```
<MPS>NEWFSP.NLS;2, 2-JAN-74 14:54 EHS ;                                    0
fsp: PROGRAM =

  BEGIN   -- MPS Free Storage Package --                                   1

  -- declarations                                                          1a

    wcmax: INTEGER = 777777B;     -- maximum node size                     1b
    zmax: INTEGER = 77777B;       -- maximum zone size (!)                 1b1
    thrmax: INTEGER = 77B;        -- maximum zone threshold                1b2
                                                                           1b3
    NodeHeader: MODE = RECORD[                                             1b4
      prevwc: IN [0..wcmax],      -- prev adjacent node word count
      ptr: POINTER,               -- free list link
      wc: IN [0..wcmax],          -- word count for node
      nodfre: BOOLEAN,            -- this node free
      panfre: BOOLEAN];           -- prev adjacent node free

    ZoneHeader: MODE = RECORD[                                             1b5
      zrvr: IN [0..zmax],         -- rover for zone
      zprv: IN [0..zmax],         -- previous for zone
      zthr: IN [0..thrmax]];      -- threshold for zone
                                                                           1b6
    nhl: INTEGER = RECORDSIZE[NodeHeader];                                 1b7
    zhl: INTEGER = RECORDSIZE[ZoneHeader];                                 1b8

    NoRoomInZone, BadZoneSize, BadNodeSize, CrummyNode: SIGNAL CODE;       1b9

  -- procedures
                                                                           1c
    MakeNode: PROCEDURE [zone: POINTER, n: INTEGER] RETURNS [POINTER] =    1c1
      -- return address of node of (at least) size n in the specified zone.
      BEGIN
      rvr: POINTER;           -- rover (pointer into free list)
      srv: POINTER;           -- saved rover, to check zone full
      prv: POINTER;           -- previous rover (free list entry)
      thr: INTEGER;           -- min free list node size in zone
      node: POINTER;          -- pointer to the found node
      nodwc: INTEGER;         -- node word count (includes nhl)
      newwc: INTEGER;         -- new word count
      fanod: POINTER;         -- following adjacent node pointer
      panod: POINTER;         -- previous adjacent node pointer
      panwc: INTEGER;         -- prev adjacent node word count
      ptr: POINTER;
      srv ← rvr ← zone.zrvr + zone;
      prv ← zone.zprv + zone;
      thr ← zone.zthr;
      IF n <= 0 THEN    -- bad size for node
        SIGNAL[BadNodeSize];
      n ← n + nhl;    -- actual size needed
      IF n < thr THEN  -- new threshold for zone
        thr ← zone.zthr ← n;
      (search) DO BEGIN
        fanod ← rvr + (nodwc ← rvr.wc);
        IF NOT rvr.panfre THEN BEGIN
          IF fanod.nodfre THEN BEGIN
            -- remove this from free list and move to next
            rvr ← zone + (prv.ptr ← (rvr.ptr := 0));
            REPEAT search;
```

```
        END;
      CASE newwc ← nodwc - n OF
        >= nhl:  -- split
          BEGIN
          node ← rvr + (rvr.wc ← newwc);
          node.prevwc ← newwc;
          node.panfre ← TRUE;
          node.wc ← n;
          IF newwc < thr THEN
            -- remove from free list
            zone.zrvr ← prv.ptr ← (rvr.ptr := 0)
          ELSE zone.zrvr ← rvr - zone;
          END;
        < 0:  -- too small
          BEGIN
          prv ← (rvr := rvr.ptr + zone);
          IF rvr = srv THEN  -- no node in zone big enough
            SIGNAL[NoRoomInZone];
          REPEAT search;
          END;
        ENDCASE  -- big enough for "exact" fit
          BEGIN
          node ← rvr;
          -- remove from free list
          zone.zrvr ← prv.ptr ← rvr.ptr;
          END;
      node.nodfre ← fanod.panfre ← FALSE;
      zone.zprv ← prv - zone;
      FOR ptr THRU [node+nhl .. node+n) DO ptr.word0 ← 0;
      RETURN [node + nhl]
      END;
    -- prev node is free - merge with this node
    panod ← rvr - (panwc ← rvr.prevwc);
    IF panod.ptr = 0 THEN
      BEGIN
      panod.ptr ← rvr.ptr;
      rvr ← panod;
      END
    ELSE rvr ← rvr.ptr + zone;
    srv ← rvr;  -- must not give up too soon
    prv.ptr ← rvr - zone;
    panod.wc ← fanod.prevwc ← panwc + nodwc;
    END;
  END;

FreeNode: PROCEDURE [zone, node: POINTER] =                    1c2
  BEGIN
  prv, fanod, panod: POINTER;
  panwc, nodwc: INTEGER;
  node ← node - nhl;  -- actual location of header
  IF node.wc<nhl OR node<zone
    THEN  BEGIN  -- be cautious
      SIGNAL[CrummyNode,node];
      RETURN
    END;
  IF node.panfre THEN  -- merge with previous node
    BEGIN
    nodwc ← node.wc;
    node ← node - (panwc ← node.prevwc);
```

```
    node.wc ← panwc + nodwc;
    -- fix up following node's prevwc field below
    END
  ELSE
    BEGIN
    node.nodfre ← TRUE;
    node.ptr ← 0;
    END;
  (MergeFreeNodes)  -- check for following free nodes
    BEGIN
    fanod ← node + (nodwc ← node.wc);
    IF fanod.nodfre THEN
      IF fanod.ptr = 0 THEN  -- not on free list
        BEGIN
        -- coalesce and repeat
        node.wc ← nodwc + fanod.wc;
        REPEAT MergeFreeNodes;
        END
      ELSE NULL
        -- if the following node is on the free list, then don't add this
        one.  They will be coalesced in MakeNode.
    ELSE IF node.ptr = 0 THEN
      BEGIN
      -- find current end of free list
      prv ← zone.zprv + zone;
      -- add to the end of the free list
      node.ptr ← (prv.ptr := zone.zprv ← node - zone);
      END;
    END;
  fanod.prevwc ← nodwc;
  fanod.panfre ← TRUE;
  RETURN END;

SplitNode: PROCEDURE [zone, node: POINTER, size: INTEGER] =                1c3
  -- split the specified node into two sections, the first of which is a node
  of size words, and the second of which is freed.
  BEGIN
  newwc: INTEGER;
  node ← node - nhl;  -- actual location of node header
  size ← size + nhl;  -- actual size needed
  IF (newwc ← node.wc - size) < nhl THEN RETURN;
  node ← node + (node.wc ← size);
  node.prevwc ← size;
  node.panfre ← node.nodfre ← FALSE;
  node.wc ← newwc;
  FreeNode[zone, node + nhl];
  RETURN END;

NodeSize: PROCEDURE [zone, node: POINTER] RETURNS [INTEGER] =              1c4
  -- return the size of the node (not including header).
  BEGIN
  node ← node - nhl;  -- actual location of node header
  RETURN[node.wc - nhl] END;

MakeZone: PROCEDURE [zone: POINTER, size: INTEGER] =                       1c5
  -- the zone consists of
    -- zone header  (length zhl)
    -- available space - initially one free node (length size-zhl-nhl)
    -- "free" node always on free list  (length nhl)
```

```
    BEGIN
    rvr: POINTER;
    IF size NOT IN (zhl+2*nhl,zmax] THEN
      SIGNAL[BadZoneSize];
    zone.zthr ← thrmax;  -- initial value for threshold
    -- available space
      rvr ← zone + (zone.zrvr ← zhl);
      rvr.nodfre ← TRUE;
      rvr.panfre ← FALSE;
      rvr.wc ← size - zhl - nhl;
      rvr ← zone + (rvr.ptr ← zone.zprv ← size - nhl);
    -- "free" node for free list anchor
      rvr.ptr ← zhl;
      rvr.nodfre ← FALSE;
      rvr.panfre ← FALSE;
      rvr.wc ← 0;
    RETURN END;


END.
LIST ...                                                    1d
```

```
<MPS>NEWMPSSTR.NLS;2, 2-JAN-74 20:31 EHS ;                                    0
mpsstr: PROGRAM =

                                                                              1
  BEGIN   -- MPS string package --                                            1a

  DIRECTORY                                                                   1b
    Registers: INCLUDE (mps,mplregs,),                                        1b1
    PDP10Words: INCLUDE (mps,pdp10words,);                                    1b2

  -- signals generated by MPSSTR                                            1c
    STRBoundsFault: SIGNAL CODE;    -- index out of bounds                    1c1
    STRInvalidDesc: SIGNAL CODE;    -- type field not text                    1c2
    STRWriteFault: SIGNAL CODE;     -- write of read-only string              1c3

  -- descriptor tag codes                                                    1d
    TextType: INTEGER = 1;     -- text descriptor                             1d1

  StrDesc: MODE = RECORD[   -- text descriptor --                             1e
    sIdent: WORD,             -- user-set identification                      1e1
    sType: IN [0 .. 777777B],                                                 1e2
    sReadOnly: BOOLEAN,       -- read-only flag                               1e3
    sEnd: INTEGER,            -- current end position (next char index)       1e4
    sFront: INTEGER,          -- current front position                       1e5
    sMaxEnd: INTEGER,         -- maximum value of sEnd                        1e6
    sPointer: POINTER];       -- text base address                            1e7

  lTextDesc: INTEGER = RECORDSIZE[StrDesc];                                   1f
  -- characters in word                                                       1g
    WordSize: INTEGER = wordsize;  CharSize: INTEGER = charsize;              1g1
    CharsPerWord: INTEGER = WordSize/CharSize;                                1g2

  -- some PDP10 opcodes                                                      1h
    Move: INTEGER=200B; Ildb: INTEGER=134B; Idpb: INTEGER=136B;               1h1

  CharPointer: ARRAY OF WORD ←                                                1i
    -- skeleton pointers suitable for selecting each character position in a word
    with a PDP10 Ildb/Idpb instruction --                                     1i1
    [4407B8, 3507B8, 2607B8,  1707B8, 1007B8];

                                                                              1i2
  StrByteP: PROCEDURE [string: STRING] RETURNS [WORD] =                       1j
    --  make an Ildb byte pointer for the first char of the string --        1j1
    BEGIN                                                                     1j2
    RETURN[CharPtr[string.sPointer, string.sFront]]                           1j3
    END.

                                                                              1j4
  NumChars: PROCEDURE [bptr1, bptr2: WORD] RETURNS [INTEGER] =                1k
    -- returns the length of the string from bptr1 to bptr2 inclusive --     1k1
    BEGIN                                                                     1k2
    RETURN[ (bptr2$address-bptr1$address)*CharsPerWord +
    (bptr1$bitsfollowing-bptr2$bitsfollowing)/CharSize]                       1k3
    END;

                                                                              1k4
  MakeStrDesc: PROCEDURE [string: STRING, front, end, maxend: INTEGER, pointer:
  POINTER] =                                                                  11
    -- initialize text descriptor --.                                         111
    BEGIN                                                                     112
    IF string.sReadOnly THEN                                                  113
      ERROR[STRWriteFault, string];
    IF front<0 OR end<front OR maxend<end THEN                                114
```

```
    ERROR[STRBoundsFault, string];                                          115
  string.sType ← TextType;  string.sPointer ← pointer;                      116
  string.sFront ← front; string.sEnd ← end;                                 117
  string.sMaxend ← maxend;  RETURN
  END;                                                                      118

MakeString: PROCEDURE RETURNS [STRING] =                                    1m
  -- create a string descriptor --                                          1m1
  BEGIN  string: POINTER;                                                   1m2
  string ← mplAllocate[lTextDesc];                                          1m3
  string.sType ← TextType;  string.sReadOnly ← FALSE;                       1m4
  RETURN[string]                                                            1m5
  END;                                                                      1m6

ReleaseString: PROCEDURE [string: STRING] =                                 1n
  -- release a string descriptor --                                         1n1
  BEGIN                                                                     1n2
  IF string.sReadOnly THEN                                                  1n3
    SIGNAL[STRWriteFault, string];
  mplFree[string];  RETURN                                                  1n4
  END;                                                                      1n5

STRCopyIntoZone: PROCEDURE [string: STRING, zone: POINTER] RETURNS [STRING] =
                                                                            1o
  -- copies descriptor and text into the given zone --                      1o1
  BEGIN   copy: STRING[0];  -- descriptor only                              1o2
  n: INTEGER;                                                               1o3
  n ← STRLength[string];  copy ← MakeNode[zone, lTextDesc];                 1o4
  copy.sType ← TextType;  copy.sReadOnly ← FALSE;                           1o5
  MakeStrDesc[copy,0,0,n,MakeNode[zone,MAX[WordsForBody[n],1]]];            1o6
  AppendString[copy,string];  RETURN[copy]                                  1o7
  END;                                                                      1o8

STREraseFromZone: PROCEDURE [string: STRING, zone: POINTER] =               1p
  -- deletes descriptor and text from the given zone --                     1p1
  BEGIN                                                                     1p2
  IF string.sType # TextType THEN                                           1p3
    ERROR[STRInvalidDesc, string];
  FreeNode[zone, string.sPointer];  string.sType ← 0;                       1p4
  FreeNode[zone, string];  RETURN                                           1p5
  END;                                                                      1p6

ValidString: PROCEDURE [string: STRING] RETURNS [BOOLEAN] =                 1q
  -- checks a descriptor for a valid type code --                           1q1
  BEGIN                                                                     1q2
  RETURN[string.sType = TextType]                                           1q3
  END;                                                                      1q4

STRType: PROCEDURE [string: STRING] RETURNS [INTEGER] =                     1r
  BEGIN                                                                     1r1
  RETURN[string.sType]                                                      1r2
  END;                                                                      1r3

STRIdent: PROCEDURE [string: STRING] RETURNS [WORD] =                       1s
  BEGIN                                                                     1s1
  RETURN[string.sIdent]                                                     1s2
  END;                                                                      1s3

SetDescIdent: PROCEDURE [string: STRING, ident: WORD] =                     1t
```

```
  BEGIN                                                                 1t1
  IF string.sReadOnly THEN                                              1t2
    ERROR[STRWriteFault, string];
  string.sIdent ← ident;  RETURN                                        1t3
  END;

                                                                        1t4
STRReadOnly: PROCEDURE [string: STRING] RETURNS [BOOLEAN] =              1u
  BEGIN                                                                 1u1
  RETURN[string.sReadOnly]                                              1u2
  END;

                                                                        1u3
STRLength: PROCEDURE [string: STRING] RETURNS [INTEGER] =               1v
  BEGIN                                                                 1v1
  IF string.sType # TextType THEN                                       1v2
    ERROR[STRInvalidDesc, string];
  RETURN[string.sEnd - string.sFront]                                   1v3
  END;

                                                                        1v4
STRLast: PROCEDURE [string: STRING] RETURNS [INTEGER] =                 1w
  BEGIN                                                                 1w1
  RETURN[STRLength[string]-1]                                           1w2
  END;

                                                                        1w3
STRFront: PROCEDURE [string: STRING] RETURNS [INTEGER] =                1x
  BEGIN                                                                 1x1
  IF string.sType # TextType THEN                                       1x2
    ERROR[STRInvalidDesc, string];
  RETURN[string.sFront]                                                 1x3
  END;

                                                                        1x4
STREnd: PROCEDURE [string: STRING] RETURNS [INTEGER] =                  1y
  BEGIN                                                                 1y1
  IF string.sType # TextType THEN                                       1y2
    ERROR[STRInvalidDesc, string];
  RETURN[string.sEnd]                                                   1y3
  END;

                                                                        1y4
STRMaxend: PROCEDURE [string: STRING] RETURNS [INTEGER] =               1z
  BEGIN                                                                 1z1
  IF string.sType # TextType THEN                                       1z2
    ERROR[STRInvalidDesc, string];
  RETURN[string.sMaxend]                                                1z3
  END;

                                                                        1z4
STRPointer: PROCEDURE [string: STRING] RETURNS [POINTER] =              1a@
  BEGIN                                                                 1a@1
  IF string.sType # TextType THEN                                       1a@2
    ERROR[STRInvalidDesc, string];
  RETURN[string.sPointer]                                               1a@3
  END;

                                                                        1a@4
SetStrNull: PROCEDURE [string: STRING] =                                1aa
  BEGIN                                                                 1aa1
  IF string.sType # TextType THEN                                       1aa2
    ERROR[STRInvalidDesc, string];
  IF string.sReadOnly THEN                                              1aa3
    ERROR[STRWriteFault, string];
  string.sEnd ← string.sFront;  RETURN                                  1aa4
  END;
```

```
                                                                        1aa5
NullString: PROCEDURE [s: STRING] RETURNS [BOOLEAN] =                    1ab
  -- returns TRUE iff the string is null --                             1ab1
  BEGIN                                                                 1ab2
  RETURN[STRLength[s]=0]                                                1ab3
  END.

                                                                        1ab4
SetStrReadOnly: PROCEDURE [string: STRING, value: BOOLEAN] =             1ac
  BEGIN                                                                 1ac1
  IF string.sType # TextType THEN                                       1ac2
    ERROR[STRInvalidDesc, string];
  IF string.sReadOnly AND ~value THEN                                   1ac3
    SIGNAL[STRWriteFault, string];
  string.sReadOnly ← value;  RETURN                                     1ac4
  END;

                                                                        1ac5
SetStrLength: PROCEDURE [string: STRING, nchars: INTEGER] =              1ad
  BEGIN                                                                 1ad1
  IF string.sType # TextType THEN                                       1ad2
    ERROR[STRInvalidDesc, string];
  IF string.sReadOnly THEN                                              1ad3
    ERROR[STRWriteFault, string];
  IF nchars NOT IN [0, string.sMaxend-string.sFront] THEN               1ad4
    ERROR[STRBoundsFault,string];
  string.sEnd ← string.sFront+nchars;  RETURN                           1ad5
  END;

                                                                        1ad6
ScanPointer: PROCEDURE [string: STRING, position: INTEGER] RETURNS [WORD,
INTEGER] =                                                               1ae
  BEGIN   di: WORD;  b: INTEGER;  w: POINTER;                           1ae1
  IF string.sType # TextType THEN                                       1ae2
    ERROR[STRInvalidDesc, string];
  IF string.sReadOnly THEN                                              1ae3
    ERROR[STRWriteFault, string];
  -- di ← CharPtr[string.sPointer, string.sFront+position]; --          1ae4
    w ← string.sPointer + DIV[(string.sFront+position)/CharsPerWord :b];
    di ← CharPointer[b] + w;
  RETURN[di, string.sMaxend-string.sFront]                              1ae5
  END;

                                                                        1ae6
CharPtr: PROCEDURE [base: POINTER, byte: INTEGER] RETURNS [WORD] =       1af
  -- make an Ildb/Idpb byte pointer for the CharSize byte following base
  (position indexed by byte from zero) --                               1af1
  -- see also NthChar, AppendChar for an inline expansion --            1af2
  BEGIN   b, w: INTEGER;                                                1af3
  w ← DIV[byte/CharsPerWord :b];                                        1af4
  RETURN[CharPointer[b] + base+w]                                       1af5
  END;

                                                                        1af6
NthChar: PROCEDURE [string: STRING, position: INTEGER] RETURNS [CHARACTER] =
                                                                         1ag
  BEGIN   si: WORD;  b: INTEGER;  w: POINTER;                           1ag1
  IF string.sType # TextType THEN                                       1ag2
    ERROR[STRInvalidDesc, string];
  IF position NOT IN [0 .. string.sEnd-string.sFront) THEN              1ag3
    RETURN[EOS];
  -- si ← CharPtr[string.sPointer, string.sFront+position]; --          1ag4
    w ← string.sPointer + DIV[(string.sFront+position)/CharsPerWord :b];
    si ← CharPointer[b] + w;
```

```
  INLINE[ Ildb a1,si ];  RETURN[a1]                                    1ag5
  END;                                                                 1ag6

SetNthChar: PROCEDURE [string: STRING, position: INTEGER, char: CHARACTER] =
                                                                        1ah
  BEGIN   di: WORD;                                                    1ah1
  IF string.sType # TextType THEN                                      1ah2
    ERROR[STRInvalidDesc, string];
  IF string.sReadOnly THEN                                             1ah3
    ERROR[STRWriteFault, string];
  IF position NOT IN [0 .. string.sEnd-string.sFront) THEN             1ah4
    ERROR[STRBoundsFault, string];
  di ← CharPtr[string.sPointer, string.sFront+position];               1ah5
  INLINE[ Move a1,char;  Idpb a1,di ];  RETURN                         1ah6
  END;                                                                 1ah7

AppendChar: PROCEDURE [string: STRING, char: CHARACTER] =               1ai
  BEGIN   di: WORD;  b: INTEGER;  w: POINTER;                          1ai1
  (appendcheck) DO                                                     1ai2
    BEGIN
    IF string.sType # TextType THEN
      ERROR[STRInvalidDesc, string];
    IF string.sReadOnly THEN
      ERROR[STRWriteFault, string];
    IF string.sEnd < string.sMaxend THEN EXIT appendcheck;
    SIGNAL[STRBoundsFault, string];
    END;
  -- di ← CharPtr[string.sPointer, string.sEnd]; --                    1ai3
    w ← string.sPointer + DIV[string.sEnd/CharsPerWord :b];
    di ← CharPointer[b] + w;
  string.sEnd ← string.sEnd + 1;                                       1ai4
  INLINE[ Move a1,char;  Idpb a1,di ];  RETURN                         1ai5
  END;                                                                 1ai6

AppendString: PROCEDURE [to, from: STRING] =                            1aj
  BEGIN                                                                1aj1
  n: INTEGER;  si, di: WORD;                                           1aj2
  (appendtest) DO                                                      1aj3
    BEGIN
    IF to.sType # TextType THEN
      ERROR[STRInvalidDesc, to];
    IF from.sType # TextType THEN
      ERROR[STRInvalidDesc, from];
    IF to.sReadOnly THEN
      ERROR[STRWriteFault, to];
    n ← from.sEnd - from.sFront;
    IF n <= to.sMaxend-to.sEnd THEN EXIT appendtest;
    SIGNAL[STRBoundsFault, to];
    END;
  si ← CharPtr[from.sPointer, from.sFront];                            1aj4
  di ← CharPtr[to.sPointer, to.sEnd];                                  1aj5
  to.sEnd ← to.sEnd + n;                                               1aj6
  WHILE n > 0 DO                                                       1aj7
    BEGIN  n ← n-1;
    INLINE[ Ildb a1,si;  Idpb a1,di ];
    END;
  RETURN                                                               1aj8
  END;                                                                 1aj9
```

```
AppendSubString: PROCEDURE [to, from: STRING, first, last: INTEGER] =          1ak
  BEGIN   n: INTEGER;                                                          1ak1
  FOR n THRU [MAX[first,0] .. MIN[last,STRLast[from]]] DO                      1ak2
    AppendChar[to,NthChar[from,n]];
  RETURN                                                                       1ak3
  END;
                                                                               1ak4
AppendNumStr: PROCEDURE [to: STRING, num, base: INTEGER] =                      1a1
  BEGIN   power: INTEGER;  char: CHARACTER;                                    1a11
  IF num < 0 THEN                                                              1a12
    BEGIN  AppendChar[to, '-];  num ← -num;
    END;
  power ← 1;                                                                   1a13
  WHILE power*base <= num DO  power ← power*base;                              1a14
  UNTIL power < 1 DO                                                           1a15
    BEGIN  char ← DIV[num/power :num];
    AppendChar[to, char+'0];  power ← power/base;
    END;
  RETURN                                                                       1a16
  END;
                                                                               1a17
AppendBlanks: PROCEDURE [to: STRING, count: INTEGER] =                          1am
  BEGIN                                                                        1am1
  THRU [1 .. count] DO AppendChar[to, ' ];                                     1am2
  RETURN                                                                       1am3
  END;
                                                                               1am4
CompareString: PROCEDURE [string1, string2: STRING] RETURNS [INTEGER] =        1an
  -- return -1 if 1 < 2, 0 if 1 = 2, +1 if 1 > 2 --                           1an1
  BEGIN   n1, n2: INTEGER;  si1, si2: WORD;                                    1an2
  IF string1.sType # TextType THEN                                             1an3
    ERROR[STRInvalidDesc, string1];
  IF string2.sType # TextType THEN                                             1an4
    ERROR[STRInvalidDesc, string2];
  n1 ← string1.sEnd - string1.sFront;                                          1an5
  n2 ← string2.sEnd - string2.sFront;                                          1an6
  si1 ← CharPtr[string1.sPointer, string1.sFront];                             1an7
  si2 ← CharPtr[string2.sPointer, string2.sFront];                             1an8
  THRU [1 .. MIN[n1, n2]] DO                                                   1an9
    BEGIN  INLINE[ Ildb a1,si1;  Ildb a2,si2 ];
    CASE a1-a2 OF
      <0: RETURN[-1];
      >0: RETURN[1];
      ENDCASE;
    END;
  RETURN[ CASE n1 OF                                                           1an10
    <n2: -1;
    >n2: 1;
    ENDCASE 0]
  END;
                                                                               1an11
EqualString: PROCEDURE [string1, string2: STRING] RETURNS [BOOLEAN] =          1ao
  -- return TRUE iff string1 and string2 are (textually) equal. (a fast version
  of CompareString when ordering is not needed) --                             1ao1
  BEGIN   n: INTEGER;  si1, si2: WORD;                                         1ao2
  IF string1.sType # TextType THEN                                             1ao3
    ERROR[STRInvalidDesc, string1];
  IF string2.sType # TextType THEN                                             1ao4
    ERROR[STRInvalidDesc, string2];
```

```
    n ← string1.sEnd - string1.sFront;                                  1ao5
    IF string2.sEnd - string2.sFront # n THEN                           1ao6
      RETURN[FALSE];
    si1 ← CharPtr[string1.sPointer, string1.sFront];                    1ao7
    si2 ← CharPtr[string2.sPointer, string2.sFront];                    1ao8
    THRU [1 .. n] DO                                                    1ao9
      BEGIN  INLINE[ Ildb a1,si1;  Ildb a2,si2 ];
      IF a1 # a2 THEN
        RETURN[FALSE];
      END;
    RETURN[TRUE]                                                        1ao10
    END.

                                                                        1ao11
  WordsForBody: PROCEDURE [nchars: INTEGER] RETURNS [INTEGER] =          1ap
    -- return number of words for nchars characters --                  1ap1
    BEGIN                                                               1ap2
    RETURN[(nchars + (CharsPerWord-1))/CharsPerWord]                    1ap3
    END.

                                                                        1ap4
  END.
  LIST ...                                                              1aq
```