```
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 en
**d)

'@2DONT EDIT ABOVE HERE⇑--These classes and atoms mention
**ed
early to guarantee addresses for machine code.

          to isnew (null instance⇒(☞instance←allocate perms
**ize.
                    instance[Ø]←class. ⇑true)
          ⇑false).


@3BOOTSTRAPPING MAGIC.

@2 HEREWITH A SOMEWHAT WHIMSICAL ANNOTATED VERSION OF S'
**DEFS.
ANNOTATIONS ARE IN ITALICS.  WHILE  IT IS HOPED THAT THIS
**
WILL PROVIDE SOME ELUCIDATION OF THE CODE ESCAPES,
OBSCURITIES WILL NO DOUBT PERSIST .  THE ANNOTATIONS ARE
INTENDED TO BE BUT DIMLY LIGHTED MARKERS ON THE ROAD TO
TRUE ILLUMINATION.@1'

to print (◊..)

          '@2:x.Print its address in octal.
          Printing goes to the same place as CODE 2Ø.
```

```
        This is used primarily for bootstrapping.
        All system classes will print themselves.@1'

to read (CODE 2)

        '@2Read keyboard input into a vector.  This is al
**most identical
        in function to the SMALLTALK read routine, except
** that DOIT is
        signalled by <CR> at zero-th parenthesis level, a
**nd single-quote
        strings are ignored.  It is only available in Nov
**a versions.@1'



'@3MESSAGE HANDLING@1'

to : (CODE 18)

        '@2      to : name

        ◊☞ ⇒(:☞name nil ⇒(⇑name←caller message quotefetch
**)
                                        (⇑caller message quotefet
**ch)

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

        ◊# ⇒(:☞name nil ⇒(⇑name←caller message referencef
**etch)
                                        (⇑caller message referenc
**efetch)

        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 evalf
**etch)
                                    ⇑caller message evalfetch)

        Fetch the next thing in the message stream evalua
**ted
        and bind it to the name if one is there.@1'

to ◊ (CODE 17)

        '@2:☞token. token=caller.message.code[caller.mess
**age.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.@
**1'

to ⇑ (CODE 13)

        '@2:x. then do a return, and apply x to any furth
**er message.
        Note that in (... ⇑x+3. ☞y←y-2), the assignment t
**o y will never
        happen, since ⇑ causes a return.@1'

to ☞ (CODE 9)

        '@2⇑:☞. That is, get the next thing in the messag
**e
        stream unevalled and active return it (which
        causes it to be applied to the message).@1'

to # (:#)

        '@2Returns a REFERENCE to its argument?s binding.
**@1'


'@3CONTROL CLASSES@1'
```

```
to repeat token (:#token. CODE 1)

        '@2repeat (token eval) Not a true apply to ☞eval,
        and therefore token MUST be a vector.@1'

to done x (⋖with⇒(:x. CODE 25) CODE 25)

        '@2done causes a pop out of the nearest enclosing
** repeat , for, or do.
        ☞done with val☞ will cause the repeat to have val
**ue val@1'

to again (CODE 6)

        '@2repeat (☞active←active caller. eq active.
        class #repeat⇒(done)). That is, redo the most
        recent repeat, for, or do loop.@1'

to if exp (:exp⇒(⋖then⇒(:exp. ⋖else⇒(:☞. exp)exp)error ☞(
**no then))
                ⋖then⇒(:☞.   ⋖else⇒(:exp) false)error ☞(n
**o then))

        '@2The ALGOL ☞if ... then ... else ...☞@1'

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

        '@2An 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.@1'

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

```
'@3INITIALIZING SYSTEM CLASSES@1'

        '@2Here 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 c
**lass
        class.  The new verson of SMALLTALK will have
        class as a class with these actions intensional.@
**1'

to PUT x y z (:#x. :y. :z. CODE 12)

        '@2The first argument MUST be an atom which is bo
**und
        to a class table.  The third argument is installe
**d
        in the value side of that table corresponding to
**the
        name (atom) which was the second argument.@1'

to GET x y (:#x. :y. CODE 28)

        '@2If ☞x☞ is a class table then the binding of
        the atom in ☞y☞ will be fetched.@1'

to leech field bits : ptr (
        isnew⇒(:ptr)
        CODE 27)

        '@2Lets you subscript any instance
        a[0] gives you the class, a[1] gives the first fi
**eld, etc.
        a[2] gives you the pointer• a[2]□returns the BIT
**S in an integer
        a[2]←foo will dereference count previous contents
**,
        but a[2]⇐foo will not.@1'
```

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

PUT atom ☞DO ☞(CODE 29

        '@2∢←⇒(:x.  ⇑x -- Lookup SELF and replace its val
**ue by x.)
        ∢eval⇒(⇑ -- Lookup the binding of SELF)
        ∢=⇒(⇑SELF=:)
        ∢chars⇒(⇑ -- printname of SELF (a string))@1'

        ∢is⇒(ISIT eval)
        ∢print⇒(disp←SELF chars) )

        '@2Done 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.@1'

to ev (repeat (cr read eval print))

PUT falseclass ☞DO ☞(CODE 11

        '@2∢⇒⇒ (:☞.)
        ∢or⇒ (⇑:)
        ∢and⇒ (:.)
        ∢<⇒ (:.)
        ∢=⇒ (:.)
        ∢>⇒ (:.)@1'

        ∢is⇒(∢false⇒(⇑true) ∢?⇒ (⇑☞false) :☞.)
        ∢print⇒(☞false print) )

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

        '@2isnew⇒(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...@1'

        ◊is⇒(ISIT eval)
        ◊print⇒(disp←40. for x to SELF length
                (disp←32. SELF[x] print). disp←41)
        ◊map⇒(:y. for x to SELF length
                (evapply SELF[x] to y)) )

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

        '@2isnew⇒(Allocate string of length :.
                Fill string with 0377s.)
        ◊[⇒(:x. ◊].
                (◊←⇒(:y. ⇑y -- store y into xth element.
**)
                ⇑   xth element) )
        ◊length⇒(⇑ length of string or vector)@1'

        ◊is⇒(ISIT eval)
        ◊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 l
**ength] print)
        ◊=⇒(:y is string⇒(SELF length=y length⇒(
                for x to SELF length (SELF[x]=y[x]⇒() ⇑fa
**lse)) ⇑false)
                ⇑false)
        ◊+⇒(:y is string⇒(☞x←SELF[1 to SELF length+y leng
**th].
                ⇑x[SELF length+1 to x length]←y[1 to y le
**ngth])
                error ☞(string not found)) )
```

```
PUT number ☞DO ☞(CODE 4

        '@2◊+⇒(⇑val+:)
        ◊-⇒(⇑val-:)
        ◊*⇒(⇑val*:)
        ◊/⇒(⇑val/:)
        ◊<⇒(⇑val<:)
        ◊=⇒(⇑val=:
        ◊>⇒(⇑val>:)
        ◊◘(◊+⇒(⇑val OR :)
                ◊-⇒(⇑val XOR :)
                ◊*⇒(⇑val AND :)
                ◊/⇒(⇑val LSHIFT :))@1'

        ◊is⇒(ISIT eval)
        ◊print⇒(SELF>0⇒(nprint SELF)
                SELF=0⇒(disp←060)
                SELF=0100000⇒(disp←base8 SELF)
                disp←026. nprint 0-SELF) )

        '@2For floating point stuff see FLOAT@1'

to - x (:x*1)

        '@2An often used abbreviation.@
        has to work for float as well.1'

to base8 i x s (:x. ☞s←string 7. for i to 7
        (s[8-i] ← 060 + x ⊗ 7. ☞x ← x ø3). ⇑s)

        '@2Returns a string containing the octal represen
**tation (unsigned)
        of its integer argument.@1'


☞ISIT ← ☞(◊?⇒(⇑TITLE) ⇑TITLE=:☞).

to nil x (#x)

        '@2nil is an ☞unbound pointer☞, which is used
        to fill vectors and tables.@1'
to null x (:x. 1 CODE 37)

        '@2Null returns true if its message is ☞nil☞,
        otherwise false.@1'

to eq x (CODE 15)

        '@2(⇑:x is-identical-to :) - compare 2 SMALLTALK
**pointers.@1'



'@3UTILITIES@1'


to mem x y (:x. CODE 26)
         '@2to mem x y (:x. ◊←⇒(⇑core/mem x ←:)⇑core/mem
**x)
        mem loads integers from and stores them into real
** core.
        Tee hee...
        mem 0430 ← 0  •set alto clock to zero
        mem 0430  •read the clock
        for i to 16 (mem 0430+i ← cursor[i])  •put new bi
**ts into cursor
        mem 0424 ← mem 0425 ← 0.  •reset mouse x and y to
** 0.
        mem 0102 ← 0.  •disconnect cursor from mouse
        mem 0426 ← x. mem 0427 ← y.  •move the cursor
        mem 0100 ← 0177.  •make DEL the interrupt char (i
**nstead of ESC).
        mem 0420.  •get pointer to display control block
        mem 0177034.  •reads the first of 4 keyboard inpu
**t words.@1'

to mouse x (:x. CODE 35)

        '@2 x = 0-7  are a map on the mouse buttons.
        E.g. (4=mouse 4) comes back true if the top mouse
**
```

```
        button is depressed, (1=mouse 1)) comes back true
**
        if bottom mouse button depressed, (7=mouse 7))
        comes back true if all three mouse buttons depres
**sed,
        etc.  Mouse 8 returns the x coordinate of the
        mouse and mouse 9 returns the y coordinate.@1'
to mx (⇑mouse 8)
to my (⇑mouse 9)

to core ((mem Ø77)-mem Ø76)

        '@2Returns the amount of space left in your Small
**talk.@1'
to kbd (Ø CODE 2Ø)

        '@2Waits until a key is struck.
        Returns an ascii code when a key is struck on the
** keyboard.
        Use to kbck (1 CODE 2Ø) to return true if kbd has
** a character,
        otherwise false.
        Used in multiprocessing.@1'

to disp x i (
        ◊←⇒(:x is string⇒(for i to x length (TTY←x[i])) T
**TY←x)
        ◊clear⇒() ◊sub⇒(:x eval))

        '@2This disp is used for bootstrapping.
        Later in these definitions (READER)it will
        be restored to an instance of ☞display frame.☞@1'

to TTY (Ø CODE 2Ø)

        '@2TTY←<integer> will print an ascii  on the Nova
** tty.
        On altos, TTY  prints in little error window at b
**ottom
        of screen.@1'
```

```
to dsoff (mem 272←0)

        '@2Turns display off by storing 0 in display cont
**rol block ptr.
        Speeds up Alto Smalltalk by factor of 2.@1'

to dson (mem 0420 ← 072)

        '@2Turns display back on by refreshing display
        control block pointer.@1'

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)

        '@2Causes its argument to be applied to the messa
**ge 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 t
**he caller is
        not bypassed in any global symbol lookup unless t
**he in-clause
        is used to specify another context.@1'

to cr (disp←13).  to sp (disp←32)

☞true←☞true
☞eval←☞eval
to is ( ◊?⇒(⇑☞untyped):☞. ⇑false)

        '@2These are used to handle messages to classes w
**hich
        can?t answer queston invoking ☞is☞, ☞eval☞, etc.@
**1'

to t nprint substr (ev). t  '@2prevent -to- from making t
**hese global.@1'
```

```
to nprint digit n (:n=0⇒()
        ☞digit←n mod 10. nprint n/10. disp←060+digit)
PUT number ☞nprint #nprint.

        '@2Prints (non-neg) integers in decimal
        with leading zeroes suppressed⇑@1'

to substr op byte s lb ub s2 lb2 ub2 (
        :#s. :lb. :ub. :MESS. ☞CLOB←ub.  '@2tee hee@1'
        :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).  COD
**E 40).
PUT string ☞substr #substr.
PUT vector ☞substr #substr.
done

        '@2substr takes care of copying, moving and searc
**hing
        within strings and vectors.  It first gets its fa
**ther (string/vector)
        and the lower bound, and then proceeds to fetch t
**he 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 i
**dentical.@1'

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)

        '@2Vecmod makes a copy of old vector with ndel el
**ements deleted
        beginning at posn.  If ins is a vector, its eleme
**nts are inserted
        at the same place. It is the heart of edit.@1'

to addto func v w (:#func. :w. ☞v←GET func ☞DO. null v⇒(e
**rror ☞(no code))
        PUT func ☞DO vecmod v v length 0 w)

        '@2Addto appends code to a class definition.@1'


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
    '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])'
    ◊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 obset i input : vec size end (
          ◊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⇒(⇑f
**alse)
                    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⇒(:input. for i ← end to 1 by 1 (input eval)
**)
        ◊print⇒(SELF map ☞(vec[i] print. sp))
        ◊is⇒(ISIT eval)
        isnew⇒(☞end←0. ☞vec←vector ☞size←4)
        )

to { set (☞set←stream of vector 10. repeat(
        ◊}⇒(⇑set contents)
        set ← :)
        )



'@3PRETTY-PRINT@1'

        '@2This prints the code• classprint makes the hea
**der.@1'

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 ☞(: ☞ # ⇑ [ ◊ ⇒ Φ ptr[i-1]⇒(disp
**←32. t print)
                t print)
```

```
          disp←41)
to t tabin index (ev)
t
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. COD
**E 40)
          '@2A piece of substr which runs faster.@1'
PUT pshow ☞index #index.
done

'@3FLOATING POINT@1'

PUT float ☞DO ☞(0 CODE 42
         ◊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)
        ◊print⇒
           (SELF = 0.0⇒(disp ← 48. disp←46. disp←48)
            SELF < 0.0⇒
              (disp ← 22.
               fprint - SELF)
            fprint SELF)
          )
to t fprint (ev)
t
to fprint n i p q s : : fuzz (
    'Normalize to [1,10)'
      (:n < 1⇒
         (☞p ← -(10.0 / n)
          epart 10.0)
       ☞p ← n epart 10.0)
       ☞n ← fuzz + n / 10.0 ipow p.
    'Scientific or decimal'
      (☞q ← p.
       ☞s ← fuzz*2.
       p > 6⇒
         (☞p ← 0)
       p < 3⇒
         (☞p ← 0)
       ☞q ← 0.
       p < 0⇒
         (disp ← 48. disp←46.
          for i ← p to 2(disp ← 48))
       ☞s ← s * 10.0 ipow p)
    'Now print (s suppresses trailing zeros)'
    for i to 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 float ☞fprint #fprint.
done

'@3TEXT DISPLAY ROUTINES@1'

        '@2Display frames are declared with five paramete
**rs.
        They are a left x, a width, a top y, a height, an
**d 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 256 bits(raster units) wide and 256
**bits high.
        The string (buf) serves as the text buffer, and i
**s altered
        by ← and scrolling.

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

        The four instance variables defining the window a
**re
        ☞winx☞, ☞winwd☞, ☞winy☞, and ☞winht☞.  The
        boundaries of this rectangle are intersected with
        the physical display.  The window actually used b
**y
        the machine language will reduce the size of the
        window, if necessary, to be confined by the physi
**cal
        display.  Clipping and scrolling are done on the
**basis
        of window boundaries.  If a character is in the w
**indow
```

it will be displayed. If a string or character c
**ause
overflow of the bottom of the window, scrolling w
**ill
occur.

The four instance variables defining the frame ar
**e
☞frmx☞, ☞frmwd☞, ☞frmy☞, and ☞frmht☞. This rect
**angle
may be smaller or larger than its associated wind
**ow
as well as the physical display. Frame boundari
**es
are the basis for word-wraparound. (Presently, i
**f frmy+
frmht will cause overflow of the window bottom[wi
**nx+winht],
frmht will get changed to a height consonant with
** the
bottom of the window. This has been done to mana
**ge
scrolling, but may get changed 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 string
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 curr
**ent
last character in the buffer. That is, the last
character resulting from a ☞←☞.

```
        ☞Lstln☞ also points into the buffer at the charac
**ter
        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 intersection
        between the window and the display.
        A 2 reply means no intersection between window an
**d frame.
        A 3 reply means window height less than font heig
**ht --
        hence no room for scan conversion of even one lin
**e of text.
        A 4 means that the frame height has been increase
**d
        in order to accomodate the input.
        A 5 means the bottom of the window (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 and
        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 ← fontstring ?somefontfile?.
        Then you can say disp's (☞font←something) or
        you can declare the font at the same time as the
        dispframe is declared as e.g.
         ☞yourframe ← dispframe 3 40 3 40 string 20 font
**something.
        @1'


(to dispframe input
        : winx winwd winy winht frmx frmwd frmy frmht
        last mark lstln charx chary reply justify buf fon
**t editor
        : sub frame dread reread (
◊ ← ⇒(0 CODE 51)

        '@2:s. s is number ⇒ (append this ascii char)
                s is string ⇒(append string)
                error.@1'

◊'s ⇒(⇑(:☞.)eval)

        '@2Allows access to instance variables. For examp
**le,
                yourframe 's (☞winx←32)
        will alter the value of window x in the
        instance of dispframe called ☞yourframe☞.@1'

◊show⇒(4 CODE 51 3 CODE 51)

◊display⇒(SELF show. frame black)

        '@2Show 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.@1'

◊hasmouse⇒(frmx<mx<frmx+frmwd⇒(⇑frmy<my<frmy+frmht)⇑false
**)

        '@2Tells you if the mouse is within a frame.@1'

◊fclear⇒(4 CODE 51)

        '@2Fclear clears the intersection of the window a
**nd frame.
        Hence if the frame is defined as smaller than the
** window,
        only the frame area will be cleared.  If the fram
**e is
        defined as larger than the window, only the windo
**w area
        will be cleared, since that space is in fact
        your ☞window☞ on that frame.@1'

◊put⇒(:input. ◊at. ☞winx←☞frmx←:. ☞winy←☞frmy←☞chary←:.
        ☞last←0. ☞lstln←1. SELF←input. ⇑charx-winx)

        '@2For them as would rather do it themselves.@1'

◊wclear⇒(5 CODE 51)

        '@2Wclear clears the intersection of a  window
        and the physical display.@1'

◊scroll⇒(2 CODE 51)

        '@2Scroll removes the top line of text from the f
**rame?s
        string buffer, and moves the text up one line.@1'

◊clear⇒(1 CODE 51)
```

```
        '@2Clear does an fclear and sets the ☞last☞ point
**er
        into the string buffer to 0 and ☞lstln☞ to 1.
        It has the effect of cleaning out the string buff
**er
        as well as clearing the frame area.@1'

◁mfindc ⇒(7 CODE 51)

        '@2 Find character.
        Takes two arguments -- x and y (typically msex an
**d 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 strin
**g.
        If vec[2] is -2 x,y is not in the window.
        Sample call:
                ☞myvec←yourframe mfindc mouse 8 mouse 9.@
**1'


◁mfindw ⇒(8 CODE 51)

        '@2 Find word.
        Takes two arguments -- x and y (typically msex an
**d 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 strin
**g.
        If vec[2] is -2 x,y is not in the window.
        Sample call:
                ☞myvec←yourframe mfindw mouse 8 mouse 9.@
**1'
```

```
◊mfindt ⇒(6 CODE 51)

        '@2 Find token.
        Takes two arguments -- x and y (typically msex an
**d msey).
        Returns vector:
                vec[1] = token count, ala Smalltalk token
                         Spaces and carriage returns are
                         considered as delimiters,but mul
**tiple
                         delimiters do not bump the count
**.
                         Text delimited by single quotes
**is
                         counted as one token, and embedd
**ed
                         text (i.e. more than one quote i
**n
                         sequence will not cause the toke
**n
                         count to be bumped (allows for e
**mbedding
                         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 strin
**g.
        If vec[2] is -2 x,y is not in the window.
        A sample call--
                ☞variable←yourframe mfindt mouse 8 mouse
**9.@1'

◊read⇒(⇑dread)

        '@2Makes a code vector out of keyboard input.
        See dread below.@1'
```

```
◊reread⇒(⇑reread :)

        '@2Used by redo and fix.  Goes back n(its argumen
**t),
        prompts and does a read from there.
        See reread below.  @1'

◊sub⇒(☞input ← sub :. SELF show. ⇑input)

        '@2Evals its argument in a sub-window.  Used by f
**ix and
        shift-esc.  See sub below.@1'

◊knows⇒(ev)

        '@2Whilst 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.  @1'

◊frame ⇒ (apply frame)

        '@2Draws a border of the given color
        around the frame.  E.g.,
                yourframe frame -1.@1'

◊is ⇒(ISIT eval)

isnew ⇒ (☞winx←:frmx. ☞winwd←:frmwd. ☞chary←☞winy←:frmy.
        :frmht.☞winht←682-winy. :buf. ☞lstln←1.
        ☞mark←☞last←☞charx←☞reply←☞justify←Ø.
        (◊font⇒(:font)) ◊noframe⇒() frame black)  ))

dispframe knows
to dread t flag (
        disp←2Ø. ☞flag←false. ☞mark←last.
        (null #DRIBBLE⇒() DRIBBLE flush)
        repeat (Ø5Ø>disp←☞t←kbd⇒(
                t=Ø1Ø⇒(last<mark⇒(disp←buf[last+1])
                '@2Backspace only up to prompt.@1'
                        buf[last+1]=047⇒(☞flag←flag is fa
**lse))
                        '@2Backspace out of string flips
**flag.@1'
                t=012⇒(flag⇒() done)
                '@2DOIT checks if in a string.@1'
                t=047⇒(☞flag←flag is false)
                '@2Flag is true if in a string@1'
                t=023⇒(sub ☞(ev). ☞last←last-1. disp show
**)
                '@2Shift-Esc make sub-eval.@1'
                t=027⇒(disp←010. ☞done print. disp←012. ⇑
**☞(done))
                ))
        disp←13. ⇑read of stream of buf from mark+1 to la
**st)
to sub disp (
        ☞disp←dispframe winx+48 winwd-64 winy+14 winht-28
** string 300.
        disp clear. (:)eval)

        '@2Opens a sub-frame, and evals its argument
        in that context.@1'

to frame a (☞a ← turtle at frmx - 1 frmy - 1.
        a's width ← 2 . a's ink ← (◊white⇒(0) ◊black. 1)
        do 2 (a turn 90 go frmwd + 2 turn 90 go frmht + 2
**)     )

        '@2Draws a double line around the frame.@1'


to reread n i p reader (
        ☞p←mark. 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)

        '@2Counts back n prompts (n is integer arg)
        and then does a read from there.@1'

done

to dclear (CODE 52)

        '@2This 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. @1'


to dcomp (CODE 53)

        '@2Just like dclear only complement rectangle.@1'


to dmove (CODE 54)

        '@2This function takes six parameters -- source x
** width
        source y height destination x destination y.  It
**takes the
        source rectangle (x and width mod 16?d as in dcle
**ar) and
        moves it to the destination x and y. Clipping wil
**l occur on
        display boundaries. The source will remain intact
** unless
        it overlaps with the destination, in which case t
**he over-
        lapping portion of the destination wins.@1'

to dmovec (CODE 55)

        '@2Dmovec takes the same parameters as dmove, but
        in addition clears the non-intersecting source ma
**terial.
        It is the general case of what happens on the dis
**play
```

```
        screen during a scroll, i.e. scrolling could
        be accomplished by saying
        disp's (dmovec winx winwd winy+fontheight
        winht-fontheight winx winy).
        A sample call --
        dmovec 0 256 0 256 256 256.  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. @1'


to redo (disp 's  (☞last←mark-2). (disp reread :) eval. dis
**p show.)

        '@2Causes re-evaluation of the input typed n prom
**pts
        before this.  Setting last←mark-2 makes the redo
        statement and its prompt disappear with a disp sh
**ow.@1'

to fix vec (disp 's  (☞last←mark-2). ☞vec←disp reread :.
        (disp sub ☞(veced vec)) eval)

        '@2Like redo, except that the previous input is g
**iven
        to the editor in a subwindow.  When editing is do
**ne,
        the resulting code is evalled before returning.@1
**'

to fontstring f s (☞f←file (:) old. ☞s ← string (2*03154)
**. f next into s. f close. ⇑s)

        '@2Gets a font into core.  Getting it into a stri
**ng
        is simple expedient for grabbing contiguous core.
**@1'



'@3TURTLES@1'
```

```
to turtle var : pen ink width dir x xf y yf frame : f (
        CODE 21  '◊go⇒(draw a line of length :)
                  ◊turn⇒(turn right : (degrees))
                  ◊goto⇒(draw a line to :(x), :(y))'
        ◊'s ⇒(:☞var. ◊←⇒(⇑var ← :)
                     ⇑var eval)
        ◊pendn⇒(☞pen ← 1. ⇑SELF)
        ◊penup⇒(☞pen ← 0. ⇑SELF)
        ◊black⇒(☞ink ← 1. ⇑SELF)
        ◊white⇒(☞ink ← 0. ⇑SELF)
        ◊xor⇒(☞ink ← 2. ⇑SELF)
        ◊is⇒(ISIT eval)
        ◊home⇒(☞x ← frame 's  (frmx+frmwd/2).
               ☞y ← frame 's  (frmy+frmht/2).
               ☞xf ← ☞yf ← 0. ☞dir←270. ⇑SELF)
        ◊erase⇒(frame fclear. ⇑SELF)
        ◊up⇒(☞dir ← 270. ⇑SELF)
        isnew⇒(☞pen ← ☞ink ← ☞width ← 1.
                (◊frame⇒(☞frame ← :) ☞frame ← f)
                ◊at⇒(:x. :y. ☞xf ← ☞yf ← 0. ☞dir←270)
                SELF home)
        )
```

```
  '@3THE TRUTH ABOUT FILES

@2FILESMALL: Smalltalk file and directory definitions

also see <SMALLTALK> on Maxc for:
copym, dskstat, install, purge, type, xplot, undribble


a file is found in a directory (☞dirinst☞) by its file na
**me (☞fname☞),
and has a one ☞page☞, 512 character string (☞sadr☞).  ☞rv
**ec☞ is
an optional vector of disk addresses used for random page
** access.

@1☞fi ←@2 <directory> file <string> old   finds an old fi
```

```
**le named <string>
in <directory> or returns false if does not exist or a di
**sk error occurs.

@1☞fi ←@2 <directory> file <string> new   creates a new f
**ile or returns
false if it already exists.  if neither old or new is spe
**cified,
an existing file named <string> will be found or a new fi
**le created.
if <directory> is not specified, the current default dire
**ctory is used.

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

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

<directory> file <string> load   loads a previously ☞save
**d☞ Smalltalk
virtual memory, thereby destroying your current state.

<directory> file <string> save   saves Smalltalk virtual
**memory.

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

☞dirty☞ = 1 if any label block integers (☞nextp☞ thru ☞sn
**2☞) have
```

been changed• = -1 if ☞sadr☞ has been changed• = 0 if the
current page is clean. the user need not worry about thi
**s unless
(s)he deals directly with the label or ☞sadr☞. it might b
**e
noted here that multiple instances 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 language pointer to offending di
**sk command
block (dcb)) signals a disk error.

the next 8 integers are the alto disk label block. ☞next
**p☞ and
☞backp☞ are the forward and backward alto address pointer
**s. ☞lnused☞
is currently unused. ☞numch☞ is number of characters on t
**he current
page, numch must be 512, except on the last page. ☞pagen☞
** is the
current page number. page numbers are non-negative intege
**rs, and the
format demands that the difference in consecutive page nu
**mbers is 1.
normal file access starts at page 1, although all files p
**ossess page 0
(the ☞leader☞ page). ☞version☞ numbers > 1 are not implem
**ented. ☞sn1☞
and ☞sn2☞ are the unique 2-word serial number for the fil
**e.

the class function ☞ncheck☞ checks that file names contai
**n
alphabetic or ☞legal☞ characters or digits, and end with
**a period.@1'

(to file : dirinst fname sadr rvec leader curadr bytec di
**rty status nextp

```
        backp lnused numch pagen version sn1 sn2 : ncheck
** x (

        ◊←⇒     (17 CODE 5Ø)

                '@2fi←<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) by
**tec mod 512))
                sadr[bytec]←x ⊗ Ø377@1'

        ◊next⇒  ((◊word⇒ (◊←⇒ (7)

                        '@2fi next word←<integer> -- writ
**e integer.
                        possibly increment pointer to wor
**d boundary.
                        (Ø=bytec ⊗ 1⇒ () ☞bytec←bytec+1)
                        SELF ← :x/256. SELF ← x mod 256.@
**1'

                         6)

                        '@2fi next word -- read an intege
**r
                        (Ø=bytec ⊗ 1⇒ () ☞bytec←bytec+1)
                        ⇑(SELF next*256) + SELF next@1'

                  ◊into⇒ (16)

                        '@2fi next into <string> -- read
**a string
                        for i to :x length(x[i]←SELF next
**).⇑x@1'

                  25) CODE 5Ø)
```

```
                                         '@2fi next -- read a character
                                         (numch<☞bytec←bytec+1⇒
                                          (SELF set to read (pagen+bytec/5
**12)
                                            bytec mod 512⇒ () ⇑Ø))  ⇑sadr[
**bytec]@1'

          ◇set⇒    (◇to. (◇end⇒(13)

                                         '@2fi set to end -- set file poin
**ter to end
                                         of file.  SELF set to read Ø37777
** Ø@1'

                                ◇write⇒(5)

                                         '@2fi set to write <integer> <int
**eger> -- 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 a
**fter end if
                                         necessary (-1 512 is treated as s
**tart of next
                                         page, i.e. pagen+1 Ø). ☞bytec←sch
**ar@1'

                               ◇read. 4) CODE 5Ø)

                                         '@2same as ☞write☞ except stop at
** end@1'

          ◇skipnext⇒ (18 CODE 5Ø)

                               '@2fi skipnext <integer> -- set character
** pointer
                               relative to current position. (useful for
** skipping
                    rather than reading, or for reading and b
**acking up,
                    but ☞end☞ may not work if ☞bytec☞ points
**off the current
                    page)  ☞bytec ← bytec + :.@1'

          ⋖end⇒     (1Ø CODE 5Ø)

                    '@2fi end -- return false if end of file
**has not
                    occurred.  nextp=Ø⇒ (bytec<numch⇒ (⇑false
**))⇑false@1'

          ⋖'s ⇒     (⇑(:☞) eval)

          ⋖flush⇒   (12 CODE 5Ø)

                    '@2fi flush -- dirty=Ø⇒ () write current
**page@1'

          ⋖writeseq⇒ (22 CODE 5Ø)

                    '@2transfer words from memory to a file
                    :adr. :count. for i←adr to adr+count-1
                    (SELF next word ← mem i)@1'

          ⋖readseq⇒          (21 CODE 5Ø)

                    '@2...from a file to memory...(mem i ← SE
**LF next word)@1'

          ⋖is⇒      (ISIT eval)

          ⋖remove⇒ (dirinst forget SELF)

                    '@2remove file from filesopen list of dir
**ectory@1'

          ⋖close⇒   (dirinst 's  (bitinst flush).
                     SELF flush. SELF remove. ⇑☞closed)
```

```
                        '@2fi close or ☞fi←fi close (if fi is glo
**bal) --
                        flush bittable and current page, remove i
**nstance
                        from filesopen list of directory@1'

        ◊shorten⇒ (◊to. ◊here⇒ (SELF shorten pagen bytec)
** 14 CODE 50)

                        '@2fi shorten to <integer> <integer> -- s
**horten a file
                        SELF set to read :spage :schar. ☞x←nextp.
** ☞nextp←0.
                        ☞numch←schar. ☞dirty←1. deallocate x and
**successors@1'

        ◊print⇒  (disp ← fname)    '@2file prints its name@1
**'

        ◊reset⇒ (11 CODE 50)

                        '@2fi reset -- reposition to beginning of
** file
                        SELF set 1 0@1'

        ◊random⇒ (SELF set to end. ☞rvec ← vector pagen.
                          for x to rvec length (
                           SELF set x 0. rvec[x] ← curadr))

                        '@2fi random -- initialize a random acces
**s vector
                        to be used in fi set...  new pages append
**ed to the
                        file will not be randomly accessed@1'

        ◊pages⇒  (20 CODE 50)

                        '@2fi pages <integer> ... <integer> -- ou
**t of the same
                        great tradition as ☞mem☞ comes the power
```

```
**to do
                    potentially catastrophic direct disk i/o
**(not for the
                    faint-hearted).  :coreaddress. :diskaddre
**ss. :diskcommand.
                    :startpage. :numberofpages. :coreincremen
**t. if -1 =
                    coreaddress, copy ☞sadr☞ to a buffer befo
**re the i/o call.
                    diskaddress (=-1 yields ☞curadr☞) and dis
**kcommand are
                    the alto disk address and command. startp
**age is relevant
                    if label checking is performed. numberofp
**ages is the
                    number of disk pages to process. coreincr
**ement is usually
                    0 (for writing in same buffer) or 256 for
** using
                    consecutive pages of core.  use label blo
**ck from instance
                    of ☞fi☞. copy label block from instance.
**perform i/o call.
                    copy ☞curadr☞ and label block into instan
**ce. if -1=coreaddress
                    copy buffer to ☞sadr☞.@1'

          isnew⇒  (☞fname←ncheck :. fname is false⇒
                              (error ☞(bad file name)⇑nil)
                     (null ☞dirinst←#curdir⇒ (
                      (☞dirinst←directory 's  (defdir)) is dire
**ctory⇒
                        (dirinst open) error ☞(illegal directo
**ry)))

                              '@2set directory instance for fil
**e. if curdir
                              is not a directory (null global v
**alue because
                              file was not called from the cont
**ext of a
```

```
                              directory instance), use the defa
**ult directory@1'

                       ◊exists⇒ (24 CODE 50. ⇑fname)

                              '@2return false if file name does
** not
                              occur in the directory@1'

                       ◊delete⇒ (15 CODE 50. ⇑☞deleted)

                              '@2delete a file (see intro)@1'

                       ☞sadr ← (◊using⇒ (:) string 512).

                              '@2set up file string buffer@1'

                       ◊rename⇒ (☞x ← ncheck :. x is false⇒
                                        (error ☞(bad new name)⇑ni
**l)
                                file x exists⇒ (error ☞(name al
**ready in use))
                                2 CODE 50. ☞fname ← x. 23 CODE
**50.
                                SELF set 0 12. SELF ← fname len
**gth.
                                SELF ← fname. SELF flush. ⇑fnam
**e)

                              '@2check 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@1'

                        ◊load⇒ (2 CODE 50. 8 CODE 50)

                       (◊old⇒ (2)
                        sadr[13] ← fname length.
                        sadr[14 to 13+fname length] ← fname.
```

```
                      <new⇒ (dirinst 's  (filinst) is file⇒ (3)
**19)
                      1) CODE 50.
                            '@2find an old file or add a new
**entry (with
                            its name as a BCPL string in its
**leader page.
                            special handling for new director
**ies). machine
                            code may return false@1'
                     <save⇒  (9 CODE 50.
                            dp0 's  filesopen map ☞(vec[i] 's  (☞
**dirinst←nil)).
                             directory 's  (☞curdir ← 0.
                              ☞defdir ← ☞dp0 ← directory dirn
**ame)disp show)
                            '@2load returns via ☞save☞. virtu
**al memory on
                            file should have no active files
**or directories.
                            dp0 is reinitialized upon load. h
**ow to reopen
                            other files (e.g. DRIBBLE)⇒@1'
                     dirinst remember SELF) ))
                            '@2finally, file puts itself into
** the
                            filesopen list of its directory@1
**'
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)
'@2check that the file name is a proper length string con
**taining only
lower/upper case letters, digits, or legal characters. if
** name does
not end with a period, append one.@1'

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


'@2a directory is found in a directory (☞dirinst☞), has a
** bittable file
(☞bitinst☞) for allocating new pages, a file of file entr
**ies (☞filinst☞
-- file names, disk addresses etc.), and a list of curren
**tly open files
(☞filesopen☞ which is an ☞obset☞). the top level, ☞distin
**guished node☞
of the directory structure is the system directory ☞dp0☞
**(see ☞directory
knows☞ below if you also want ☞dp1☞). dp0 knows the disk
**number (☞dirinst☞)
and the true identity of the bittable. each file must ask
** its directory
for the bittable when page allocation is necessary, and t
**he system
directory (via its local directory) for the disk number.

@1☞di ←@2 <directory> directory <string> old/new

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

☞dirname☞ is the system directory name and ☞bitname☞ is t
**he bittable name.
☞curdir☞ is a class variable bound to the last directory
**instance ☞opened☞,
and provides information ☞who called you☞ (i.e. CALLER) t
```

```
**o a file or
directory. ☞defdir☞ is a default directory, initially set
** to dpØ, which
is invoked when ☞curdir☞ fails to be a directory, i.e. fi
**le was not
called in the context of a directory, but globally@1'

(to directory name exp : dirinst bitinst filinst filescope
**n : dirname bitname
        curdir defdir (

        ◊file⇒   (SELF open. ⇑apply file)

                '@2di file <string>... -- open directory.
** create file
                instance (see file intro)@1'

        ◊directory⇒ (SELF open. ⇑apply directory)

                '@2di directory <string>... -- open direc
**tory. create
                directory instance@1'

        ◊open⇒  (☞curdir←SELF. filinst is file⇒ ()
                 (bitinst>Ø⇒        (☞bitinst ← dirinst 's  (bi
**tinst).
                                     ☞filinst ← file filinst
**new)
                 ☞filinst ← file filinst old.
                 ☞bitinst ← (dirinst is directory⇒
                        (dirinst 's  (bitinst)) file bitnam
**e old)).
                dirinst is directory⇒ (dirinst remember S
**ELF))

                        '@2di open -- (normally not user-
**called since access
                        to the directory always reopens i
**t) initialize
                        directory file and bittable insta
**nces. directory
```

```
                            (except for ☞top level☞) puts its
**elf into filesopen
                            list of its directory@1'

       ◊is⇒     (ISIT eval)

       ◊print⇒  (disp←Ø133. filesopen print. disp←Ø135)

                            '@2di or di print. --print the fi
**lesopen list@1'

       ◊map⇒    (SELF open. ☞exp←:. filinst reset.
                repeat (filinst end⇒ (cr. done)
                 1Ø24 > ☞name← filinst next word⇒
                            (name < 2⇒ () filinst skipnext 2*
**name-1)
                filinst skipnext 1Ø.
                ☞name ← filinst next into string filinst
**next.
                exp eval))

                            '@2di map expression -- evaluate
**an
                            expression for each file name@1'

       ◊list⇒   (SELF map ☞(disp←name. sp))

                            '@2di list -- print the entry nam
**es contained in filinst@1'

       ◊remember⇒ (filesopen ← :)

       ◊forget⇒ (filesopen delete :)

                            '@2...add or delete file instance
**s in filesopen@1'

       ◊close⇒  ((filinst is file⇒ (filesopen map ☞(vec[e
**nd] close).
                     (dirinst is directory⇒ (dirinst forget
**SELF)).
```

```
                  ☞filinst ← filinst 's (fname).
                  bitinst flush. ☞bitinst ← 1)). ⇑☞close
**d)

                        '@2di close (e.g. dp0 close) or ☞
**di←di close
                        (to release instance) --close a d
**irectory by
                        closing all files and directories
** in its
                        filesopen list and deleting it fr
**om the
                        filesopen list of its directory.
**this is
                        currently one way to regain space
** by closing
                        unwanted file instances, and to c
**hange disk packs@1'

        ◊use⇒   (☞defdir ← SELF)

                        '@2di use -- change the default d
**irectory@1'

        ◊'s ⇒   (⇑(:☞) eval)

        isnew⇒  (☞filesopen ← obset. ☞dirinst ← curdir.
                 ☞filinst ← :.
                 dirname = filinst⇒ (☞bitinst ← 1. ☞curd
**ir ← SELF)

                        '@2store the directory file name
**in filinst
                        and flag old/new in bitinst.  sys
**tem
                        directories are not opened@1'

                 ☞bitinst ← (◊new⇒ (1) ◊old. 1). SELF op
**en)))

directory 's (ev)
```

```
☞dirname ← fill string 7
SYSDIR.
☞bitname ← fill string 9
SYS.STAT.

        '@2names of the system directory and bittable@1'

☞curdir←0. ☞defdir←☞dp0←directory dirname.

        '@2create the system directory instance (the init
**ial default)
        on disk 0 in a ☞closed☞ state. to initialize a se
**cond disk:

        @1directory 's  (☞curdir ← 1. ☞dp1 ← directory dirn
**ame)'
done

☞curdir ← nil.              '@2so default directories will wo
**rk@1'


to error adr ptr arec class :: c shocode find 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 cpc (
        null arec[5]⇒(.) ☞arec←leech arec[5]. ☞class←arec
**[0].
        (GET class ☞TITLE) print. ☞: print.
        arec[6] is vector⇒(find arec[1]□arec[6] ⇒ (shoco
**de))
        find arec[1]□GET class ☞DO ⇒ (shocode).
        )
to shocode i (
        for i←1 to code length
                (i<cpc-5⇒(disp←056) i>cpc+5⇒(disp←056)
                sp. (i=cpc⇒(disp←1))
                code[i] is vector⇒(☞[] print) code[i] prin
**t).
        )
to find adr vec vadr l ( 'a tree search in vec for the ad
**dress adr'
        ☞adr←:. ☞l←leech :vec.
        vec is vector is false⇒(⇑false)
        ☞vadr←(leech l)[1]□+1.
        (adr>vadr⇒(adr<vadr+vec length+1⇒
                (☞cpc ← adr-vadr. ☞l←0. ☞code←vec. ⇑true)
**))
        ☞l←0. for l to vec length
                (vec[l] is vector⇒(find adr vec[l]⇒(⇑true
**)))
        ⇑false)
to sub disp (☞disp ← GET USER ☞disp. (:) eval)
done

to kbck (1 CODE 20)
        '@2Returns true if the keyboard has been hit@1'

to button n (⇑:n=mouse 7)
        '@2Returns true if that pattern is being held dow
**n@1'


        '@3THE SMALLTALK EDITOR - -@1'


to edit func t (:#func.
        ☞t←GET func ☞DO.
        null t ⇒ (⇑☞(no code))
        ⋖title⇒ ((veced classprint func header) eval)
        PUT func ☞DO veced t.
        ⇑☞edited)

        '@2Edit picks up a code vector, makes sure it is
**not empty
```

```
        and calls veced to edit the code body.  If you sa
**y edit foo title,
        veced will edit the header as well, and the chang
**ed form will be
        evalled upon exit to redefine the function, title
** and all.

        Veced can be used on any vector, and is used by F
**IX as well
        as EDIT.  It creates two new windows within the d
**efault DISP
        which exists when it is called.  One is used for
**a menu of
        commands, 'the other becomes the new default wind
**ow DISP.
        The new default is passed to an intermediary• and
** the newly
        edited vector is returned.@1'

(to veced back newdisp  menu x ::  menuwidth menulen menu
**str
ed edpush edtarget gettwo bugin getvec  (
        ◊knows⇒(ev)
        ☞back←false.
        disp fclear.
        disp 's  (☞menu←dispframe winx+winwd-menuwidth menu
**width
                        winy (winht>139⇒(winht) 14Ø) menu
**str.
                menu 's  (☞last ← menustr length).
                mem Ø425 ← winy + 1Ø3.
                ☞newdisp ← dispframe winx winwd-menuwidth
**+2
                        winy winht string buf length nofr
**ame)
        :x. ☞x ← indisp newdisp (ed x).
        disp show.
        ⇑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 indisp disp (:disp. ⇑(:☞)eval)
        '@2used to make DISP a new local.@1'

to ed ptr l n nrun command temp i nv n1 fnth hfnth   (
        ☞command ← 0.
        :ptr.
        ☞fnth ← mem ((mem 70)-2).
        ☞hfnth ← fnth/2.
        repeat(
                ☞l←ptr length.
                back⇒(done with ptr)
                mem 0424 ← menu 's  (winx + winwd/2).
                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) b
**oth).
                mem 0424 ← disp 's  (winx + winwd/2).

                ☞(
                        (☞ptr←vecmod ptr 1 Ø read)
                        (☞ptr←vecmod ptr (edcomp edtarget both) Ø
** read)
                        (gettwo.  ☞ptr←vecmod ptr n nrum read)
                        (gettwo. ☞ptr←vecmod ptr n nrum nil)
                        (gettwo. ☞temp ← ptr[n to n+nrum]
                                temp[nrum + 1] ← nil.
                                ☞i←(edcomp edtarget both).
                                ☞ptr←vecmod ptr n nrum nil.
                                (i>n ⇒ (☞i←i-nrum))
                                ☞ptr←vecmod ptr i Ø temp)
                        (getvec⇒(☞ptr←vecmod ptr n 1 ptr[n]) agai
**n)
                        (gettwo. edpush)
                        (getvec⇒(ptr[n]←ed ptr[n]) again)
                        (done with ptr)
                        (☞back←true. done with ptr)
                                ) [command] eval.
                        )
                )
                '@2The 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.@1'


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

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

to bugin  someframe  max index(
        :someframe.
        ☞max ← 1+:.
        repeat (button 0 ⇒ (repeat (
                                button 7 ⇒(disp sub ☞(ev))
                                button 0 ⇒()
                                done)
                done)
                )
        ☞index←someframe mfindt mouse 8 mouse 9
        0<index[1]< max ⇒
                (⇑index)
  'returns token index, if within range, else'
        again
  'causes an exit out of this command by restarting ed?s
**repeat'
        )

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

'@3BOOTSTRAPPING REVISITED@1'

to classprint fn a b i j k flags clsv clsm arecv arecm in
**stv instm code (
        :#fn. ☞code ← GET fn ☞DO. null code⇒(☞(no code))
        ☞a←leech #fn. ☞b←vector 1. ☞b←leech b.  ☞clsm←☞ar
**ecm←☞instm←0.
        ☞k←a[1]□ ☞clsv←vector k. ☞arecv←vector k. ☞instv
***←vector k.
```

```
        '@2Pull symbols out of class table@1'
        for i←4 to 4+2*k by 2   '@2k is no. dbl entries -
**1, here@1'
                (☞k←a[i]◘
                k=1⇒(again).  ☞flags ← k∅14. '@20=clas
**s, 2=arec, 3=inst@1'
                flags=0⇒(0=☞(DO TITLE SIZE) [1 to 3] find
** a[i]⇒
                        (clsv[☞clsm←clsm+1] ← a[i]))
                b[2]□← k⊗03777.  ☞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⇒(☞in
**stm←j+1))
                )

        '@2Now make up input form.@1'
        ☞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)
                                null shVal⇒(☞nil print))
                        shVal print. ☞. print)
                classprint shVal))
        shAtom print)
        disp←10.)

'@2Keyboard translation@1'

to kbd (⇑kmap[TTY])
☞kmap ← string 0377.
        for i←001 to 0177(kmap[i]←kmap[0200+i] ← i)
        kmap[0200]←kmap[0233]← 040. '@2ctl null and esc@1
**'
        to t i (kmap[:i]←kmap[0200+i]←:)
        t 0020 0010        '@2SHIFT BS@1'
        t 0021 0011        '@2SHIFT TAB@1'
        t 0037 0040        '@2SHIFT UP@1'
        t 0036 0040        '@2UP@1'
        t 0035 0040        '@2SHIFT DOWN@1'
        t 0034 0040        '@2DOWN@1'
        t 0030 0040        '@2SHIFT SPACE@1'
        t 0025 0015        '@2SHIFT RETURN@1'
        t 0027 0010        '@2SHIFT DEL->BS@1'
        t 0032 0040        '@2SHIFT INS@1'
        t 0031 0040        '@2INS@1'
        t 0022 0012        '@2SHIFT LF@1'
        kmap[0260] ← 0034          'ctl-0'
        kmap[0261] ← 0001          'ctl-1'
        kmap[0262] ← 0002
        kmap[0263] ← 0003
        kmap[0264] ← 0004
        kmap[0265] ← 0032
        kmap[0266] ← 0006
        kmap[0267] ← 0007
        kmap[0270] ← 0024          'ctl-8'
        kmap[0271] ← 0011
        kmap[0140] ← 0026          'shift-minus'
        kmap[0255] ← 0140          'ctl-minus'
        kmap[0344] ← 0027          'ctl-d'
        kmap[0351] ← 0030          'ctl-i'
        kmap[0271] ← 0005          'ctl-('
        kmap[0136] ← 0031          ''s moved in font'
        kmap[0363] ← 0136          'ctl-s'
```

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

        '@2Filout basically does a show in an environment
        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 n
**eater but takes
        longer to generate. if ☞add☞ is used, function de
**finitions are
        appended to the file. if <vector> is not specifie
**d, ☞defs☞ is used.@1'

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

        '@2Filin basically does a read-eval-print loop, b
**ut
        gets its input from a file instead of a dispframe
**.@1'

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 fname ::x y z (4knows⇒(ev)
        PUT turtle ☞f dispframe 0 512 0 684 string 1 nofr
**ame.
        ☞disp←dispframe 16 480 514 168 string 520.
        ☞☺ ← turtle frame dispframe 0 512 0 512 string 1
**noframe.
        disp←x. ☞defs ← obset.
        ☞fname ← read[1].
        (null fname⇒()
                PUT kbd ☞DO ☞(⇑DRIBBLE ← translation[TTY]
**)
                ☞DRIBBLE←file fname chars+y.
                DRIBBLE set to end. do 20 (DRIBBLE ← 052)
**)
        disp←z. ☞fool←#to. to to toAtm (CODE 19 defs←toAt
**m. toAtm)
        PUT USER ☞DO ☞(cr read eval print). ☞t←0.)
t knows
☞x←fill string 36
[ 2/9 ] Hi, please type your name...
☞y←fill string 6
.drib.
☞z←fill string 20
Welcome to SMALLTALK
done
to expand x (:x. disp 's (☞winy←☞frmy←winy-x. frame black)
        disp show CODE 38)

        '@2t is called to set up a display frame, and def
**s.
        It also sets up a dribble file (if a name is give
**n),
        and then self-destructs to save space.@
        expand can be called to grab some storage
        from the display area to augment the SMALLTALK wo
**rkspace.
        expand 200 would take 200 lines off the top of th
**e display
```

```
        and increase core by 6400 words.@1'

        'THE SMALLTALK READ ROUTINE (name changed to prot
**ect ev)'

(to junta scanner : : read1 tablscan rdnum mknum rdstr rt
**b1 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=026⇒(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)*si
**gn)
        ⇑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)

         'INITIALIZATION OF READ TABLES'
         ☞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) 0
**101 to 0132 and 0141 to 0172.
        scanner digbit ☞(rbuf←rdnum. flag⇒(rbuf←☞.)) 060
**to 071 and 026.
        scanner sepbit ☞(scanner next sepbit) 011 and 013
** and 015 and 040.
        scanner qtbit ☞(rbuf ← rdstr) 047.
        scanner 0 ☞(scanner skip. rbuf ← (read1 rtb1) eva
**l) 05.
        scanner 0 ☞(scanner skip. rbuf ← read1 rtb1) 050.
        scanner 0 ☞(scanner skip. rbuf ← nil. done) 051.
        scanner 0 ☞(rbuf ← nil. done) 0 and 012.
        for i to type length (type[i] ← type[i] ⊖ qtbit)
done
☞read ← #junta.
PUT read ☞TITLE ☞read. 'cover our tracks'

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