```
-- file Pass3D.Mesa
-- last modified by Satterthwaite, August 2, 1978  12:10 PM

DIRECTORY
  ComData: FROM "comdata"
    USING [
      bodyIndex, idANY, idCARDINAL, mainBody, mainCtx, seAnon, textIndex,
      typeINTEGER, typeSTRING, xref],
  CopierDefs: FROM "copierdefs"
    USING [
      CopyUnion, CreateFileTable, EnterFile, FillModule, LocateTables,
      UnknownModule],
  ErrorDefs: FROM "errordefs" USING [error, errorhti, errorsei, errortree],
  P3Defs: FROM "p3defs"
    USING [
      CircuitCheck, CircuitSignal, P3Mark,
      CheckDisjoint, CompleteRecord, Exp, FindSe, GetDirectoryIds,
      PopCtx, PushCtx, RConst, RecordReference, ReplaceCtx, Rhs, RPop, RType,
      SearchCtxList, TopCtx, TreeStringValue,
      AmbiguousIdentifier, UndeclaredIdentifier],
  SymbolTableDefs: FROM "symboltabledefs"
    USING [SetSymbolCacheSize, SymbolCacheSize],
  SymDefs: FROM "symdefs"
    USING [setype, ctxtype, mdtype, bodytype,
      SERecord, CTXRecord, BodyRecord,
      HTIndex, SEIndex, ISEIndex, CSEIndex, recordCSEIndex,
      CTXIndex, includedCTXIndex, BTIndex, CBTIndex,
      HTNull, SENull, ISENull, CTXNull, BTNull, CBTNull,
      codeINTEGER, typeANY, typeTYPE],
  SymTabDefs: FROM "symtabdefs"
    USING [
      CtxEntries, makenonctxse, NextSe, NormalType, TransferTypes, TypeForm,
      UnderType, XferMode],
  TableDefs: FROM "tabledefs" USING [TableBase, TableNotifier, Allocate, TableBounds],
  TreeDefs: FROM "treedefs"
    USING [treetype,
      TreeIndex, TreeLink, TreeMap, TreeScan,
      empty, nullTreeIndex,
      GetNode, listlength, scanlist, testtree, updatelist];

Pass3D: PROGRAM
    IMPORTS
        CopierDefs, ErrorDefs, P3Defs,
        SymbolTableDefs, SymTabDefs, TableDefs, TreeDefs,
        dataPtr: ComData
    EXPORTS P3Defs =
  BEGIN
  OPEN TreeDefs, SymTabDefs, SymDefs, P3Defs;

  tb: TableDefs.TableBase;       -- tree base address (local copy)
  seb: TableDefs.TableBase;      -- se table base address (local copy)
  ctxb: TableDefs.TableBase;     -- context table base address (local copy)
  mdb: TableDefs.TableBase;      -- module table base address (local copy)
  bb: TableDefs.TableBase;       -- body table base address (local copy)

  DeclNotify: PUBLIC TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever table area is repacked
    tb ← base[treetype];
    seb ← base[setype];  ctxb ← base[ctxtype];  mdb ← base[mdtype];
    bb ← base[bodytype];  RETURN
    END;


 -- signals for loop detection

  CheckTypeLoop: PUBLIC CircuitCheck = CODE;
  LogTypeLoop: PUBLIC CircuitSignal = CODE;

  CheckExprLoop: PUBLIC CircuitCheck = CODE;
  LogExprLoop: PUBLIC CircuitSignal = CODE;


 -- declaration processing

  DeclItem: PUBLIC TreeScan =
    BEGIN
```

```
node: TreeIndex = GetNode[t];
expNode: TreeIndex;
type: SEIndex;
eqFlag, constFlag: BOOLEAN;
v: WORD;

ExpInit: PROCEDURE [t: TreeLink, type: SEIndex] RETURNS [val: TreeLink] =
  BEGIN
  val ← Rhs[t, TargetType[UnderType[type]]];
  constFlag ← eqFlag AND RConst[];  RPop[];
  v ← 1;  RETURN
  END;

saveIndex: CARDINAL = dataPtr.textIndex;
  BEGIN
  ENABLE
    BEGIN
    CheckTypeLoop, CheckExprLoop => IF loopNode=node THEN RESUME [TRUE];
    LogTypeLoop, LogExprLoop => IF loopNode=node THEN RESUME
    END;
  IF (tb+node).mark = P3Mark THEN RETURN;    -- already processed
  (tb+node).mark ← P3Mark;
  dataPtr.textIndex ← (tb+node).info;
  IF dataPtr.xref THEN scanlist[(tb+node).son1, RecordId];
  IF testtree[(tb+node).son2, modeTC]
    THEN
      BEGIN  OPEN (tb+node);
      son3 ← TypeExp[son3];  type ← typeTYPE;
      DefineTypeSe[son1, TypeForTree[son3]];
      END
    ELSE
      BEGIN  OPEN (tb+node);
      son2 ← TypeExp[son2];  type ← TypeForTree[son2];
      IF son3 = empty
        THEN  BEGIN  eqFlag ← constFlag ← FALSE;  v ← 0  END
        ELSE
          BEGIN
          IF dataPtr.xref THEN scanlist[son1, RecordLhs];
          eqFlag ← attr1;
          WITH son3 SELECT FROM
            subtree =>
              BEGIN  expNode ← index;
              SELECT (tb+expNode).name FROM
                body =>
                  BEGIN  -- defer processing of bodies (see Body)
                  SELECT XferMode[type] FROM
                    procedure, program => NULL;
                    ENDCASE =>
                      IF TypeForm[type] # definition
                        THEN ErrorDefs.error[bodyType];
                  v ← IF (constFlag ← eqFlag) THEN 0 ELSE 1;
                  END;
                inline =>
                  BEGIN
                  IF XferMode[type] # procedure OR ~eqFlag
                    THEN ErrorDefs.error[inlineType];
                  (tb+expNode).son1 ←
                        updatelist[(tb+expNode).son1, InlineOp];
                  constFlag ← eqFlag;   v ← 0;
                  END;
                apply =>
                  IF (tb+expNode).son1 # empty
                   OR UnderType[type] # dataPtr.typeSTRING
                   OR listlength[(tb+expNode).son2] # 1
                    THEN  son3 ← ExpInit[son3, type]
                    ELSE
                      BEGIN  (tb+expNode).name ← stringinit;
                      (tb+expNode).info ← dataPtr.typeSTRING;
                      (tb+expNode).son2 ← Rhs[
                            (tb+expNode).son2, dataPtr.typeINTEGER];
                      IF ~RConst[]
                        THEN ErrorDefs.errortree[
                            nonConstant, (tb+expNode).son2];
                      RPop[];  constFlag ← FALSE;  v ← 1;
                      END;
                signalinit =>  v ← IF (constFlag←eqFlag) THEN 0 ELSE 1;
```

```
                ENDCASE =>  son3 ← ExpInit[son3, type];
              END;
            ENDCASE =>  son3 ← ExpInit[son3, type];
          END;
        DefineSe[(tb+node).son1, type, eqFlag, constFlag, v];
        END;
    END;
  dataPtr.textIndex ← saveIndex;  RETURN
  END;


InlineOp: TreeMap =
  BEGIN

  EvalConst: TreeMap =
    BEGIN
    v ← Rhs[t, dataPtr.typeINTEGER];
    IF ~RConst[] THEN ErrorDefs.errortree[nonConstant, v];
    RPop[];  RETURN
    END;

  RETURN [updatelist[t, EvalConst]]
  END;


RecordId: TreeScan =
  BEGIN
  WITH t SELECT FROM
    symbol =>  RecordReference[index, declaration];
    ENDCASE;
  RETURN
  END;

RecordLhs: TreeScan =
  BEGIN
  WITH t SELECT FROM
    symbol =>  RecordReference[index, lhs];
    ENDCASE;
  RETURN
  END;


DefineSe: PROCEDURE [t: TreeLink, type: SEIndex, fixed, constant: BOOLEAN, info: WORD] =
  BEGIN

  UpdateSe: TreeScan =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        (seb+sei).idtype ← type;
        (seb+sei).mark3 ← TRUE;  (seb+sei).writeonce ← fixed;
        (seb+sei).constant ← constant;  (seb+sei).idinfo ← info;
        END;
      ENDCASE =>  ERROR;
    RETURN
    END;

  scanlist[t, UpdateSe];  RETURN
  END;


DefineTypeSe: PROCEDURE [t: TreeLink, info: SEIndex] =
  BEGIN
  first: BOOLEAN ← TRUE;

  UpdateSe: TreeScan =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        (seb+sei).idtype ← typeTYPE;  (seb+sei).mark3 ← TRUE;
        (seb+sei).writeonce ← (seb+sei).constant ← TRUE;
        (seb+sei).idinfo ← info;
```

```
        IF first THEN  BEGIN  info ← sei;  first ← FALSE  END;
        END;
      ENDCASE => ERROR;
    RETURN
    END;

  scanlist[t, UpdateSe];   RETURN
  END;


BumpCount: PUBLIC PROCEDURE [sei: ISEIndex] =
  BEGIN  OPEN (seb+sei);
  IF idtype # typeTYPE AND
   mark3 AND (~mark4 OR (ctxb+ctxnum).ctxType = imported)
    THEN  idinfo ← idinfo + 1;
  RETURN
  END;


ResolveId: PROCEDURE [hti: HTIndex, ctx: CTXIndex] RETURNS [sei: ISEIndex] =
  BEGIN
  currentCtx: CTXIndex = TopCtx[];
  IF ctx = currentCtx
    THEN  sei ← FindSe[hti].symbol
    ELSE
      BEGIN  PopCtx[];
      sei ← ResolveId[hti, ctx];  PushCtx[currentCtx];
      END;
  RETURN
  END;

ResolveReference: PUBLIC PROCEDURE [sei: ISEIndex] =
  BEGIN
  currentCtx: CTXIndex = TopCtx[];
  IF (seb+sei).ctxnum = currentCtx
    THEN DeclItem[TreeLink[subtree[index: (seb+sei).idvalue]]]
    ELSE
      BEGIN  PopCtx[];
      ResolveReference[sei];  PushCtx[currentCtx];
      END;
  RETURN
  END;


-- include module processing

DirectoryScan: PUBLIC PROCEDURE [t: TreeLink] RETURNS [nLists: CARDINAL] =
  BEGIN

  FileEntry: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    hti: HTIndex;
    hti ← CopierDefs.EnterFile[TreeStringValue[(tb+node).son2]];
    (tb+node).son2 ← TreeLink[hash[index: hti]];
    IF (tb+node).son3 # empty THEN  nLists ← nLists+1;
    RETURN
    END;

  n: CARDINAL = listlength[t];
  nLists ← 0;
  CopierDefs.CreateFileTable[n];
  IF n # 0
    THEN  BEGIN  scanlist[t, FileEntry];  CopierDefs.LocateTables[n]  END;
  RETURN
  END;


DirectoryItem: PUBLIC TreeScan =
  BEGIN
  node: TreeIndex = GetNode[t];
  saveIndex: CARDINAL = dataPtr.textIndex;
  saveSize: CARDINAL = SymbolTableDefs.SymbolCacheSize[];
  dataPtr.textIndex ← (tb+node).info;
  -- clear the symbol cache
    SymbolTableDefs.SetSymbolCacheSize[0];
```

```
    SymbolTableDefs.SetSymbolCacheSize[256];
  IF dataPtr.xref THEN RecordId[(tb+node).son1];
  WITH id: (tb+node).son1 SELECT FROM
    symbol =>
      BEGIN
      WITH (tb+node).son2 SELECT FROM
        hash =>  CopierDefs.FillModule[id.index, index
                  !CopierDefs.UnknownModule =>
                    BEGIN  ErrorDefs.errorhti[moduleId, hti];  RESUME  END];
        ENDCASE => ERROR;
      GetDirectoryIds[t];
      END;
    ENDCASE => ERROR;
  -- restore symbol caching
    SymbolTableDefs.SetSymbolCacheSize[saveSize];
  dataPtr.textIndex ← saveIndex;  RETURN
  END;


ImportItem: PUBLIC TreeScan =
  BEGIN
  node: TreeIndex = GetNode[t];
  type, vType: CSEIndex;
  ctx: CTXIndex;
  includedCtx: includedCTXIndex;
  const: BOOLEAN;
  saveIndex: CARDINAL = dataPtr.textIndex;
  dataPtr.textIndex ← (tb+node).info;
  IF dataPtr.xref THEN RecordId[(tb+node).son1];
  (tb+node).son2 ← Exp[(tb+node).son2, typeANY];
  vType ← RType[];  const ← RConst[];  RPop[];
  WITH v: (seb+vType) SELECT FROM
    definition =>
      WITH c: (ctxb+v.defCtx) SELECT FROM
        included =>
          BEGIN
          includedCtx ← LOOPHOLE[v.defCtx];
          ctx ← TableDefs.Allocate[ctxtype, SIZE[imported CTXRecord]];
          (ctxb+ctx)↑ ← CTXRecord[
                        sn: snNil,
                        selist: ISENull,
                        ctxlevel: c.ctxlevel,
                        extension: imported[includeLink: includedCtx]];
          type ← makenonctxse[SIZE[definition constructor SERecord]];
          (seb+type)↑ ← SERecord[mark3: TRUE, mark4: TRUE,
                        sebody: constructor[definition[
                          nGfi: v.nGfi,
                          defCtx: ctx]]];
          IF ~(tb+node).attr1
            THEN ReplaceCtx[old: includedCtx, new: ctx];
          END;
        ENDCASE =>
          BEGIN  type ← typeANY;
          ErrorDefs.errortree[notPortable, (tb+node).son2];
          END;
    transfer =>
      BEGIN
      IF v.mode # program
        THEN ErrorDefs.errortree[notPortable, (tb+node).son2];
      WITH (tb+node).son1 SELECT FROM
        symbol => (seb+index).writeonce ← TRUE;
        ENDCASE => ERROR;
      type ← MakePointerType[MakeFrameRecord[(tb+node).son2], typeANY];
      const ← FALSE;
      END;
    ENDCASE =>
      BEGIN
      IF vType # typeANY
        THEN ErrorDefs.errortree[typeClash, (tb+node).son2];
      type ← typeANY;
      END;
  DefineSe[(tb+node).son1, type, TRUE, const, 0];
  dataPtr.textIndex ← saveIndex;  RETURN
  END;

ExportId: PUBLIC TreeMap =
```

```
    BEGIN
    type: CSEIndex;
    saveXref: BOOLEAN = dataPtr.xref;
    dataPtr.xref ← FALSE;
    v ← Exp[t, typeANY];  type ← RType[];  RPop[];
    WITH d: (seb+type) SELECT FROM
      definition =>
        BEGIN
        WITH (ctxb+d.defCtx) SELECT FROM
          included => (mdb+ctxmodule).mdExported ← TRUE;
          ENDCASE => ErrorDefs.errortree[notPortable, v];
        END;
      ENDCASE =>
        BEGIN
        IF type # typeANY THEN ErrorDefs.errortree[typeClash, v];
        type ← typeANY;
        END;
    dataPtr.xref ← saveXref;  RETURN
    END;


  CheckTypeId: PROCEDURE [sei: ISEIndex] RETURNS [BOOLEAN] =
    BEGIN
    node: TreeIndex;
    IF (seb+sei).mark3 THEN RETURN [(seb+sei).idtype = typeTYPE];
    node ← (seb+sei).idvalue;
    RETURN [node = nullTreeIndex OR testtree[(tb+node).son2, modeTC]]
    END;

  TypeSymbol: PROCEDURE [sei: ISEIndex] RETURNS [val: TreeLink] =
    BEGIN
    declNode: TreeIndex;
    saveIndex: CARDINAL;
    entryIndex: CARDINAL = dataPtr.textIndex;
    IF ~(seb+sei).mark3
      THEN
        BEGIN
        ENABLE
          LogTypeLoop =>
            BEGIN  saveIndex ← dataPtr.textIndex;
            dataPtr.textIndex ← entryIndex;
            ErrorDefs.errorsei[circularType, sei];
            dataPtr.textIndex ← saveIndex;
            END;
        declNode ← (seb+sei).idvalue;
        IF (tb+declNode).mark # P3Mark
          THEN ResolveReference[sei]
          ELSE
            IF SIGNAL CheckTypeLoop[declNode]
              THEN SIGNAL LogTypeLoop[declNode];
        END;
    IF CheckTypeId[sei]
      THEN  val ← TreeLink[symbol[index: sei]]
      ELSE
        BEGIN
        ErrorDefs.errorsei[nonTypeId, sei];
        val ← TreeLink[symbol[index: dataPtr.idANY]];
        END;
    IF dataPtr.xref THEN RecordReference[sei, mention];
    RETURN
    END;


  PushArgCtx: PUBLIC PROCEDURE [sei: recordCSEIndex] =
    BEGIN
    IF sei # SENull THEN PushCtx[(seb+sei).fieldctx];  RETURN
    END;

  PopArgCtx: PUBLIC PROCEDURE [sei: recordCSEIndex] =
    BEGIN
    IF sei # SENull THEN PopCtx[];  RETURN
    END;


  TypeExp: PUBLIC PROCEDURE [typeExp: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
```

```
    iSei: ISEIndex;
    WITH typeExp SELECT FROM
      hash =>
        BEGIN
        [symbol: iSei] ← FindSe[index
          !UndeclaredIdentifier =>
            BEGIN
            IF hti # HTNull THEN ErrorDefs.errorhti[unknownId, hti];
            RESUME [dataPtr.idANY]
            END;
           AmbiguousIdentifier =>
            BEGIN
            ErrorDefs.errorhti[ambiguousId, (seb+sei).htptr];
            RESUME [dataPtr.idANY]
            END];
        val ← TypeSymbol[iSei];
        END;
      symbol =>  val ← TypeSymbol[index];
      subtree =>
        BEGIN
        node: TreeIndex = index;
        SELECT (tb+node).name FROM
          discrimTC =>
            BEGIN  OPEN (tb+node);
            son1 ← TypeExp[son1];
            WITH son2 SELECT FROM
              hash =>
                iSei ← SelectVariantType[ConsType[TypeForTree[son1]], index];
              ENDCASE =>  ERROR;
            info ← iSei;  son2 ← TreeLink[symbol[index: iSei]];
            END;
          dot =>
            BEGIN  OPEN (tb+node);
            found: BOOLEAN;
            nDerefs: CARDINAL;
            sei: SEIndex;
            subType: CSEIndex;
            ctx: CTXIndex;
            son1 ← Exp[son1, typeANY];
            subType ← RType[];   RPop[];
            WITH son2 SELECT FROM
              hash =>
                BEGIN
                nDerefs ← 0;
                  DO
                  WITH t: (seb+subType) SELECT FROM
                    definition =>
                      BEGIN  ctx ← t.defCtx;  GO TO search  END;
                    record =>
                      BEGIN  ctx ← t.fieldctx;  GO TO search  END;
                    pointer =>
                      BEGIN
                      IF (nDerefs ← nDerefs+1) > 255 THEN GO TO fail;
                      t.dereferenced ← TRUE;  sei ← t.pointedtotype;
                      END;
                    long =>  sei ← t.rangetype;
                    subrange =>  sei ← t.rangetype;
                    ENDCASE =>  GO TO fail;
                  subType ← ConsType[sei];
                  REPEAT
                    fail =>  found ← FALSE;
                    search =>
                      [found, iSei] ← SearchCtxList[index, ctx];
                  ENDLOOP;
                IF ~found
                  THEN
                    BEGIN  iSei ← dataPtr.idANY;
                    ErrorDefs.errorhti[unknownField, index];
                    END;
                name ← cdot;  info ← iSei;  son2 ← TypeSymbol[iSei];
                END;
              ENDCASE =>  ERROR;
            END;
          frameTC =>
            BEGIN  OPEN (tb+node);
            son1 ← Exp[son1, typeANY];  RPop[];
```

```
    info ← MakeFrameRecord[son1];
    END;
  ENDCASE =>
    BEGIN  OPEN (tb+node);
    subType: SEIndex;
    consType: CSEIndex;
    sei: CSEIndex = info;
    WITH (seb+sei) SELECT FROM
      enumerated =>  IF dataPtr.xref THEN scanlist[son1, RecordId];
      record =>
        BEGIN
        PushCtx[fieldctx];  scanlist[son1, DeclItem];
        PopCtx[];
        END;
      pointer =>
        BEGIN
        son1 ← TypeExp[son1 !CheckTypeLoop => RESUME[FALSE]];
        pointedtotype ← TypeForTree[son1];
        END;
      array =>
        BEGIN
        IF son1 = empty
          THEN  subType ← dataPtr.idCARDINAL
          ELSE
            BEGIN
            son1 ← TypeExp[son1];  subType ← TypeForTree[son1];
            IF ~OrderedType[subType]
              THEN
                BEGIN  subType ← typeANY;
                ErrorDefs.error[nonOrderedType];
                END;
            END;
        indextype ← subType;
        son2 ← TypeExp[son2];  componenttype ← TypeForTree[son2];
        END;
      arraydesc =>
        BEGIN
        son1 ← TypeExp[son1 !CheckTypeLoop => RESUME[FALSE]];
        describedType ← TypeForTree[son1];
        IF TypeForm[describedType] # array
          THEN ErrorDefs.error[descriptor];
        END;
      transfer =>
        BEGIN
        ENABLE  CheckTypeLoop => RESUME[FALSE];
        IF inrecord # SENull AND outrecord # SENull
          THEN CheckDisjoint[
                (seb+inrecord).fieldctx, (seb+outrecord).fieldctx];
        PushArgCtx[inrecord];  scanlist[son1, DeclItem];
        PushArgCtx[outrecord];  scanlist[son2, DeclItem];
        PopArgCtx[outrecord];  PopArgCtx[inrecord];
        END;
      definition =>  defCtx ← dataPtr.mainCtx;
      union =>
        BEGIN
        tagType: CSEIndex;
        DeclItem[son1];
        (seb+tagsei).writeonce ← TRUE;
        tagType ← ConsType[(seb+tagsei).idtype];
          DO
          WITH (seb+tagType) SELECT FROM
            enumerated =>  EXIT;
            subrange =>  tagType ← ConsType[rangetype];
            ENDCASE =>
              BEGIN  ErrorDefs.errorsei[nonTagType, tagsei];
              tagType ← typeANY;  EXIT
              END;
          ENDLOOP;
        VariantList[son2, tagType];
        END;
      relative =>
        BEGIN
        vType: CSEIndex;
        son1 ← TypeExp[son1];  baseType ← TypeForTree[son1];
        IF (seb+NormalType[ConsType[baseType]]).typetag # pointer
          THEN ErrorDefs.error[relative];
```

```
            son2 ← TypeExp[son2];  offsetType ← TypeForTree[son2];
            vType ← ConsType[offsetType];  consType ← NormalType[vType];
            SELECT (seb+consType).typetag FROM
              pointer, arraydesc => NULL;
              ENDCASE =>
                BEGIN
                ErrorDefs.error[relative];  consType ← typeANY;
                END;
            IF (seb+ConsType[baseType]).typetag = long
             OR (seb+vType).typetag = long
              THEN consType ← MakeLongType[consType,ConsType[offsetType]];
            resultType ← consType;
            END;
          subrange =>
            BEGIN
            son1 ← TypeExp[son1];  subType ← TypeForTree[son1];
            rangetype ← subType;  consType ← ConsType[subType];
            SELECT TRUE FROM
              (TypeForm[consType] = pointer) =>
                Interval[son2, dataPtr.typeINTEGER, TRUE];
              OrderedType[consType] =>
                Interval[son2, consType, TRUE];
              ENDCASE =>
                BEGIN  ErrorDefs.error[nonOrderedType];
                Interval[son2, typeANY, TRUE];
                END;
            END;
          long =>
            BEGIN
            son1 ← TypeExp[son1];  subType ← TypeForTree[son1];
            rangetype ← subType;  consType ← ConsType[subType];
            WITH (seb+consType) SELECT FROM
              pointer, arraydesc => NULL;
              basic => IF code # codeINTEGER THEN ErrorDefs.error[long];
              ENDCASE =>  ErrorDefs.error[long];
            END;
          ENDCASE =>  ERROR;
        (seb+sei).mark3 ← TRUE;
        END;
      val ← typeExp;
      END;
    ENDCASE =>  ERROR;
  RETURN
  END;

VariantList: PROCEDURE [t: TreeLink, tagType: CSEIndex] =
  BEGIN

  DefineTag: TreeScan =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        (seb+sei).idvalue ← TagValue[(seb+sei).htptr, tagType];
        END;
      ENDCASE =>  ERROR;
    RETURN
    END;

  VariantItem: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    saveIndex: CARDINAL = dataPtr.textIndex;
    DeclItem[t];
    dataPtr.textIndex ← (tb+node).info;
    scanlist[(tb+node).son1, DefineTag];
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  scanlist[t, VariantItem];
  RETURN
  END;

TagValue: PROCEDURE [tag: HTIndex, tagType: CSEIndex] RETURNS [CARDINAL] =
```

```
    BEGIN
    matched: BOOLEAN;
    sei: ISEIndex;
    WITH (seb+tagType) SELECT FROM
      enumerated =>
        BEGIN
        [matched, sei] ← SearchCtxList[tag, valuectx];
        IF matched THEN RETURN [(seb+sei).idvalue];
        END;
      ENDCASE;
    ErrorDefs.errorhti[unknownTag, tag];  RETURN [0]
    END;


  SelectVariantType: PROCEDURE [type: CSEIndex, tag: HTIndex] RETURNS [sei: ISEIndex] =
    BEGIN
    matched: BOOLEAN;
    WITH (seb+type) SELECT FROM
      record =>
        BEGIN
        [matched, sei] ← SearchCtxList[tag, TagContext[type]];
        IF matched THEN RETURN [sei];
        END;
      ENDCASE;
    IF type # typeANY THEN ErrorDefs.errorhti[unknownVariant, tag];
    RETURN [dataPtr.idANY]
    END;

  UnionField: PUBLIC PROCEDURE [rSei: recordCSEIndex] RETURNS [ISEIndex] =
    BEGIN
    sei, root, next: ISEIndex;
    ctx: CTXIndex = (seb+rSei).fieldctx;
    repeated: BOOLEAN;
    IF (ctxb+ctx).ctxType = simple
      THEN
        FOR sei ← (ctxb+ctx).selist, next UNTIL sei = SENull
          DO
          next ← NextSe[sei];
          IF next = SENull THEN RETURN [sei];
          ENDLOOP
      ELSE
        BEGIN
        repeated ← FALSE;
          DO
          sei ← root ← (ctxb+ctx).selist;
            DO
            IF sei = SENull THEN EXIT;
            IF TypeForm[(seb+sei).idtype] = union THEN RETURN [sei];
            IF (sei ← NextSe[sei]) = root THEN EXIT;
            ENDLOOP;
          IF repeated THEN EXIT;
          CopierDefs.CopyUnion[(seb+rSei).fieldctx];  repeated ← TRUE;
          ENDLOOP;
        END;
    RETURN [dataPtr.seAnon]
    END;

  VariantUnionType: PUBLIC PROCEDURE [type: SEIndex] RETURNS [CSEIndex] =
    BEGIN
    vType: CSEIndex = UnderType[type];
    RETURN [WITH (seb+vType) SELECT FROM
      record =>
        IF variant
          THEN ConsType[TypeForSe[UnionField[LOOPHOLE[vType, recordCSEIndex]]]]
          ELSE typeANY,
      ENDCASE => typeANY]
    END;

  TagContext: PROCEDURE [type: CSEIndex] RETURNS [CTXIndex] =
    BEGIN
    subType: CSEIndex = VariantUnionType[type];
    RETURN [WITH (seb+subType) SELECT FROM
      union => casectx,
      ENDCASE => CTXNull]
    END;
```

```
TypeForTree: PUBLIC PROCEDURE [t: TreeLink] RETURNS [SEIndex] =
  -- N.B. assumes t evaluated by TypeExp or Exp
  BEGIN
  RETURN [WITH t SELECT FROM
    symbol => index,
    subtree =>
      SELECT (tb+index).name FROM
        cdot => TypeForTree[(tb+index).son2],
        ENDCASE => (tb+index).info,
    ENDCASE => typeANY]
  END;

TypeForSe: PROCEDURE [sei: ISEIndex] RETURNS [type: SEIndex] =
  BEGIN
  node: TreeIndex;
  t: TreeLink;
  IF (seb+sei).mark3 THEN RETURN [(seb+sei).idtype];
  node ← (seb+sei).idvalue;
  IF (tb+node).name # declitem THEN RETURN [typeTYPE];
  t ← (tb+node).son2;
  type ← WITH t SELECT FROM
    hash => ResolveId[index, (seb+sei).ctxnum],
    symbol => index,
    subtree => (tb+index).info,
    ENDCASE => typeANY;
  RETURN
  END;

ConsType: PROCEDURE [type: SEIndex] RETURNS [CSEIndex] =
  BEGIN
  sei, next: SEIndex;
  node: TreeIndex;
  next ← type;
    DO
    sei ← next;
    WITH (seb+sei) SELECT FROM
      id =>
        BEGIN
        IF ~CheckTypeId[LOOPHOLE[sei, ISEIndex]]
          THEN
            BEGIN
            IF sei # dataPtr.seAnon
              THEN ErrorDefs.errorsei[nonTypeId, LOOPHOLE[sei, ISEIndex]];
            RETURN [typeANY]
            END;
        IF mark3
          THEN  next ← idinfo
          ELSE
            BEGIN  node ← idvalue;
            next ← ResolveTreeType[IF (tb+node).name = declitem
              THEN (tb+node).son3
              ELSE (tb+node).son2, ctxnum];
            END;
        END;
      constructor => RETURN [LOOPHOLE[sei, CSEIndex]];
      ENDCASE;
    ENDLOOP;
  END;

ResolveTreeType: PROCEDURE [t: TreeLink, ctx: CTXIndex] RETURNS [type: SEIndex] =
  BEGIN
  node: TreeIndex;
  WITH t SELECT FROM
    hash =>    type ← ResolveId[index, ctx];
    symbol =>  type ← index;
    subtree =>
      BEGIN  node ← index;
      IF (tb+node).info # SENull
        THEN  type ← (tb+node).info
        ELSE
          SELECT (tb+node).name FROM
            discrimTC =>
              WITH (tb+node).son2 SELECT FROM
                hash =>
                  type ← SelectVariantType[
```

```
                    ConsType[ResolveTreeType[(tb+node).son1, ctx]],
                    index];
                ENDCASE => ERROR;
            ENDCASE => ERROR;
      END;
    ENDCASE => ERROR;
  RETURN
  END;


Bundling: PUBLIC PROCEDURE [type: CSEIndex] RETURNS [nLevels: CARDINAL] =
  BEGIN
  ctx: CTXIndex;
  nLevels ← 0;
    DO
    IF type = SENull THEN EXIT;
    WITH (seb+type) SELECT FROM
      record =>
        BEGIN
        IF ~unifield THEN EXIT;
        ctx ← fieldctx;
        WITH (ctxb+ctx) SELECT FROM
          included =>
            BEGIN
            IF privateFields AND ~(mdb+ctxmodule).mdshared THEN EXIT;
            IF ~ctxcomplete
              THEN CompleteRecord[LOOPHOLE[type, recordCSEIndex]];
            IF ~ctxcomplete THEN EXIT;
            END;
          ENDCASE;
        IF CtxEntries[fieldctx] # 1 OR variant THEN EXIT;
        nLevels ← nLevels + 1;
        type ← Unbundle[LOOPHOLE[type, recordCSEIndex]];
        END;
      ENDCASE => EXIT;
    ENDLOOP;
  RETURN
  END;

Unbundle: PUBLIC PROCEDURE [record: recordCSEIndex] RETURNS [CSEIndex] =
  BEGIN  OPEN (seb+record);
  RETURN [UnderType[(seb + (ctxb+fieldctx).selist).idtype]]
  END;


TargetType: PUBLIC PROCEDURE [type: CSEIndex] RETURNS [target: CSEIndex] =
  BEGIN
  target ← type;
    DO
    WITH (seb+target) SELECT FROM
      subrange =>  target ← UnderType[rangetype];
      ENDCASE =>  EXIT;
    ENDLOOP;
  RETURN [target]
  END;

CanonicalType: PUBLIC PROCEDURE [sType: CSEIndex] RETURNS [type: CSEIndex] =
  BEGIN
  type ← sType;
    DO
    WITH (seb+type) SELECT FROM
      subrange =>  type ← UnderType[rangetype];
      record =>
        IF Bundling[type] # 0
          THEN  type ← Unbundle[LOOPHOLE[type, recordCSEIndex]]
          ELSE RETURN;
      ENDCASE =>  RETURN
    ENDLOOP;
  END;


IdentifiedType: PUBLIC PROCEDURE [type: CSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  WITH (seb+type) SELECT FROM
    mode => RETURN [FALSE];
    definition => RETURN [FALSE];
```

```
    nil => RETURN [FALSE];
    record =>
      BEGIN
      IF variant AND ~comparable
        THEN [] ← UnionField[LOOPHOLE[type, recordCSEIndex]];
      RETURN [TRUE]
      END;
    ENDCASE => RETURN [TRUE]
  END;

OrderedType: PUBLIC PROCEDURE [type: SEIndex] RETURNS [BOOLEAN] =
  BEGIN
  sei: CSEIndex ← ConsType[type];
    DO
    WITH (seb+sei) SELECT FROM
      basic =>  RETURN [ordered];
      enumerated =>  RETURN [ordered];
      pointer => RETURN [ordered];
      relative =>  sei ← ConsType[offsetType];
      subrange =>  sei ← ConsType[rangetype];
      long, real =>  sei ← ConsType[rangetype];
      ENDCASE =>  RETURN [FALSE];
    ENDLOOP;
  END;


MakeLongType: PUBLIC PROCEDURE [rType: SEIndex, hint: CSEIndex] RETURNS [type: CSEIndex] =
  BEGIN
  WITH (seb+hint) SELECT FROM
    long =>
      IF UnderType[rangetype] = UnderType[rType] THEN RETURN [hint];
    ENDCASE;
  type ← makenonctxse[SIZE[long constructor SERecord]];
  (seb+type)↑ ← SERecord[mark3: TRUE, mark4: TRUE,
      sebody: constructor[long[rangetype: rType]]];
  RETURN
  END;

MakePointerType: PUBLIC PROCEDURE [cType: SEIndex, hint: CSEIndex] RETURNS [type: CSEIndex] =
  BEGIN
  WITH (seb+hint) SELECT FROM
    pointer =>
      IF ~ordered AND UnderType[pointedtotype] = UnderType[cType]
        THEN RETURN [hint];
    ENDCASE;
  type ← makenonctxse[SIZE[pointer constructor SERecord]];
  (seb+type)↑ ← SERecord[
      mark3: TRUE,
      mark4: TRUE,
      sebody: constructor[pointer[
          ordered: FALSE,
          readonly: FALSE,
          basing: FALSE,
          dereferenced: FALSE,
          pointedtotype: cType]]];
  RETURN
  END;


AllocFrameRecord: PROCEDURE [bti: CBTIndex, link: SEIndex] RETURNS [sei: recordCSEIndex] =
  BEGIN
  sei ← LOOPHOLE[makenonctxse[SIZE[linked record constructor SERecord]]];
  (seb+sei)↑ ← SERecord[
      mark3: TRUE,
      mark4: FALSE,
      sebody: constructor[record[
          machineDep: FALSE,
          unifield: FALSE,
          argument: FALSE,
          defaultFields: FALSE,
          fieldctx: (bb+bti).localCtx,
          length: 0 -- n*wordlength --,
          comparable: FALSE,
          privateFields: FALSE,
          lengthUsed: FALSE,
          monitored: (bb+bti).monitored,
```

```
          variant: FALSE,
          linkpart: linked[link]]]];
  RETURN
  END;

MakeFrameRecord: PUBLIC PROCEDURE [t: TreeLink] RETURNS [rSei: CSEIndex] =
  BEGIN
  bti: CBTIndex = XferBody[t];
  IF bti # BTNull
    THEN
      rSei ← AllocFrameRecord[bti, TransferTypes[(bb+bti).ioType].typeIn]
    ELSE
      BEGIN  ErrorDefs.error[nonTypeCons];  rSei ← typeANY  END;
  RETURN
  END;


XferBody: PUBLIC PROCEDURE [t: TreeLink] RETURNS [bti: CBTIndex] =
  BEGIN
  sei: ISEIndex;
  type: CSEIndex;
  WITH t SELECT FROM
    symbol =>
      BEGIN  sei ← index;
      type ← UnderType[(seb+sei).idtype];
      bti ← WITH (seb+type) SELECT FROM
        transfer =>
          IF ~(seb+sei).constant
            THEN CBTNull
            ELSE
              SELECT mode FROM
                program =>
                  IF (seb+sei).mark4
                    THEN (seb+sei).idinfo
                    ELSE dataPtr.mainBody,
                procedure =>
                  IF sei = (bb+dataPtr.bodyIndex).id
                    THEN dataPtr.bodyIndex
                    ELSE CBTNull,
                ENDCASE => CBTNull,
        ENDCASE =>  CBTNull;
      END;
    ENDCASE =>  bti ← CBTNull;
  RETURN
  END;

XferForFrame: PUBLIC PROCEDURE [ctx: CTXIndex] RETURNS [CSEIndex] =
  BEGIN
  bti: BTIndex;
  btLimit: BTIndex = LOOPHOLE[TableDefs.TableBounds[bodytype].size];
  bti ← FIRST[BTIndex];
  UNTIL bti = btLimit
    DO
    WITH entry: (bb+bti) SELECT FROM
      Callable =>
        BEGIN
        IF entry.localCtx = ctx THEN RETURN [UnderType[entry.ioType]];
        bti ← bti + (WITH  entry SELECT FROM
                  Inner => SIZE[Inner Callable BodyRecord],
                  ENDCASE => SIZE[Outer Callable BodyRecord]);
        END;
      ENDCASE => bti ← bti + SIZE[Other BodyRecord];
    ENDLOOP;
  ERROR
  END;


Interval: PUBLIC PROCEDURE [t: TreeLink, type: CSEIndex, constant: BOOLEAN] =
  BEGIN
  node: TreeIndex = GetNode[t];
    BEGIN  OPEN (tb+node);
    type ← TargetType[type];
    son1 ← Rhs[son1, type];
    IF constant AND ~RConst[]
      THEN ErrorDefs.errortree[nonConstant, son1];
    RPop[];
```

```
    son2 ← Rhs[son2, type];
    SELECT (seb+type).typetag FROM
      long => BEGIN attr1 ← TRUE; attr2 ← FALSE END;
      real => attr1 ← attr2 ← TRUE;
      ENDCASE => attr1 ← attr2 ← FALSE;
    IF constant AND ~RConst[]
      THEN ErrorDefs.errortree[nonConstant, son2];
    RPop[];
    END;
  RETURN
  END;


Sharing: PUBLIC TreeScan =
  BEGIN
  v: TreeLink = Exp[t, typeANY];
  type: CSEIndex = RType[];
  ctx: CTXIndex;
  sei: ISEIndex;
  ctx ← CTXNull;
  WITH (seb+type) SELECT FROM
    definition => ctx ← defCtx;
    transfer =>
      WITH v SELECT FROM
        symbol =>
          BEGIN sei ← index;
          IF (seb+sei).mark4 AND (seb+sei).constant AND mode = program
            THEN ctx ← (bb+LOOPHOLE[(seb+sei).idinfo, CBTIndex]).localCtx;
          END;
        ENDCASE;
    ENDCASE;
  IF ctx # CTXNull
    THEN
      WITH (ctxb+ctx) SELECT FROM
        included => (mdb+ctxmodule).mdshared ← TRUE;
        ENDCASE
    ELSE IF type # typeANY THEN ErrorDefs.errortree[typeClash, v];
  RPop[]; RETURN
  END;


END.
```