```
-- file Pass2.Mesa
-- last modified by Satterthwaite, July 16, 1978  9:53 AM

DIRECTORY
  ComData: FROM "comdata"
    USING [
      bodyIndex, bodyRoot, defBodyLimit, idLOCK,
      importCtx, mainBody, mainCtx, moduleCtx,
      monitored, nBodies, nSigCodes, textIndex],
  CompilerDefs: FROM "compilerdefs",
  ErrorDefs: FROM "errordefs" USING [error, errorhti, Warning],
  SymDefs: FROM "symdefs"
    USING [
      BodyLink, BodyInfo, BodyRecord, ContextLevel, SERecord, TransferMode,
      HTIndex, SEIndex, CSEIndex, ISEIndex, recordCSEIndex,
      CTXIndex, BTIndex, CBTIndex,
      HTNull,  SENull,  CSENull,  ISENull,  recordCSENull,
      CTXNull,  BTNull,  CBTNull,
      1G, 1L, 1Z, typeANY, typeTYPE,
      setype, ctxtype, bodytype],
  SymTabDefs: FROM "symtabdefs"
    USING [
      fillctxse, makenewctx, makenonctxse, makeSEChain, NameClash,
      nextlevel, NextSe, StaticNestError],
  TableDefs: FROM "tabledefs"
    USING [
      TableBase, TableNotifier,
      AddNotify, Allocate, DropNotify, TableBounds],
  TreeDefs: FROM "treedefs"
    USING [
      TreeLink, TreeIndex, TreeMap, TreeScan,
      empty, nullTreeIndex,
      treetype,
      freenode, GetNode, listhead, listlength,
      scanlist, testtree, updatelist];

Pass2: PROGRAM
    IMPORTS
        ErrorDefs, SymTabDefs, TableDefs, TreeDefs,
        dataPtr: ComData
    EXPORTS CompilerDefs =
  BEGIN
  OPEN TreeDefs, SymTabDefs, SymDefs;

  tb: TableDefs.TableBase;       -- tree base (private copy)
  seb: TableDefs.TableBase;      -- se table base (private copy)
  ctxb: TableDefs.TableBase;     -- context table base (private copy)
  bb: TableDefs.TableBase;       -- body table base (private copy)

  Notify: TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever tables are repacked
    tb ← base[treetype];
    seb ← base[setype];  ctxb ← base[ctxtype];
    bb ← base[bodytype];
    RETURN
    END;

  ContextInfo: TYPE = RECORD [
    ctx: CTXIndex,
    staticLevel: ContextLevel,
    seChain: ISEIndex];

  current: ContextInfo;

  NewContext: PROCEDURE [level: ContextLevel, entries: CARDINAL, unique: BOOLEAN] =
    BEGIN
    OPEN current;
    staticLevel ← level;
    IF entries = 0 AND ~unique
      THEN  BEGIN  ctx ← CTXNull;  seChain ← ISENull  END
      ELSE
        BEGIN
        ctx ← makenewctx[level];
        (ctxb+ctx).selist ← seChain ← makeSEChain[ctx, entries, FALSE];
        END;
    RETURN
```

```
    END;


-- main driver

  P2Unit: PUBLIC PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
    BEGIN
    node: TreeIndex;
    TableDefs.AddNotify[Notify];
    node ← GetNode[t];
      BEGIN
      ENABLE    -- default error reporting
        BEGIN
        NameClash => BEGIN ErrorDefs.errorhti[duplicateId, hti]; RESUME END;
        StaticNestError => BEGIN ErrorDefs.error[staticNesting]; RESUME END;
        END;
      dataPtr.textIndex ← (tb+node).info;
      NewContext[1Z, listlength[(tb+node).son1]+listlength[(tb+node).son3], FALSE];
      dataPtr.moduleCtx ← current.ctx;
      scanlist[(tb+node).son1, IdDefinition];
      scanlist[(tb+node).son3, Module];
      END;
    TableDefs.DropNotify[Notify];
    RETURN [t]
    END;


  lockLambda: TreeIndex;

  Module: TreeScan =
    BEGIN
    saved: ContextInfo;
    saveIndex: CARDINAL = dataPtr.textIndex;
    node: TreeIndex = GetNode[t];
    dataPtr.bodyIndex ← CBTNull;  dataPtr.nBodies ← dataPtr.nSigCodes ← 0;
    btLink ← [which:parent, index:BTNull];
    dataPtr.textIndex ← (tb+node).info;
    -- process import list
      saved ← current;
      NewContext[1G, listlength[(tb+node).son1], FALSE];
      dataPtr.importCtx ← current.ctx;
      scanlist[(tb+node).son1, IdDefinition];
      current ← saved;
    dataPtr.monitored ← (tb+node).son4 # empty;
    lockLambda ← Lambda[(tb+node).son4, 1L];
    DeclList[(tb+node).son5, SENull];
    BodyList[dataPtr.bodyRoot];
    dataPtr.defBodyLimit ← TableDefs.TableBounds[bodytype].size;
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  IdDefinition: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    saveIndex: CARDINAL = dataPtr.textIndex;
    dataPtr.textIndex ← (tb+node).info;
    (tb+node).son1 ← Ids[list: (tb+node).son1, public: FALSE, link: node];
    dataPtr.textIndex ← saveIndex;  RETURN
    END;


-- monitor lock processing

  Lambda: PROCEDURE [item: TreeLink, level: ContextLevel] RETURNS [node: TreeIndex] =
    BEGIN
    saved: ContextInfo = current;
    node ← GetNode[item];
    IF node # nullTreeIndex
      THEN
        BEGIN
        NewContext[level, CountIds[(tb+node).son1], FALSE];
        (tb+node).info ← current.ctx;
        DeclList[(tb+node).son1, SENull];
        IF (tb+node).son2 # empty THEN Exp[(tb+node).son2];
        END;
    current ← saved;  RETURN
```

```
    END;

  ImplicitLock: PROCEDURE [sei: ISEIndex] =
    BEGIN
    WITH (tb+lockLambda).son2 SELECT FROM
      hash =>  fillctxse[sei, index, FALSE];
      ENDCASE => ERROR;
     BEGIN  OPEN (seb+sei);
     extended ← public ← writeonce ← constant ← linkSpace ← FALSE;
     idtype ← dataPtr.idLOCK;
     idinfo ← 1;  idvalue ← nullTreeIndex;
     mark3 ← TRUE;  mark4 ← FALSE;
     END;
    (tb+lockLambda).son2 ← [symbol[index: sei]];  RETURN
    END;


 -- body processing

  btLink: BodyLink;

  AllocateBody: PROCEDURE [node: TreeIndex] RETURNS [bti: CBTIndex] =
    BEGIN       -- queue body for later processing
    dataPtr.nBodies ← dataPtr.nBodies+1;
    -- force nesting message here
      SELECT nextlevel[current.staticLevel] FROM
        1G, 1L =>
          BEGIN
          bti ← TableDefs.Allocate[bodytype, SIZE[Outer Callable BodyRecord]];
          (bb+bti)↑ ← BodyRecord[,,,,, Callable[,,,,,,, Outer[]]];
          END;
        ENDCASE =>
          BEGIN
          bti ← TableDefs.Allocate[bodytype, SIZE[Inner Callable BodyRecord]];
          (bb+bti)↑ ← BodyRecord[,,,,, Callable[,,,,,,, Inner[]]];
          END;
    (bb+bti).firstSon ← BTNull;
    (bb+bti).info ← BodyInfo[Internal[
        bodyTree: node,
        sourceIndex: dataPtr.textIndex,
        stOrigin: ,
        frameSize: ]];
    (bb+bti).id ← IF (tb+node).attr1
        THEN FirstId[(tb+node).son1]
        ELSE ISENull;
    LinkBody[bti];  RETURN
    END;

  LinkBody: PROCEDURE [bti: BTIndex] =
    BEGIN
    IF btLink.which = parent
      THEN
        BEGIN
        (bb+bti).link ← btLink;
        IF btLink.index = BTNull
          THEN dataPtr.bodyRoot ← bti
          ELSE (bb+btLink.index).firstSon ← bti;
        END
      ELSE
        BEGIN
        (bb+bti).link ← (bb+btLink.index).link;
        (bb+btLink.index).link ← [which:sibling, index: bti];
        END;
    btLink ← [which:sibling, index: bti];
    RETURN
    END;


  BodyList: PROCEDURE [firstBti: BTIndex] =
    BEGIN
    bti: BTIndex;
    IF (bti ← firstBti) # BTNull
      THEN
        DO
        WITH (bb+bti) SELECT FROM
          Callable =>  Body[LOOPHOLE[bti, CBTIndex]];
```

```
        ENDCASE => NULL;
      IF (bb+bti).link.which = parent THEN EXIT;
      bti ← (bb+bti).link.index;
      ENDLOOP;
  RETURN
  END;

Body: PROCEDURE [bti: CBTIndex] =
  BEGIN
  node: TreeIndex;
  bodyLevel: ContextLevel;
  nLocks: [0..1];
  oldBodyIndex: CBTIndex = dataPtr.bodyIndex;
  oldBtLink: BodyLink = btLink;
  saved: ContextInfo = current;
  dataPtr.bodyIndex ← bti;
  btLink ← [which:parent, index:bti];
  WITH (bb+bti).info SELECT FROM
    Internal => node ← GetNode[(tb+LOOPHOLE[bodyTree,TreeIndex]).son3];
    ENDCASE => ERROR;
  bodyLevel ← nextlevel[saved.staticLevel !StaticNestError => RESUME];
  nLocks ← IF dataPtr.monitored AND
      bodyLevel = 1G AND (tb+lockLambda).attr1 THEN 1 ELSE 0;
  NewContext[
      level: bodyLevel,
      entries: nLocks + CountIds[(tb+node).son2],
      unique: bodyLevel = 1G];
  (bb+bti).localCtx ← current.ctx;  (bb+bti).level ← bodyLevel;
  (bb+bti).monitored ← nLocks # 0;
  IF bodyLevel = 1G
    THEN BEGIN  dataPtr.mainCtx ← current.ctx;  dataPtr.mainBody ← bti  END;
  scanlist[(tb+node).son1, Exp];
  IF nLocks # 0
    THEN
      BEGIN
      ImplicitLock[current.seChain];
      current.seChain ← NextSe[current.seChain]
      END;
  DeclList[(tb+node).son2, SENull];
  scanlist[(tb+node).son3, Stmt];
  BodyList[(bb+bti).firstSon];
  current ← saved;  dataPtr.bodyIndex ← oldBodyIndex;  btLink ← oldBtLink;
  RETURN
  END;


Inline: TreeScan =
  BEGIN
  scanlist[t, Exp];  RETURN
  END;


-- declarations

DeclList: PROCEDURE [t: TreeLink, linkId: SEIndex] =
  BEGIN

  DeclItem: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    subNode: TreeIndex;
    saveIndex: CARDINAL = dataPtr.textIndex;
    dataPtr.textIndex ← (tb+node).info;
    (tb+node).mark ← FALSE;
    (tb+node).son1 ← Ids[
              list: (tb+node).son1,
              public: (tb+node).attr2,
              link: node];
    IF testtree[(tb+node).son2, modeTC]
      THEN  TypeExp[(tb+node).son3, FirstId[(tb+node).son1], linkId]
      ELSE
        BEGIN
        TypeExp[(tb+node).son2, SENull, linkId];
        IF (tb+node).son3 # empty AND (tb+node).son3.tag = subtree
          THEN
            BEGIN
```

```
            subNode ← GetNode[(tb+node).son3];
            SELECT (tb+subNode).name FROM
              entry, internal =>
                BEGIN
                IF ~dataPtr.monitored OR ~testtree[(tb+subNode).son1, body]
                  THEN ErrorDefs.error[misplacedEntry]
                  ELSE
                    BEGIN
                    WITH (tb+subNode).son1 SELECT FROM
                      subtree =>
                        SELECT (tb+subNode).name FROM
                          entry =>  (tb+index).attr1 ← TRUE;
                          internal =>  (tb+index).attr2 ← TRUE;
                          ENDCASE;
                      ENDCASE;
                    IF (tb+subNode).name = internal AND (tb+node).attr2
                      THEN  ErrorDefs.Warning[attrClash];
                    END;
                (tb+node).son3 ← (tb+subNode).son1;
                (tb+subNode).son1 ← empty;  freenode[subNode];
                END;
              ENDCASE;
            END;
        IF (tb+node).son3 # empty
          THEN
            WITH (tb+node).son3 SELECT FROM
              subtree =>
                SELECT (tb+index).name FROM
                  body => (tb+index).info ← AllocateBody[node];
                  signalinit =>
                    BEGIN
                    (tb+index).info ← dataPtr.nSigCodes;
                    dataPtr.nSigCodes ← dataPtr.nSigCodes+1;
                    END;
                  inline => scanlist[(tb+index).son1, Inline];
                  ENDCASE => Exp[(tb+node).son3];
              ENDCASE =>  Exp[(tb+node).son3];
        END;
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  scanlist[root:t, action:DeclItem];  RETURN
  END;


CountIds: PROCEDURE [declList: TreeLink] RETURNS [n: CARDINAL] =
  BEGIN

  nIds: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    n ← n + listlength[(tb+node).son1];
    RETURN
    END;

  n ← 0;  scanlist[declList, nIds];  RETURN
  END;


-- id list manipulation

Ids: PROCEDURE [list: TreeLink, public: BOOLEAN, link: TreeIndex]
    RETURNS [TreeLink] =
  BEGIN

  Id: TreeMap =
    BEGIN
    hti: HTIndex;
    sei: ISEIndex;
    ctx: CTXIndex = current.ctx;
    WITH t SELECT FROM
      hash =>  hti ← index;
      symbol =>  hti ← (seb+index).htptr;
      ENDCASE =>  ERROR;
    sei ← current.seChain;  current.seChain ← NextSe[current.seChain];
```

```
      fillctxse[sei, hti, public];
      v ← TreeLink[symbol[index: sei]];
      (seb+sei).idtype ← typeANY;
      (seb+sei).public ← public;
      (seb+sei).idvalue ← link;
      (seb+sei).extended ← (seb+sei).linkSpace ← FALSE;
      RETURN
      END;

    RETURN [updatelist[root:list, map:Id]]
    END;


  FirstId: PROCEDURE [t: TreeLink] RETURNS [ISEIndex] =
    BEGIN
    head: TreeLink = listhead[t];
    WITH head SELECT FROM
      symbol =>  RETURN [index];
      ENDCASE =>  ERROR;
    END;


-- type manipulation

  TypeExp: PROCEDURE [t: TreeLink, typeId, linkId: SEIndex]  =
    BEGIN -- processes type-node of declitem subtree
    node: TreeIndex;
    sei: CSEIndex;
    tCtx: CTXIndex;
    nFields: CARDINAL;
    WITH t SELECT FROM
      subtree =>
        BEGIN  node ← index;
        SELECT (tb+node).name FROM
          modeTC =>  sei ← typeTYPE;
          enumeratedTC =>
            BEGIN
            sei ← makenonctxse[SIZE[enumerated constructor SERecord]];
            tCtx ← Enumeration[node];
            (seb+sei).typeinfo ← enumerated[
                ordered: TRUE,
                valuectx: tCtx,
                nvalues: ];
            AssignValues[sei, IF typeId # SENull THEN typeId ELSE sei];
            END;
          recordTC, monitoredTC =>
            BEGIN
            sei ← makenonctxse[SIZE[notlinked record constructor SERecord]];
            [tCtx, nFields] ← FieldList[
                t: (tb+node).son1,
                level: 1Z,
                typeId: IF typeId # SENull THEN typeId ELSE sei];
            (seb+sei).typeinfo ← record[
                machineDep: (tb+node).attr1,
                unifield: nFields = 1 AND ~(tb+node).attr2,
                argument: FALSE,
                defaultFields: FALSE,
                length: ,
                comparable: FALSE,
                privateFields: FALSE,
                lengthUsed: FALSE,
                fieldctx: tCtx,
                monitored: (tb+node).name = monitoredTC,
                variant: (tb+node).attr2,
                linkpart: notlinked[]];
            END;
          variantTC =>
            BEGIN
            sei ← makenonctxse[SIZE[linked record constructor SERecord]];
            tCtx ← FieldList[t:(tb+node).son1, level:1Z, typeId:typeId].ctx;
            (seb+sei).typeinfo ← record[
                machineDep: (tb+node).attr1,
                unifield: FALSE,
                argument: FALSE,
                defaultFields: FALSE,
                length: ,
```

```
        comparable: FALSE,
        privateFields: FALSE,
        lengthUsed: FALSE,
        fieldctx: tCtx,
        monitored: FALSE,
        variant: (tb+node).attr2,
        linkpart: linked[linkId]];
    END;
  pointerTC =>
    BEGIN  sei ← makenonctxse[SIZE[pointer constructor SERecord]];
    (seb+sei).typeinfo ← pointer[
        ordered: (tb+node).attr1,
        basing: (tb+node).attr2,
        readonly: FALSE,
        dereferenced: FALSE,
        pointedtotype: ];
    TypeExp[(tb+node).son1, SENull, SENull];
    END;
  arrayTC =>
    BEGIN  sei ← makenonctxse[SIZE[array constructor SERecord]];
    (seb+sei).typeinfo ← array[
        packed: (tb+node).attr1,
        lengthUsed: FALSE,
        comparable: FALSE,
        indextype: ,
        componenttype: ];
    IF (tb+node).son1 # empty
      THEN TypeExp[(tb+node).son1, SENull, SENull];
    TypeExp[(tb+node).son2, SENull, SENull];
    END;
  arraydescTC =>
    BEGIN  sei ← makenonctxse[SIZE[arraydesc constructor SERecord]];
    (seb+sei).typeinfo ← arraydesc[describedType: ];
    TypeExp[(tb+node).son1, SENull, SENull];
    END;
  procTC =>  sei ← Transfer[node, procedure];
  portTC =>  sei ← Transfer[node, port];
  signalTC =>  sei ← Transfer[node, signal];
  errorTC =>  sei ← Transfer[node, error];
  processTC =>  sei ← Transfer[node, process];
  programTC =>  sei ← Transfer[node, program];
  definitionTC =>
    BEGIN
    sei ← makenonctxse[SIZE[definition constructor SERecord]];
    (seb+sei).typeinfo ← definition[nGfi: 1, defCtx: ];
    END;
  unionTC =>  sei ← Union[node, linkId];
  relativeTC =>
    BEGIN
    sei ← makenonctxse[SIZE[relative constructor SERecord]];
    (seb+sei).typeinfo ← relative[
        baseType: ,
        offsetType: ,
        resultType: ];
    TypeExp[(tb+node).son1, SENull, SENull];
    TypeExp[(tb+node).son2, SENull, SENull];
    END;
  subrangeTC =>
    BEGIN  sei ← makenonctxse[SIZE[subrange constructor SERecord]];
    (seb+sei).typeinfo ← subrange[
        filled: FALSE,
        empty: FALSE,
        flexible: FALSE,
        rangetype: ,
        origin: ,
        range: ];
    TypeExp[(tb+node).son1, SENull, SENull];
    Interval[(tb+node).son2];
    END;
  longTC =>
    BEGIN  sei ← makenonctxse[SIZE[long constructor SERecord]];
    (seb+sei).typeinfo ← long[rangetype: ];
    TypeExp[(tb+node).son1, SENull, SENull];
    END;
  implicitTC, frameTC =>  sei ← CSENull;
  dot, discrimTC =>
```

```
          BEGIN  TypeExp[(tb+node).son1, SENull, SENull];  sei ← CSENull;
          END;
        ENDCASE =>
          BEGIN  sei ← CSENull;  ErrorDefs.error[nonTypeCons]  END;
      (tb+node).info ← sei;
      END;
    ENDCASE => NULL;
  RETURN
  END;

Enumeration: PROCEDURE [node: TreeIndex] RETURNS [ctx: CTXIndex] =
  BEGIN
  saved: ContextInfo = current;
  NewContext[1Z, listlength[(tb+node).son1], TRUE];  ctx ← current.ctx;
  (tb+node).son1 ← Ids[
      list: (tb+node).son1,
      public: (tb+node).attr1,
      link: nullTreeIndex];
  current ← saved;  RETURN
  END;

AssignValues: PROCEDURE [type: CSEIndex, valueType: SEIndex] =
  BEGIN
  i: CARDINAL;
  sei: ISEIndex;
  WITH (seb+type) SELECT FROM
    enumerated =>
      BEGIN  i ← 0;
      FOR sei ← (ctxb+valuectx).selist, NextSe[sei] UNTIL sei = SENull
        DO  OPEN (seb+sei);
        idtype ← valueType;  idinfo ← 0;
        idvalue ← i;  i ← i+1;
        writeonce ← constant ← mark3 ← mark4 ← TRUE;
        ENDLOOP;
      nvalues ← i;
      END;
    ENDCASE =>  ERROR;
  END;


FieldList: PROCEDURE [t: TreeLink, level: ContextLevel, typeId: SEIndex]
    RETURNS [ctx: CTXIndex, nFields: CARDINAL] =
  BEGIN
  saved: ContextInfo = current;
  nFields ← CountIds[t];
  NewContext[level, nFields, TRUE];  ctx ← current.ctx;
  DeclList[t, typeId];
  current ← saved;  RETURN
  END;

Transfer: PROCEDURE [node: TreeIndex, mode: TransferMode] RETURNS [sei: CSEIndex] =
  BEGIN
  tSei1, tSei2: recordCSEIndex;
  sei ← makenonctxse[SIZE[transfer constructor SERecord]];
  tSei1 ← ArgList[(tb+node).son1];
  tSei2 ← ArgList[(tb+node).son2];
  (seb+sei).typeinfo ← transfer[
      mode: mode,
      inrecord: tSei1,
      outrecord: tSei2];
  RETURN
  END;

ArgList: PROCEDURE [t: TreeLink] RETURNS [type: recordCSEIndex] =
  BEGIN
  tCtx: CTXIndex;
  nFields: CARDINAL;
  IF t = empty
    THEN type ← recordCSENull
    ELSE
      BEGIN
      type ← LOOPHOLE[makenonctxse[SIZE[notlinked record constructor SERecord]]];
      [tCtx, nFields] ← FieldList[t, nextlevel[current.staticLevel], type];
      (seb+type).typeinfo ← record[
              machineDep: FALSE,
              unifield: nFields = 1,
```

```
            argument: TRUE,
            defaultFields: FALSE,
            length: ,
            comparable: FALSE,
            privateFields: FALSE,
            lengthUsed: FALSE,
            fieldctx: tCtx,
            monitored: FALSE,
            variant: FALSE,
            linkpart: notlinked[]];
      END;
  RETURN
  END;

Union: PROCEDURE [node: TreeIndex, linkId: SEIndex] RETURNS [sei: CSEIndex] =
  BEGIN
  tagId: ISEIndex;
  subnode: TreeIndex;
  saved: ContextInfo = current;
  current.ctx ← CTXNull;  current.seChain ← makeSEChain[CTXNull, 1, FALSE];
  DeclList[(tb+node).son1, SENull];
  subnode ← GetNode[(tb+node).son1];
  tagId ← FirstId[(tb+subnode).son1];
  WITH (tb+subnode).son2 SELECT FROM
    subtree =>
      IF (tb+index).name = implicitTC
        THEN (tb+index).info ← MakeTagType[(tb+node).son2];
    ENDCASE => NULL;
  NewContext[1Z, CountIds[(tb+node).son2], TRUE];
  DeclList[(tb+node).son2, linkId
    !NameClash =>
      BEGIN  ErrorDefs.errorhti[duplicateTag, hti];  RESUME  END];
  sei ← makenonctxse[SIZE[union constructor SERecord]];
  (seb+sei).typeinfo ← union[
            casectx: current.ctx,
            overlayed: (tb+node).attr1,
            controlled: (seb+tagId).htptr # HTNull,
            tagsei: tagId,
            equalLengths: FALSE];
  current ← saved;  RETURN
  END;

MakeTagType: PROCEDURE [t: TreeLink] RETURNS [type: CSEIndex] =
  BEGIN
  saved: ContextInfo = current;

  CollectTags: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    (tb+node).son1 ← Ids[
            list: (tb+node).son1,
            public: (tb+node).attr2,
            link: nullTreeIndex
        !NameClash => RESUME];
    RETURN
    END;

  NewContext[1Z, CountIds[t], TRUE];
  type ← makenonctxse[SIZE[enumerated constructor SERecord]];
  (seb+type).typeinfo ← enumerated[
      ordered: FALSE,
      valuectx: current.ctx,
      nvalues: ];
  scanlist[t, CollectTags];
  AssignValues[type, type];
  current ← saved;  RETURN
  END;

-- statements

Stmt: PROCEDURE [stmt: TreeLink] =
  BEGIN
  node, subNode: TreeIndex;
```

```
saveIndex: CARDINAL = dataPtr.textIndex;
IF stmt = empty THEN RETURN;
WITH stmt SELECT FROM
  subtree =>
    BEGIN  node ← index;
    dataPtr.textIndex ← (tb+node).info;
    SELECT (tb+node).name FROM
      assign =>  BEGIN  Exp[(tb+node).son1];  Exp[(tb+node).son2]  END;
      extract =>
        BEGIN  scanlist[(tb+node).son1, Exp];  Exp[(tb+node).son2]  END;
      apply =>
        BEGIN
        Exp[(tb+node).son1];  scanlist[(tb+node).son2, Exp];
        IF (tb+node).nsons > 2 THEN CatchPhrase[(tb+node).son3];
        END;
      block =>  Block[node];
      ifstmt =>
        BEGIN  OPEN (tb+node);
        Exp[son1];  scanlist[son2, Stmt];  scanlist[son3, Stmt];
        END;
      casestmt =>
        BEGIN  OPEN (tb+node);
        Exp[son1];  SelectionList[son2, Stmt];  Stmt[son3];
        END;
      bindstmt =>
        BEGIN  OPEN (tb+node);
        Exp[son1];
        IF son2 # empty THEN Exp[son2];
        SelectionList[son3, Stmt];
        Stmt[son4];
        END;
      dostmt =>
        BEGIN  OPEN (tb+node);
        IF son1 # empty
          THEN
            BEGIN  subNode ← GetNode[son1];
            IF (tb+subNode).son1 # empty THEN Exp[(tb+subNode).son1];
            SELECT (tb+subNode).name FROM
              forseq =>
                BEGIN
                Exp[(tb+subNode).son2];  Exp[(tb+subNode).son3];
                END;
              upthru, downthru =>  Range[(tb+subNode).son2];
              ENDCASE => ERROR;
            END;
        IF son2 # empty THEN Exp[son2];
        scanlist[son3, Exp];
        scanlist[son4, Stmt]; scanlist[son5, Stmt]; scanlist[son6, Stmt];
        END;
      return, resume =>  scanlist[(tb+node).son1, Exp];
      label =>
        BEGIN
        scanlist[(tb+node).son1, Stmt];  scanlist[(tb+node).son2, Stmt];
        END;
      goto, exit, loop, continue, retry, syserror, nullstmt =>  NULL;
      signal, error, xerror, start, restart,
      join, wait, notify, broadcast, dst, lst, lstf =>
        Exp[(tb+node).son1];
      stop =>  IF (tb+node).son1 # empty THEN CatchPhrase[(tb+node).son1];
      openstmt =>
        BEGIN
        scanlist[(tb+node).son1, Exp];  scanlist[(tb+node).son2, Stmt];
        END;
      enable =>
        BEGIN
        CatchPhrase[(tb+node).son1];  scanlist[(tb+node).son2, Stmt];
        END;
      list =>  scanlist[stmt, Stmt];
      item =>  Stmt[(tb+node).son2];
      ENDCASE =>  ErrorDefs.error[unimplemented];
    END;
  ENDCASE =>  NULL;
dataPtr.textIndex ← saveIndex;  RETURN
END;
```

```
Block: PROCEDURE [node: TreeIndex] =
  BEGIN
  bti: BTIndex;
  oldBtLink: BodyLink;
  saved: ContextInfo = current;
  NewContext[
      level: saved.staticLevel,
      entries: CountIds[(tb+node).son1],
      unique: FALSE];
  bti ← TableDefs.Allocate[bodytype, SIZE[Other BodyRecord]];
  (bb+bti)↑ ← BodyRecord[
      link: ,
      firstSon: BTNull,
      localCtx: current.ctx,
      level: current.staticLevel,
      info: BodyInfo[Internal[
        bodyTree: node,
        sourceIndex: (tb+node).info,
        stOrigin: ,
        frameSize: ]],
      extension: Other[]];
  LinkBody[bti];  oldBtLink ← btLink;  btLink ← [which:parent, index:bti];
  (tb+node).info ← bti;
  DeclList[(tb+node).son1, SENull];
  scanlist[(tb+node).son2, Stmt];
  BodyList[(bb+bti).firstSon];
  current ← saved;  btLink ← oldBtLink;  RETURN
  END;


SelectionList: PROCEDURE [t: TreeLink, selection: TreeScan] =
  BEGIN

  Item: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    saveIndex: CARDINAL = dataPtr.textIndex;
    dataPtr.textIndex ← (tb+node).info;
    scanlist[(tb+node).son1, Exp];  selection[(tb+node).son2];
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  scanlist[t, Item];  RETURN
  END;


CatchPhrase: PROCEDURE [t: TreeLink] =
  BEGIN
  node: TreeIndex = GetNode[t];
  saved: ContextInfo = current;
  NewContext[
      level: nextlevel[saved.staticLevel],
      entries: 0,
      unique: FALSE];
  SelectionList[(tb+node).son1, Stmt];
  IF (tb+node).nsons > 1 THEN scanlist[(tb+node).son2, Stmt];
  current ← saved;  RETURN
  END;


-- expressions

Exp: PROCEDURE [exp: TreeLink] =
  BEGIN
  node, subNode: TreeIndex;
  IF exp = empty THEN RETURN;
  WITH exp SELECT FROM
    subtree =>
      BEGIN  node ← index;
      SELECT (tb+node).name FROM
        apply =>
          BEGIN
          Exp[(tb+node).son1];  scanlist[(tb+node).son2, Exp];
          IF (tb+node).nsons > 2 THEN CatchPhrase[(tb+node).son3];
          END;
        signal, error, start, fork, join,
```

```
        dot, uparrow,
        uminus, lengthen, float, abs, not,
        addr, base, length, register, memory, new =>
          Exp[(tb+node).son1];
        plus, minus, times, div, mod,
        relE, relN, relL, relGE, relG, relLE,
        or, and, assignx =>
          BEGIN  Exp[(tb+node).son1];  Exp[(tb+node).son2]  END;
        in, notin =>
          BEGIN  Exp[(tb+node).son1];  Range[(tb+node).son2]  END;
        ifexp =>
          BEGIN
          Exp[(tb+node).son1];  Exp[(tb+node).son2];  Exp[(tb+node).son3];
          END;
        caseexp =>
          BEGIN  OPEN (tb+node);
          Exp[son1];  SelectionList[son2, Exp];  Exp[son3];
          END;
        bindexp =>
          BEGIN  OPEN (tb+node);
          Exp[son1];
          IF son2 # empty THEN Exp[son2];
          SelectionList[son3, Exp];
          Exp[son4];
          END;
        min, max =>  scanlist[(tb+node).son1, Exp];
        arraydesc =>
          SELECT listlength[(tb+node).son1] FROM
            1 => Exp[(tb+node).son1];
            3 =>
              BEGIN
              subNode ← GetNode[(tb+node).son1];
              Exp[(tb+subNode).son1];  Exp[(tb+subNode).son2];
              IF (tb+subNode).son3 # empty
                THEN TypeExp[(tb+subNode).son3, SENull, SENull];
              END;
            ENDCASE => ERROR;
        clit, llit, mwconst =>  NULL;
        loophole =>
          BEGIN
          Exp[(tb+node).son1];
          IF (tb+node).son2 # empty
            THEN TypeExp[(tb+node).son2, SENull, SENull];
          END;
        size, first, last =>  TypeExp[(tb+node).son1, SENull, SENull];
        item =>  Exp[(tb+node).son2];
        ENDCASE =>  ErrorDefs.error[unimplemented];
      END;
    ENDCASE => NULL;
  RETURN
  END;

Interval: PROCEDURE [t: TreeLink] =
  BEGIN
  node: TreeIndex = GetNode[t];
  Exp[(tb+node).son1];  Exp[(tb+node).son2];  RETURN
  END;

Range: PROCEDURE [t: TreeLink] =
  BEGIN
  node: TreeIndex;
  WITH t SELECT FROM
    subtree =>
      BEGIN  node ← index;
      SELECT (tb+node).name FROM
        subrangeTC =>
          BEGIN
          TypeExp[(tb+node).son1, SENull, SENull]; Interval[(tb+node).son2];
          END;
        IN [intOO .. intCC] => Interval[t];
        ENDCASE => TypeExp[t, SENull, SENull];
      END;
    ENDCASE => TypeExp[t, SENull, SENull];
  RETURN
  END;
```

```
END.
```