```
-- file TreePack.Mesa
-- last modified by Satterthwaite, April 25, 1978  8:39 AM

DIRECTORY
  SystemDefs: FROM "systemdefs",
  TableDefs: FROM "tabledefs",
  LitDefs: FROM "litdefs",
  SymDefs: FROM "symdefs",
  TreeDefs:  FROM "treedefs";

TreePack: PROGRAM
    IMPORTS SystemDefs, TableDefs
    EXPORTS TreeDefs =
 PUBLIC
  BEGIN
  OPEN TreeDefs;

  endTreeIndex: TreeIndex = LAST[TreeIndex];
  EndMark: TreeLink = [subtree[index: endTreeIndex]];

  TreeSonField: TYPE = RECORD [soni: TreeLink]; -- for sequencing through sons
  TreeXIndex: TYPE = POINTER [0..TableDefs.TableLimit) TO TreeSonField;

  treeOpen: PRIVATE BOOLEAN ← FALSE;

  TreeLinkStack: PRIVATE TYPE = DESCRIPTOR FOR ARRAY OF TreeLink;

  kStack: PRIVATE TreeLinkStack;
  kI: PRIVATE CARDINAL;

  tb: PRIVATE TableDefs.TableBase;              -- tree base

  UpdateBase: PRIVATE TableDefs.TableNotifier =
    BEGIN
    tb ← base[treetype];  RETURN
    END;


  TreeInit: PROCEDURE =
    BEGIN
    IF treeOpen THEN TreeErase[];
    kStack ← AllocStack[100];  kI ← 0;
    TableDefs.AddNotify[UpdateBase];
    IF maketree[none,0] # empty THEN ERROR;     -- reserve null
    treeOpen ← TRUE;  RETURN
    END;

  TreeErase: PROCEDURE =
    BEGIN
    treeOpen ← FALSE;
    TableDefs.DropNotify[UpdateBase];  FreeStack[kStack];  RETURN
    END;


  AllocStack: PRIVATE PROCEDURE [size: CARDINAL] RETURNS [s: TreeLinkStack] =
    BEGIN
    base: POINTER;
    base ← SystemDefs.AllocateSegment[size*SIZE[TreeLink]];
    s ← DESCRIPTOR[base, SystemDefs.SegmentSize[base]/SIZE[TreeLink]];
    RETURN
    END;

  FreeStack: PRIVATE PROCEDURE [s: TreeLinkStack] =
    BEGIN
    IF LENGTH[s] # 0 THEN SystemDefs.FreeSegment[BASE[s]];
    RETURN
    END;

  ExpandStack: PRIVATE PROCEDURE [s: TreeLinkStack, delta: CARDINAL]
      RETURNS [t: TreeLinkStack] =
    BEGIN
    i: CARDINAL;
    t ← AllocStack[LENGTH[s]+delta];
    FOR i IN [0 .. MIN[LENGTH[s], LENGTH[t]]) DO t[i] ← s[i] ENDLOOP;
    FreeStack[s];  RETURN
    END;
```

```
mlpush: PROCEDURE [v: TreeLink] =
  BEGIN
  IF kI >= LENGTH[kStack] THEN  kStack ← ExpandStack[kStack, 25];
  kStack[kI] ← v;  kI ← kI+1;
  RETURN
  END;

mlpop: PROCEDURE RETURNS [TreeLink] =
  BEGIN
  RETURN [kStack[kI←kI-1]]
  END;


mlinsert: PROCEDURE [v: TreeLink, n: CARDINAL] =
  BEGIN
  i: CARDINAL;
  IF kI >= LENGTH[kStack] THEN  kStack ← ExpandStack[kStack, 25];
  i ← kI;  kI ← kI+1;
  THROUGH [1 .. n) DO  kStack[i] ← kStack[i-1];  i ← i-1  ENDLOOP;
  kStack[i] ← v;
  RETURN
  END;

mlextract: PROCEDURE [n: CARDINAL] RETURNS [v: TreeLink] =
  BEGIN
  i: CARDINAL;
  i ← kI - n;  v ← kStack[i];
  THROUGH [1 .. n) DO kStack[i] ← kStack[i+1];  i ← i+1  ENDLOOP;
  kI ← kI - 1;
  RETURN [v]
  END;


maketree: PROCEDURE [name: NodeName, count: INTEGER] RETURNS [TreeLink] =
  BEGIN
  nSons: CARDINAL = ABS[count];
  node: TreeIndex = TableDefs.GetChunk[TreeNodeSize+nSons];
  p: TreeXIndex;
  d: INTEGER;
  p ← LOOPHOLE[node + TreeNodeSize + (IF count<0 THEN 0 ELSE nSons-1)];
  d ← IF count<0 THEN 1 ELSE -1;
  THROUGH [1 .. nSons]
    DO  (tb+p).soni ← kStack[kI←kI-1];  p ← p + d  ENDLOOP;
  (tb+node).name ← name;  (tb+node).nsons ← nSons;
  (tb+node).info ← 0;
  (tb+node).shared ← (tb+node).attr1 ← (tb+node).attr2 ← FALSE;
  RETURN[TreeLink[subtree[index: node]]]
  END;

makelist: PROCEDURE [size: INTEGER] RETURNS [TreeLink] =
  BEGIN
  pushlist[size];
  RETURN [mlpop[]]
  END;


pushtree: PROCEDURE [name: NodeName, count: INTEGER] =
  BEGIN
  mlpush[maketree[name, count]];
  RETURN
  END;

pushlist: PROCEDURE [size: INTEGER] =
  BEGIN
  nSons: CARDINAL = ABS[size];
  node: TreeIndex;
  p: TreeXIndex;
  d: INTEGER;
  SELECT nSons FROM
    1 =>  NULL;
    0 =>  mlpush[empty];
    ENDCASE =>
      BEGIN
      IF nSons IN (0..MaxNSons]
```

```
        THEN
          BEGIN
          node ← TableDefs.GetChunk[TreeNodeSize+nSons];
          p ← LOOPHOLE[node + TreeNodeSize+(nSons-1)];
          END
        ELSE
          BEGIN
          node ← TableDefs.GetChunk[TreeNodeSize+(nSons+1)];
          p ← LOOPHOLE[node + TreeNodeSize+nSons];
          (tb+p).soni ← EndMark;  p ← p-1;
          END;
      IF size > 0
        THEN  d ← -1
        ELSE  BEGIN  d ← 1;  p ← LOOPHOLE[node + TreeNodeSize]  END;
      THROUGH [1 .. nSons]
        DO  (tb+p).soni ← kStack[kI ← kI-1];  p ← p+d  ENDLOOP;
      (tb+node).name ← list;
      (tb+node).info ← 0;
      (tb+node).shared ← (tb+node).attr1 ← (tb+node).attr2 ← FALSE;
      (tb+node).nsons ← IF nSons IN (0..MaxNSons] THEN nSons ELSE 0;
      mlpush[TreeLink[subtree[index: node]]];
      END;
  RETURN
  END;

pushproperlist: PROCEDURE [size: INTEGER] =
  BEGIN
  node: TreeIndex;
  IF size ~IN [-1..1]
    THEN pushlist[size]
    ELSE
      BEGIN
      node ← TableDefs.GetChunk[TreeNodeSize + 1];
      (tb+node).name ← list;
      (tb+node).info ← 0;
      (tb+node).shared ← (tb+node).attr1 ← (tb+node).attr2 ← FALSE;
      (tb+node).nsons ← ABS[size];
      (tb+node).son1 ← IF size = 0 THEN EndMark ELSE mlpop[];
      mlpush[TreeLink[subtree[index: node]]];
      END;
  RETURN
  END;


pushhashtree: PROCEDURE [hti: SymDefs.HTIndex] =
  BEGIN
  mlpush[TreeLink[hash[index: hti]]];
  RETURN
  END;

pushsymtree: PROCEDURE [sei: SymDefs.ISEIndex] =
  BEGIN
  mlpush[TreeLink[symbol[index: sei]]];
  RETURN
  END;

pushlittree: PROCEDURE [lti: LitDefs.LTIndex] =
  BEGIN
  mlpush[TreeLink[literal[info: [word[lti]]]]];
  RETURN
  END;

pushstringlittree: PROCEDURE [sti: LitDefs.STIndex] =
  BEGIN
  mlpush[TreeLink[literal[info: [string[sti]]]]];
  RETURN
  END;


setinfo: PROCEDURE [info: UNSPECIFIED] =
  BEGIN
  v: TreeLink = kStack[kI-1];
  WITH v SELECT FROM
    subtree =>  IF index # nullTreeIndex THEN  (tb+index).info ← info;
    ENDCASE =>  NULL;
  RETURN
```

```
  END;

setattr: PROCEDURE [attr: [1..2], value: BOOLEAN] =
  BEGIN
  v: TreeLink = kStack[kI-1];
  WITH v SELECT FROM
    subtree =>
      IF index = nullTreeIndex
        THEN  ERROR
        ELSE
          SELECT attr FROM
            1 => (tb+index).attr1 ← value;
            2 => (tb+index).attr2 ← value;
            ENDCASE;
    ENDCASE =>  ERROR;
  RETURN
  END;


freenode: PROCEDURE [node: TreeIndex] =
  BEGIN
  p: TreeXIndex;
  n: CARDINAL;
  IF node # nullTreeIndex AND ~(tb+node).shared
    THEN
      BEGIN  p ← LOOPHOLE[node + TreeNodeSize];
      IF (tb+node).name # list OR (tb+node).nsons # 0
        THEN
          BEGIN   n ← (tb+node).nsons;
          THROUGH [1 .. n]
            DO
            WITH (tb+p).soni SELECT FROM
              subtree =>  freenode[index];
              ENDCASE;
            p ← p+1;
            ENDLOOP;
          END
        ELSE
          BEGIN  n ← 1;
          UNTIL (tb+p).soni = EndMark
            DO
            WITH (tb+p).soni SELECT FROM
              subtree =>  freenode[index];
              ENDCASE;
            n ← n+1;  p ← p+1;
            ENDLOOP;
          END;
      TableDefs.FreeChunk[node, TreeNodeSize+n];
      END;
  RETURN
  END;

freetree: PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
  BEGIN
  WITH t SELECT FROM
    subtree =>  freenode[index];
    ENDCASE;
  RETURN [empty]
  END;


-- procedures for tree testing

GetNode: PROCEDURE [t: TreeLink] RETURNS [TreeIndex] =
  BEGIN
  WITH t SELECT FROM
    subtree => RETURN [index];
    ENDCASE => ERROR
  END;

shared: PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
  BEGIN
  RETURN [WITH t SELECT FROM
    subtree => IF index = nullTreeIndex THEN FALSE ELSE (tb+index).shared,
    ENDCASE => FALSE]
  END;
```

```
setshared: PROCEDURE [t: TreeLink, shared: BOOLEAN] =
  BEGIN
  WITH t SELECT FROM
    subtree => IF index # nullTreeIndex THEN (tb+index).shared ← shared;
    ENDCASE;
  RETURN
  END;

testtree: PROCEDURE [t: TreeLink, name: NodeName] RETURNS [BOOLEAN] =
  BEGIN
  RETURN [WITH t SELECT FROM
    subtree => index # nullTreeIndex AND (tb+index).name = name,
    ENDCASE => FALSE]
  END;

SonCount: PRIVATE PROCEDURE [node: TreeIndex] RETURNS [CARDINAL] =
  BEGIN
  RETURN [SELECT node FROM
    nullTreeIndex, endTreeIndex => 0,
    ENDCASE => IF (tb+node).name = list AND (tb+node).nsons = 0
      THEN listlength[TreeLink[subtree[index: node]]] + 1
      ELSE (tb+node).nsons]
  END;


-- procedures for tree traversal

UpdateTree: PROCEDURE [root: TreeLink, map: TreeMap] RETURNS [v: TreeLink] =
  BEGIN
  node: TreeIndex;
  nSons: CARDINAL;
  p: TreeXIndex;
  WITH root SELECT FROM
    subtree =>
      BEGIN  node ← index;
      IF node  # nullTreeIndex
        THEN
          BEGIN
          nSons ← SonCount[node];  p ← LOOPHOLE[node + TreeNodeSize];
          THROUGH [1 .. nSons]
            DO
            IF (tb+p).soni # EndMark THEN (tb+p).soni ← map[(tb+p).soni];
            p ← p+1;
            ENDLOOP;
          END;
      v ← root;
      END;
    ENDCASE =>  v ← map[root];
  RETURN
  END;


-- procedures for list testing

listlength: PROCEDURE [t: TreeLink] RETURNS [CARDINAL] =
  BEGIN
  node: TreeIndex;
  p: TreeXIndex;
  n: CARDINAL;
  IF t = empty THEN RETURN [0];
  WITH t SELECT FROM
    subtree =>
      BEGIN  node ← index;
      IF (tb+node).name # list THEN RETURN [1];
      n ← (tb+node).nsons;
      IF n # 0 THEN RETURN [n];
      FOR p ← LOOPHOLE[node+TreeNodeSize], p+1 UNTIL (tb+p).soni = EndMark
        DO  n ← n+1  ENDLOOP;
      RETURN [n]
      END;
    ENDCASE => RETURN [1]
  END;

listhead: PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
  BEGIN
```

```
    node: TreeIndex;
    IF t = empty THEN ERROR;
    WITH t SELECT FROM
      subtree =>
        BEGIN  node ← index;
        IF (tb+node).name # list THEN RETURN [t];
        IF (tb+node).son1 # EndMark THEN RETURN [(tb+node).son1];
        ERROR
        END;
      ENDCASE => RETURN [t]
    END;

  listtail: PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
    BEGIN
    node: TreeIndex;
    IF t = empty THEN ERROR;
    WITH t SELECT FROM
      subtree =>
        BEGIN  node ← index;
        IF (tb+node).name # list THEN RETURN [t];
        IF (tb+node).son1 # EndMark
          THEN RETURN [(tb + LOOPHOLE[node+TreeNodeSize+(listlength[t]-1), TreeXIndex]).soni];
        ERROR
        END;
      ENDCASE =>  RETURN [t]
    END;


  -- procedures for list traversal

  scanlist: PROCEDURE [root: TreeLink, action: TreeScan] =
    BEGIN
    node: TreeIndex;
    p: TreeXIndex;
    n: CARDINAL;
    t: TreeLink;
    IF root # empty
      THEN
        WITH root SELECT FROM
          subtree =>
            BEGIN  node ← index;
            IF (tb+node).name # list
              THEN action[root]
              ELSE
                BEGIN  p ← LOOPHOLE[node + TreeNodeSize];
                IF (n ← (tb+node).nsons) # 0
                  THEN
                    THROUGH [1 .. n]
                      DO  action[(tb+p).soni];  p ← p+1  ENDLOOP
                  ELSE
                    UNTIL (t←(tb+p).soni) = EndMark
                      DO  action[t];  p ← p+1  ENDLOOP;
                END;
            END;
          ENDCASE =>  action[root];
    RETURN
    END;

  reversescanlist: PROCEDURE [root: TreeLink, action: TreeScan] =
    BEGIN
    node: TreeIndex;
    p: TreeXIndex;
    n: CARDINAL;
    IF root # empty
      THEN
        WITH root SELECT FROM
          subtree =>
            BEGIN  node ← index;
            IF (tb+node).name # list
              THEN action[root]
              ELSE
                BEGIN  n ← listlength[root];
                p ← LOOPHOLE[node + TreeNodeSize + n];
                THROUGH [1 .. n]
                  DO  p ← p-1;  action[(tb+p).soni]  ENDLOOP;
                END;
```

```
          END;
        ENDCASE => action[root];
    RETURN
    END;

  updatelist: PROCEDURE [root: TreeLink, map: TreeMap] RETURNS [TreeLink] =
    BEGIN
    node: TreeIndex;
    p: TreeXIndex;
    t: TreeLink;
    n: CARDINAL;
    IF root = empty THEN RETURN [empty];
    WITH root SELECT FROM
      subtree =>
        BEGIN  node ← index;
        IF (tb+node).name # list THEN RETURN [map[root]];
        p ← LOOPHOLE[node + TreeNodeSize];
        IF (n ← (tb+node).nsons) # 0
          THEN
            THROUGH [1 .. n]
              DO  (tb+p).soni ← map[(tb+p).soni];  p ← p+1  ENDLOOP
          ELSE
            UNTIL (t←(tb+p).soni) = EndMark
              DO  (tb+p).soni ← map[t];  p ← p+1  ENDLOOP;
        RETURN [root]
        END;
      ENDCASE =>  RETURN [map[root]];
    END;

  reverseupdatelist: PROCEDURE [root: TreeLink, map: TreeMap] RETURNS [TreeLink] =
    BEGIN
    node: TreeIndex;
    p: TreeXIndex;
    n: CARDINAL;
    IF root = empty THEN RETURN [empty];
    WITH root SELECT FROM
      subtree =>
        BEGIN  node ← index;
        IF (tb+node).name # list THEN RETURN [map[root]];
        n ← listlength[root];  p ← LOOPHOLE[node + (TreeNodeSize + n)];
        THROUGH [1 .. n]
          DO  p ← p-1;  (tb+p).soni ← map[(tb+p).soni]  ENDLOOP;
        RETURN [root]
        END;
      ENDCASE => RETURN [map[root]];
    END;


-- cross-table tree manipulation

  CopyTree: PROCEDURE [root: TreeId, map: TreeMap] RETURNS [v: TreeLink] =
    BEGIN
    sNode, dNode: TreeIndex;
    size, nSons: CARDINAL;
    s, d: TreeXIndex;
    WITH root.link SELECT FROM
      subtree =>
        BEGIN  sNode ← index;
        IF sNode = nullTreeIndex
          THEN  v ← root.link
          ELSE
            BEGIN
            size ← NodeSize[root.baseP, sNode];  nSons ← size - TreeNodeSize;
            dNode ← TableDefs.GetChunk[size];
            s ← LOOPHOLE[sNode + TreeNodeSize];
            d ← LOOPHOLE[dNode + TreeNodeSize];
            THROUGH [1 .. nSons]
              DO
              (tb+d).soni ← IF (root.baseP↑ + s).soni = EndMark
                              THEN EndMark
                              ELSE map[(root.baseP↑ + s).soni];
              s ← s+1;  d ← d+1;
              ENDLOOP;
            (tb+dNode).name ← (root.baseP↑ + sNode).name;
            (tb+dNode).mark ← (root.baseP↑ + sNode).mark;
            (tb+dNode).shared ← FALSE;
```

```
            (tb+dNode).nsons ← (root.baseP↑ + sNode).nsons;
            (tb+dNode).info ← (root.baseP↑ + sNode).info;
            (tb+dNode).attr1 ← (root.baseP↑ + sNode).attr1;
            (tb+dNode).attr2 ← (root.baseP↑ + sNode).attr2;
            v ← [subtree[index: dNode]];
            END;
        END;
      ENDCASE => v ← map[root.link];
    RETURN
    END;

  IdentityMap: TreeMap =
    BEGIN
    RETURN [IF t.tag = subtree AND ~shared[t]
        THEN CopyTree[[baseP:@tb, link:t], IdentityMap]
        ELSE t]
    END;


  NodeSize: PROCEDURE [baseP: TableDefs.TableFinger, node: TreeIndex] RETURNS [size: CARDINAL] =
    BEGIN
    p: TreeXIndex;
    IF node = nullTreeIndex
      THEN size ← 0
      ELSE
        IF (baseP↑ + node).name # list OR (baseP↑ + node).nsons # 0
          THEN size ← TreeNodeSize + (baseP↑ + node).nsons
          ELSE
            BEGIN
            size ← TreeNodeSize + 1; p ← LOOPHOLE[node+TreeNodeSize];
            UNTIL (baseP↑ + p).soni = EndMark
              DO size ← size + 1; p ← p+1 ENDLOOP;
            END;
    RETURN
    END;


  END.
```