neater but
```

```
        takes longer to generate. if "add" is used, function definitions are
        appended to the file.  if <vector> is not specified, "defs" is used.
        ↑h60.↑f0.'

to filin fi :: ev (↞'s⇒(⇑☉eval)
    dsoff.
    (:fi is string⇒(☞fi ← file fi old⇒()
                    dson ⇑false))
    repeat
     (fi end⇒(done)
      dsoff.
      cr (read of fi) eval print.
      dson).
    fi close.
   )
filin's (to ev (repeat(cr (read of fi) eval print)))

        '↑h75.↑f1.
        Filin basically does a read-eval-print loop, but gets its input from a
        file instead of a dispframe.
        ↑h60.↑f0.'

 to type f t ((:f is string⇒(
                    ☞f ← file f old⇒(f remove)
                    ⇑false))
        ☞t←string 30.
        repeat(f end⇒(done) disp←f next into t))

 to t fool  :: fontname (dispframe 's (☞defont ←  file fontname intostring).
        ☞disp←dispframe 16 480 514 168 string 520.
        disp ← version. ☞defs ← obset.
        ☞fool←#to. to to toAtm (CODE 19 defs←toAtm. toAtm)
        PUT USER ☞DO ☞(cr read eval print). ☞t←0.)
 PUT t ☞fontname fill string 6
 ST8.AL
 ☞version←fill string 34
 Welcome to SMALLTALK [September 9]

 to expand x (:x. disp 's (☞winy←☞frmy←frmy-x. frame black)
        disp show CODE 38)
        '↑h75.↑f1.
        t is called to set up a display frame, and defs and then self-destructs
        to save space. expand can be called to grab some storage from the
        display area to augment the SMALLTALK workspace. expand 200
        would take 200 lines off the top of the display and increase core by
        6400 words.


  THE SMALLTALK READ ROUTINE (name changed to protect ev)
  ↑h60.↑f0.'

  (to junta scanner : : read1 tablscan rdnum mknum rdstr rtb1 type
          letbit digbit sepbit atbits qtbit
    (↞'s⇒(⇑☉eval)
     ↞of⇒((:scanner is string⇒
        (☞'scanner ← stream of scanner))
     ☞scanner ← tablscan scanner type.
     ⇑read1 rtb1)
     ⇑disp read))

  junta 's(ev)

  to read1 rbuf rdtb flag (
     :rdtb.
     ☞rbuf ← stream of vector 10.
     scanner read.
     ⇑rbuf contents)

  to tablscan mask : source type seq isfil nxtchr (
```

```
  ⇦next⇨
   (CODE 14 next.
    'CODE 14 is equivalent to...
    :mask=0⇨(☞t←string 1. t[1]←nxtchr.
                ☞nxtchr←source next. ⇑atom t)
    seq reset.
    repeat
     (0 = nxtchr⇨(done).
      0 = mask ⍰ type[nxtchr + 1]⇨(done).
      seq ← nxtchr.
      ☞nxtchr ← source next)
     ⇑seq contents')
   ⇦skip⇨(☞nxtchr ← source next)
   ⇦read⇨(repeat
     (rdtb[nxtchr + 1] eval))
   isnew⇨
    (:source.
     :type.
     ☞seq ← stream.
     (source is file⇨(☞isfil ← 1))
     SELF skip))

to rdnum sign base n fs(
   ☞sign ← (nxtchr=025⇨(scanner skip. ¯1)1).
   ☞base ← (nxtchr=060⇨(8)10).
   ☞n ← mknum scanner next digbit base.
   ☞flag ← false.
   056 = nxtchr⇨
    (scanner skip.
     ☞fs ← scanner next digbit
     0=fs length⇨(☞flag←true. ⇑sign*n)
     ☞n ← n + (mknum fs 10)/10.0 ipow fs length.
     nxtchr=0145⇨(scanner skip. ⇑n*(10.0 ipow rdnum)*sign)
     ⇑n*sign)
   ⇑sign*n)

to mknum str base n i(
   :str.
   :base.
   ☞n ← 0.0.
   for i to str length
    (☞n ← (n*base) + str[i]-060)
   ⇑n)

to rdstr t (scanner skip.
        ☞t←scanner next qtbit.
        scanner skip.
        nxtchr=047⇨(seq←047. ⇑seq contents+rdstr)
        ⇑t)

'↑h75.↑f1.INITIALIZATION OF READ TABLES
↑h60.↑f0.'
☞rtb1 ← vector 256.
☞type ← string 256.
☞sepbit ← 2 * ☞letbit ← 2 * ☞digbit ← 2 * ☞qtbit ← 1.
☞atbits ← letbit + digbit
to scanner n v i j (
        :n. :v. repeat (
        :i.
        (⇦to⇨(:j. for k←i+1 to j+1 (
                type[k]←n. rtb1[k]←v))
        type[i+1]←n. rtb1[i+1]←v)
        ⇦and⇨() done))
scanner 0 ☞(rbuf←scanner next 0) 0 to 0377.
scanner letbit ☞(rbuf←atom scanner next atbits) 0101 to 0132 and 0141 to 0172. 'letters'
scanner digbit ☞(rbuf←rdnum. flag⇨(rbuf←☞.)) 060 to 071. 'digits'
scanner 0 ☞(rbuf←rdnum. flag⇨(rbuf←☞.)) 025. 'high-minus'
scanner sepbit ☞(scanner next sepbit) 011 and 014 and 015 and 040. 'tab, LF, FF, CR, blank'
scanner qtbit ☞(rbuf ← rdstr) 047. 'string-quote'
```

```
scanner 0 ☞(scanner skip. rbuf ← (read1 rtb1) eval) 020.  'eval-paren'
scanner 0 ☞(scanner skip. rbuf ← read1 rtb1) 050.  'left-paren'
scanner 0 ☞(scanner skip. rbuf ← nil. done) 051.  'right-paren'
scanner 0 ☞(rbuf ← nil. done) 0 and 036.  'null and DOIT'
for i to type length (type[i] ← type[i] ⊟ qtbit)
done
☞read ← #junta.
PUT read ☞TITLE ☞read. '↑h75.↑f1.cover our tracks↑h60.↑f0.'

to quit f s t : : r b   (dsoff.
     (null :s⇒()
      ☞f ← file r.
      ☞t ← f intostring.
      f reset.
      f ← s.
      f ← 13.
      f ← t.
      f close).
   file b load)
PUT quit ☞r fill string 7
REM.CM.
PUT quit ☞b fill string 5
BOOT.

to os s : : r b   (☞s ← :.
   file b save⇒
     (quit s + r))
PUT os ☞b fill string 9
BREAK.SV.
PUT os ☞r fill string 17
;RESUME BREAK.SV.

☞fill ← nil

to junta (PUT USER ☞DO ☞(t). CODE 31)
        '↑h75.↑f1.allocates display over OS after setting up t↑h60.↑f0.'
```