```
-- file Pass3S.Mesa
-- last modified by Satterthwaite, July 16, 1978  10:04 AM

DIRECTORY
  ComData: FROM "comdata"
    USING [
      bodyIndex, idUNWIND, mainBody, monitored, ownSymbols, seAnon, stopping,
      textIndex, typeBOOLEAN, typeCONDITION, typeINTEGER, typeLOCK],
  ErrorDefs: FROM "errordefs" USING [error, errorsei, errortree],
  LitDefs: FROM "litdefs"
    USING [STIndex, FindStringLiteral, StringLiteralReference],
  Pass3: FROM "pass3" USING [implicitTree, implicitType, lockHeld, lockNode],
  P3Defs: FROM "p3defs"
    USING [
      Apply, Assignment, BumpCount, BumpFieldRefs, CanonicalType,
      CheckDisjoint, CloseBase, CountTreeIds, DeclItem, Discrimination, Exp,
      Extract, IdentifiedType, Interval, LambdaApply, LongPath,
      MakeFrameRecord, MakePointerType, MatchFields, OpenBase,
      OperandInternal, OperandLhs, OperandType, OrderedType, PopCtx,
      PopArgCtx, PushArgCtx, PushCtx, RConst, Rhs, RPop, RPush, RType,
      SearchCtxList, TargetType, TypeExp, TypeForTree, VoidExp,
      XferForFrame, XferBody],
  StringDefs: FROM "stringdefs" USING [SubString, SubStringDescriptor],
  SymDefs: FROM "symdefs"
    USING [setype, ctxtype, mdtype, bodytype,
      ContextLevel, SERecord,
      ISEIndex, CSEIndex, recordCSEIndex, MDIndex, BTIndex, CBTIndex,
      HTNull, SENull, ISENull, CSENull, recordCSENull, BTNull,
      1G, OwnMdi, typeANY],
  SymTabDefs: FROM "symtabdefs"
    USING [
      EnterString, makenonctxse, NextSe, NormalType, SubStringForHash,
      TransferTypes, TypeRoot, UnderType, XferMode],
  TableDefs: FROM "tabledefs" USING [TableBase, TableNotifier],
  TreeDefs: FROM "treedefs"
    USING [treetype,
      NodeName, TreeIndex, TreeLink, TreeMap, TreeScan,
      empty, nullid, nullTreeIndex,
      freenode, freetree, GetNode, makelist, maketree, mlpop, mlpush,
      pushsymtree, pushtree, reversescanlist, scanlist, setattr, setinfo,
      testtree, updatelist],
  TypePackDefs: FROM "typepackdefs" USING [AssignableTypes];

Pass3S: PROGRAM
    IMPORTS
        ErrorDefs, LitDefs, P3Defs, SymTabDefs, TreeDefs, TypePackDefs,
        dataPtr: ComData, passPtr: Pass3
    EXPORTS P3Defs =
  BEGIN
  OPEN SymTabDefs, SymDefs, P3Defs, TreeDefs;

  InsertCatchLabel: SIGNAL [catchSeen: BOOLEAN] = CODE;

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

  StmtNotify: 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;


 -- bodies and blocks

  BodyState: TYPE = RECORD[
    bodyNode: TreeIndex,                -- current body
    impliedReturn: BOOLEAN,             -- true if return with no args
    inputRecord: recordCSEIndex,        -- input record for current body
    returnRecord: recordCSEIndex,       -- return record for current body
    entry: BOOLEAN,                     -- set for entry procedures
```

```
    labelList: TreeLink,                -- list of accessible labels
    loopDepth: CARDINAL,                -- depth of loop nesting

    catchDepth: CARDINAL,               -- depth of catch phrase nesting
    unwindEnabled: BOOLEAN,             -- set iff in scope of unwind
    resumeFlag: BOOLEAN,                -- set iff a resume is legal
    resumeRecord: recordCSEIndex];      -- for current catch phrase

  current: BodyState;


  BodyList: PUBLIC 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
    item: TreeIndex;    -- item designates a declitem with body as last son
    saved: BodyState = current;
    saveIndex: CARDINAL = dataPtr.textIndex;
    saveBodyIndex: CBTIndex = dataPtr.bodyIndex;
    saveLockHeld: BOOLEAN = passPtr.lockHeld;
    node: TreeIndex;
    lockVar: ISEIndex;
    lockBit: BOOLEAN;
    inRecord, outRecord: recordCSEIndex;
    dataPtr.bodyIndex ← bti;
    WITH (bb+bti).info SELECT FROM
      Internal =>
        BEGIN
        dataPtr.textIndex ← sourceIndex;
        item ← bodyTree;
        current.bodyNode ← node ← GetNode[(tb+item).son3];
        bodyTree ← node;
        END;
      ENDCASE => ERROR;
    current.entry ← (bb+bti).entry ← (tb+node).attr1;
    (bb+bti).internal ← (tb+node).attr2;
    passPtr.lockHeld ← (bb+bti).entry OR (bb+bti).internal;
    (bb+bti).ioType ← TypeForTree[(tb+item).son2];
    [inRecord, outRecord] ← TransferTypes[(bb+bti).ioType];
    PushArgCtx[current.inputRecord ← inRecord];
    PushArgCtx[current.returnRecord ← outRecord];
    -- initialize computed attributes
      current.impliedReturn ← FALSE;
      current.labelList ← empty;  current.loopDepth ← 0;
      current.catchDepth ← 0;  current.unwindEnabled ← FALSE;
      current.resumeRecord ← recordCSENull;  current.resumeFlag ← FALSE;
    IF current.entry
      THEN
        BEGIN
        IF (lockVar ← FindLockParams[].actual) # SENull
          THEN
            BEGIN
            lockBit ← (seb+lockVar).writeonce; (seb+lockVar).writeonce ← TRUE;
            END;
        (tb+node).son4 ← CopyLock[];
        END;
      BEGIN
      ENABLE
        InsertCatchLabel => BEGIN  ErrorDefs.error[catchLabel];  RESUME  END;
      scanlist[(tb+node).son1, OpenItem];
      IF inRecord # SENull THEN
        CheckDisjoint[(seb+inRecord).fieldctx, (bb+bti).localCtx];
      IF outRecord # SENull THEN
```

```
        CheckDisjoint[(seb+outRecord).fieldctx, (bb+bti).localCtx];
      PushCtx[(bb+bti).localCtx];
      IF bti = dataPtr.mainBody AND dataPtr.monitored
        THEN
          BEGIN
          scanlist[(tb+passPtr.lockNode).son1, DeclItem];
          PushCtx[(tb+passPtr.lockNode).info];
          (tb+passPtr.lockNode).son2 ← LockVar[(tb+passPtr.lockNode).son2];
          PopCtx[];
          END;
      scanlist[(tb+node).son2, DeclItem];
      END;
    (tb+node).son3 ← updatelist[(tb+node).son3, Stmt
        ! InsertCatchLabel => IF catchSeen THEN RESUME];
    BodyList[(bb+bti).firstSon];
    PopCtx[];
    reversescanlist[(tb+node).son1, CloseItem];
    (bb+bti).stopping ← dataPtr.stopping AND (bb+bti).level = 1G;
    PopArgCtx[outRecord];  PopArgCtx[inRecord];
--  (tb+node).attr2 ← current.impliedReturn;     ** find another field
    IF current.entry AND lockVar # SENull
      THEN (seb+lockVar).writeonce ← lockBit;
    current ← saved;  passPtr.lockHeld ← saveLockHeld;
    dataPtr.textIndex ← saveIndex;  dataPtr.bodyIndex ← saveBodyIndex;
    RETURN
    END;

  Block: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    bti: BTIndex = info;
    saveIndex: CARDINAL = dataPtr.textIndex;
    WITH (bb+bti).info SELECT FROM
      Internal => dataPtr.textIndex ← sourceIndex;
      ENDCASE => ERROR;
    PushCtx[(bb+bti).localCtx];
    scanlist[son1, DeclItem
      ! InsertCatchLabel => BEGIN  ErrorDefs.error[catchLabel];  RESUME  END];
    son2 ← updatelist[son2, Stmt];
    BodyList[(bb+bti).firstSon];
    PopCtx[];
    dataPtr.textIndex ← saveIndex;  RETURN
    END;


  -- statements

  markCatch: BOOLEAN;   -- reset in Stmt, set in CatchPhrase

  Stmt: PROCEDURE [stmt: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    node: TreeIndex;
    saveIndex: CARDINAL = dataPtr.textIndex;
    saveMark: BOOLEAN = markCatch;
    IF stmt = empty THEN RETURN [empty];
    WITH stmt SELECT FROM
      subtree =>
        BEGIN  node ← index;
        dataPtr.textIndex ← (tb+node).info;
        val ← stmt;  markCatch ← FALSE;          -- the defaults
        SELECT (tb+node).name FROM
          assign =>  BEGIN  Assignment[node];  RPop[]  END;
          extract =>  Extract[node];
          apply =>
            BEGIN
            Apply[node, typeANY, TRUE];
            IF (tb+node).name = wait
              THEN  ErrorDefs.errortree[typeClash, (tb+node).son1];
            CheckVoid[];
            END;
          block =>  Block[node];
          ifstmt =>
            BEGIN  OPEN (tb+node);
            son1 ← Rhs[son1, dataPtr.typeBOOLEAN];  RPop[];
            son2 ← updatelist[son2, Stmt];
            son3 ← updatelist[son3, Stmt];
            END;
```

```
casestmt => Case[node, Stmt];
bindstmt => Discrimination[node, Stmt];
dostmt => DoStmt[node];
return => Return[node];
label =>
  BEGIN OPEN (tb+node);
  saveList: TreeLink = current.labelList;
  InsertLabels[son2];
  son1 ← updatelist[son1, Stmt];
  DeleteLabels[mark: saveList];
  scanlist[son2, LabelItem];
  END;
goto => ValidateLabel[(tb+node).son1];
exit, loop => IF current.loopDepth = 0 THEN ErrorDefs.error[exit];
signal, error, xerror, start, join, wait =>
  BEGIN
  mlpush[SELECT (tb+node).name FROM
    start => Start[node],
    join => Join[node],
    wait => Wait[node],
    ENDCASE => Signal[node]];
  setinfo[dataPtr.textIndex]; val ← mlpop[];
  CheckVoid[];
  END;
resume => Resume[node];
continue, retry => SIGNAL InsertCatchLabel[catchSeen: FALSE];
restart => val ← FrameXfer[node];
stop =>
  BEGIN
  IF dataPtr.bodyIndex # dataPtr.mainBody OR current.catchDepth # 0
   OR current.returnRecord # SENull
    THEN ErrorDefs.error[misplacedStop];
  IF (tb+node).son1 # empty THEN [] ← CatchPhrase[(tb+node).son1];
  dataPtr.stopping ← TRUE;
  END;
notify, broadcast =>
  BEGIN OPEN (tb+node);
  type: CSEIndex;
  IF ~passPtr.lockHeld THEN ErrorDefs.error[misplacedMonitorRef];
  son1 ← Exp[son1, typeANY];
  IF ~OperandLhs[son1] THEN ErrorDefs.errortree[nonLHS, son1];
  type ← RType[]; RPop[];
  IF type # dataPtr.typeCONDITION
    THEN ErrorDefs.errortree[typeClash, son1];
  END;
dst, lst, lstf =>
  BEGIN OPEN (tb+node);
  v: TreeLink;
  v ← son1 ← Exp[son1, typeANY]; RPop[];
    DO
    WITH v SELECT FROM
      symbol =>
        BEGIN
        IF (seb+index).constant OR
          (name = dst AND (seb+index).writeonce) THEN GO TO fail;
        EXIT
        END;
      subtree =>
        BEGIN
        IF (tb+index).name # dollar THEN GO TO fail;
        v ← (tb+index).son1
        END;
      ENDCASE => GO TO fail;
    REPEAT
      fail => ErrorDefs.errortree[nonLHS, son1];
    ENDLOOP;
  END;
syserror, nullstmt => NULL;
openstmt =>
  BEGIN OPEN (tb+node);
  scanlist[son1, OpenItem];
  son2 ← updatelist[son2, Stmt];
  reversescanlist[son1, CloseItem];
  END;
enable =>
  BEGIN OPEN (tb+node);
```

```
            saveEnabled: BOOLEAN = current.unwindEnabled;
            IF CatchPhrase[son1].unwindCaught
              THEN current.unwindEnabled ← TRUE;
            son2 ← updatelist[son2, Stmt];
            current.unwindEnabled ← saveEnabled;
            END;
          list => val ← updatelist[val, Stmt];
          ENDCASE => ErrorDefs.error[unimplemented];
        END;
      ENDCASE => ERROR;
    IF markCatch
      THEN
        BEGIN mlpush[val]; pushtree[catchmark,1];
        setinfo[dataPtr.textIndex]; val ← mlpop[];
        END;
    markCatch ← saveMark; dataPtr.textIndex ← saveIndex; RETURN
    END;

    CheckVoid: PROCEDURE =
      BEGIN
      SELECT RType[] FROM
        CSENull, typeANY => NULL;
        ENDCASE => ErrorDefs.error[nonVoidStmt];
      RPop[]; RETURN
      END;


-- case statements

 Case: PUBLIC PROCEDURE [node: TreeIndex, selection: TreeMap] =
   BEGIN OPEN (tb+node);
   saveType: CSEIndex = passPtr.implicitType;
   saveTree: TreeLink = passPtr.implicitTree;

   eqTests: BOOLEAN;

   CaseItem: TreeScan =
     BEGIN
     switchable: BOOLEAN;
     saveIndex: CARDINAL = dataPtr.textIndex;

     CaseTest: TreeMap =
       BEGIN
       node: TreeIndex = GetNode[t];
         BEGIN OPEN (tb+node);
         SELECT name FROM
           relE =>
             BEGIN
             type: CSEIndex;
             son2 ← Rhs[son2, TargetType[passPtr.implicitType]];
             type ← RType[];
             SELECT (seb+type).typetag FROM
               long => BEGIN attr1 ← TRUE; attr2 ← FALSE END;
               real => attr1 ← attr2 ← TRUE;
               ENDCASE => attr1 ← attr2 ← FALSE;
             switchable ← switchable AND RConst[]; v ← t;
             END;
           ENDCASE =>
             BEGIN
             v ← Rhs[t, dataPtr.typeBOOLEAN];
             eqTests ← switchable ← FALSE;
             END;
         RPop[];
         END;
       RETURN
       END;

     node: TreeIndex = GetNode[t];
     dataPtr.textIndex ← (tb+node).info;
       BEGIN OPEN (tb+node);
       switchable ← TRUE;
       son1 ← updatelist[son1, CaseTest]; attr1 ← switchable;
       son2 ← selection[son2];
       END;
     dataPtr.textIndex ← saveIndex; RETURN
     END;
```

```
    son1 ← Exp[son1, typeANY];
    passPtr.implicitType ← CanonicalType[RType[]];
    RPop[];
    IF ~IdentifiedType[passPtr.implicitType]
      THEN ErrorDefs.errortree[relationType, son1];
    passPtr.implicitTree ← son1;  eqTests ← TRUE;
    scanlist[son2, CaseItem];  attr1 ← eqTests;
    son3 ← selection[son3];
    passPtr.implicitType ← saveType;  passPtr.implicitTree ← saveTree;
    RETURN
    END;


-- loop statements

  DoStmt: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    forNode: TreeIndex;
    cvType: CSEIndex;
    saveList: TreeLink;
    IF son1 # empty
      THEN
        BEGIN  forNode ← GetNode[son1];
        IF (tb+forNode).son1 = empty
          THEN RPush[typeANY, FALSE]
          ELSE
            BEGIN
            (tb+forNode).son1 ← Exp[(tb+forNode).son1, typeANY];
            IF ~OperandLhs[(tb+forNode).son1]
              THEN ErrorDefs.errortree[nonLHS, (tb+forNode).son1];
            WITH (tb+forNode).son1 SELECT FROM
              symbol =>
                BEGIN   -- account for implicit references
                BumpCount[index];  BumpCount[index];
                END;
              ENDCASE =>  ErrorDefs.errortree[controlId, (tb+forNode).son1];
            END;
        cvType ← TargetType[RType[]];  RPop[];
        SELECT (tb+forNode).name FROM
          forseq =>
            BEGIN  -- sequencing using a generator
            OPEN seq: (tb+forNode);
            seq.son2 ← Rhs[seq.son2, cvType];  RPop[];
            seq.son3 ← Rhs[seq.son3, cvType];  RPop[];
            END;
          upthru, downthru =>
            BEGIN  -- stepping through an interval
            IF ~OrderedType[cvType] AND cvType # typeANY
              THEN ErrorDefs.error[nonOrderedType];
            (tb+forNode).son2 ← Range[(tb+forNode).son2,  cvType];
            END;
          ENDCASE =>  ERROR;
        END;
    IF son2 # empty
      THEN
        BEGIN  son2 ← Rhs[son2, dataPtr.typeBOOLEAN];  RPop[];
        END;
    scanlist[son3, OpenItem];
    current.loopDepth ← current.loopDepth + 1;
    saveList ← current.labelList;  InsertLabels[son5];
    son4 ← updatelist[son4, Stmt];
    DeleteLabels[mark: saveList];
    current.loopDepth ← current.loopDepth - 1;
    IF son5 # empty THEN scanlist[son5, LabelItem];
    son6 ← updatelist[son6, Stmt];
    reversescanlist[son3, CloseItem];
    RETURN
    END;


  Range: PUBLIC PROCEDURE [t: TreeLink, type: CSEIndex] RETURNS [val: TreeLink] =
    BEGIN
    subType: CSEIndex;
    node: TreeIndex;
    WITH t SELECT FROM
```

```
      subtree =>
        BEGIN  node ← index;
        SELECT (tb+node).name FROM
          subrangeTC =>
            BEGIN  val ← t;
            (tb+node).son1 ← TypeExp[(tb+node).son1];
            subType ← TargetType[
                UnderType[TypeForTree[(tb+node).son1]]];
            Interval[(tb+node).son2, subType, FALSE];
            END;
          IN [intOO .. intCC] =>
            BEGIN  val ← t;
            subType ← IF type # typeANY THEN type ELSE dataPtr.typeINTEGER;
            Interval[t, subType, FALSE];
            END;
          ENDCASE =>
            BEGIN  val ← TypeExp[t];
            subType ← TargetType[UnderType[TypeForTree[val]]];
            END;
        END;
      ENDCASE =>
        BEGIN  val ← TypeExp[t];
        subType ← TargetType[UnderType[TypeForTree[val]]];
        END;
    IF ~OrderedType[subType] AND subType # typeANY
      THEN ErrorDefs.error[nonOrderedType];
    IF ~TypePackDefs.AssignableTypes[
                [dataPtr.ownSymbols, type],
                [dataPtr.ownSymbols, subType]]
      THEN ErrorDefs.errortree[typeClash, val];
    RETURN
    END;


-- labels

  LabelItem: PROCEDURE [item: TreeLink] =
    BEGIN
    node: TreeIndex = GetNode[item];
    (tb+node).son2 ← updatelist[(tb+node).son2, Stmt];
    RETURN
    END;

  InsertLabels: PROCEDURE [t: TreeLink] =
    BEGIN
    labelMark: TreeLink = current.labelList;

    InsertLabel: PROCEDURE [labeled: TreeLink] =
      BEGIN
      node: TreeIndex = GetNode[labeled];
      saveIndex: CARDINAL = dataPtr.textIndex;
      dataPtr.textIndex ← (tb+node).info;
      scanlist[(tb+node).son1, StackLabel];
      dataPtr.textIndex ← saveIndex;  RETURN
      END;

    StackLabel: PROCEDURE [id: TreeLink] =
      BEGIN
      t: TreeLink;
      node: TreeIndex;
      FOR t ← current.labelList, (tb+node).son2 UNTIL t = labelMark
        DO
        node ← GetNode[t];
        IF (tb+node).son1 = id AND id # nullid
          THEN  ErrorDefs.errortree[duplicateLabel, id];
        ENDLOOP;
      mlpush[id];  mlpush[current.labelList];
      current.labelList ← maketree[item, 2];  RETURN
      END;

    scanlist[t, InsertLabel];
    RETURN
    END;

  ValidateLabel: PROCEDURE [id: TreeLink] =
    BEGIN
```

```
    t: TreeLink;
    node: TreeIndex;
    FOR t ← current.labelList, (tb+node).son2 UNTIL t = empty
      DO
      node ← GetNode[t];
      IF (tb+node).son1 = id THEN RETURN;
      ENDLOOP;
    ErrorDefs.errortree[unknownLabel, id];
    RETURN
    END;

  DeleteLabels: PROCEDURE [mark: TreeLink] =
    BEGIN
    node: TreeIndex;
    UNTIL current.labelList = mark
      DO
      node ← GetNode[current.labelList];
      current.labelList ← (tb+node).son2;
      (tb+node).son2 ← empty;  freenode[node];
      ENDLOOP;
    RETURN
    END;


-- control transfers

  CheckLocals: PROCEDURE [t: TreeLink] RETURNS [localsOnly: BOOLEAN] =
    BEGIN
    level: ContextLevel = (bb+dataPtr.bodyIndex).level;

    CheckElement: TreeScan =
      BEGIN
      sei: ISEIndex;
      WITH t SELECT FROM
        literal => NULL;
        symbol =>
          BEGIN  sei ← index;
          IF ~(seb+sei).constant AND (ctxb+(seb+sei).ctxnum).ctxlevel # level
            THEN localsOnly ← FALSE;
          END;
        ENDCASE => localsOnly ← FALSE;
      RETURN
      END;

    localsOnly ← TRUE;  scanlist[t, CheckElement];  RETURN
    END;


  Return: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    rSei: recordCSEIndex = current.returnRecord;
    IF current.catchDepth # 0
     OR (dataPtr.bodyIndex = dataPtr.mainBody AND rSei = SENull)
      THEN ErrorDefs.error[misplacedReturn];
    IF (attr1 ← current.entry)
      THEN CountTreeIds[(tb+current.bodyNode).son4];
    IF rSei # SENull AND son1 = empty
      THEN
        BEGIN
        attr2 ← TRUE;  current.impliedReturn ← TRUE;
        IF (seb+(ctxb+(seb+rSei).fieldctx).selist).htptr = HTNull
          THEN ErrorDefs.error[illDefinedReturn];
        END
      ELSE
        BEGIN
        son1 ← MatchFields[rSei, son1, FALSE];
        IF current.entry
          THEN  (tb+node).attr2 ← CheckLocals[son1];
        END;
    RETURN
    END;


  BodyLiteral: PROCEDURE [bti: CBTIndex] RETURNS [TreeLink] =
    BEGIN
    mdi: MDIndex;
```

```
    sti: LitDefs.STIndex;
    desc: StringDefs.SubStringDescriptor;
    s: StringDefs.SubString = @desc;
    mdi ← WITH (ctxb+(bb+bti).localCtx) SELECT FROM
            included => ctxmodule,
            ENDCASE => OwnMdi;
    SubStringForHash[s, (mdb+mdi).mdhti];
    sti ← LitDefs.FindStringLiteral[s];  LitDefs.StringLiteralReference[sti];
    RETURN [[[literal[info: [string[index: sti]]]]]]
    END;

  New: PUBLIC PROCEDURE [node: TreeIndex, target: CSEIndex] RETURNS [val: TreeLink] =
    BEGIN
    subNode: TreeIndex;
    type, mType, rType: CSEIndex;
    bti: CBTIndex;
    val ← ForceApplication[(tb+node).son1];
    (tb+node).son1 ← empty;  freenode[node];
    subNode ← GetNode[val];
      BEGIN  OPEN (tb+subNode);
      name ← new;
      son1 ← Exp[son1, typeANY];  mType ← RType[];
      RPop[];
      WITH (seb+mType) SELECT FROM
        transfer =>
          BEGIN
          IF mode # program THEN ErrorDefs.errortree[typeClash, son1];
          type ← mType;
          IF (bti ← XferBody[son1]) # BTNull
            THEN
              BEGIN
              IF (seb+target).typetag = pointer
                THEN type ←
                 MakePointerType[MakeFrameRecord[son1], target];
              [] ← freetree[son1];  son1 ← BodyLiteral[bti];
              attr1 ← FALSE;
              END
            ELSE attr1 ← TRUE;
          END;
        pointer =>
          BEGIN
          type ← mType; dereferenced ← TRUE; rType ← UnderType[pointedtotype];
          WITH (seb+rType) SELECT FROM
            record =>
              IF (ctxb+fieldctx).ctxlevel # 1G
                THEN ErrorDefs.errortree[typeClash, son1]
                ELSE  IF (seb+target).typetag = transfer
                  THEN type ← XferForFrame[fieldctx];
            ENDCASE =>
              IF pointedtotype # typeANY
                THEN ErrorDefs.errortree[typeClash, son1];
          attr1 ← TRUE;
          END;
        ENDCASE =>
          BEGIN  type ← typeANY;
          IF mType # typeANY THEN ErrorDefs.errortree[typeClash, son1];
          END;
      IF son2 # empty
        THEN
          BEGIN  ErrorDefs.errortree[noApplication, son1];
          son2 ← updatelist[son2, VoidExp];
          END;
      IF nsons > 2 THEN [] ← CatchPhrase[son3];
      RPush[type, FALSE];
      END;
    RETURN
    END;


  Start: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN
    subNode: TreeIndex;
    val ← ForceApplication[(tb+node).son1];
    subNode ← GetNode[val];
    Apply[subNode, typeANY, TRUE];
    SELECT (tb+subNode).name FROM
```

```
    start, apply => NULL;
    ENDCASE => ErrorDefs.errortree[typeClash, (tb+subNode).son1];
  (tb+node).son1 ← empty;  freenode[node];  RETURN
  END;


FrameXfer: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN
  subNode: TreeIndex;
  type: CSEIndex;
  val ← ForceApplication[(tb+node).son1];
  subNode ← GetNode[val];
    BEGIN  OPEN (tb+subNode);
    name ← (tb+node).name;  info ← (tb+node).info;
    son1 ← Exp[son1, typeANY];  type ← RType[];
    RPop[];
    WITH (seb+type) SELECT FROM
      pointer =>
        -- BEGIN  dereferenced ← TRUE;
        -- SELECT (seb+UnderType[pointedtotype]).typetag FROM
          -- record =>
               NULL;          -- a weak check for now
          -- ENDCASE =>
            -- IF pointedtotype # typeANY
              -- THEN ErrorDefs.errortree[typeClash, son1];
        -- END;
      transfer =>
        IF mode # program OR XferBody[son1] # BTNull
          THEN ErrorDefs.errortree[typeClash, son1];
      ENDCASE =>
        IF type # typeANY THEN ErrorDefs.errortree[typeClash, son1];
    IF son2 # empty
      THEN
        BEGIN  ErrorDefs.errortree[noApplication, son1];
        son2 ← updatelist[son2, VoidExp];
        END;
    IF nsons > 2 THEN [] ← CatchPhrase[son3];
    END;
  (tb+node).son1 ← empty;  freenode[node];  RETURN
  END;


Fork: PUBLIC PROCEDURE [node: TreeIndex, target: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  subNode: TreeIndex;
  type, subType: CSEIndex;
  val ← ForceApplication[(tb+node).son1];
  (tb+node).son1 ← empty;  freenode[node];
  subNode ← GetNode[val];
  Apply[subNode, typeANY, TRUE];  RPop[];
  SELECT (tb+subNode).name FROM
    call =>
      BEGIN
      IF passPtr.lockHeld AND OperandInternal[(tb+subNode).son1]
        THEN  ErrorDefs.errortree[internalCall, (tb+subNode).son1];
      subType ← OperandType[(tb+subNode).son1];
      WITH procType: (seb+subType) SELECT FROM
        transfer =>
          BEGIN
          type ← makenonctxse[SIZE[transfer constructor SERecord]];
          (seb+type)↑ ← SERecord[mark3: TRUE, mark4: TRUE,
                        sebody: constructor[transfer[
                            mode: process,
                            inrecord: recordCSENull,
                            outrecord: procType.outrecord]]];
          END;
        ENDCASE => ERROR;
      (tb+subNode).name ← fork;
      END;
    apply => type ← typeANY;
    ENDCASE =>
      BEGIN
      ErrorDefs.errortree[typeClash, (tb+node).son1];  type ← typeANY;
      END;
  (tb+subNode).info ← type;  RPush[type, FALSE];
  RETURN
```

```
    END;

  Join: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN
    subNode: TreeIndex;
    val ← ForceApplication[(tb+node).son1];
    subNode ← GetNode[val];
    Apply[subNode, typeANY, TRUE];
    SELECT (tb+subNode).name FROM
      join => NULL;
      apply => NULL;
      ENDCASE => ErrorDefs.errortree[typeClash, (tb+subNode).son1];
    (tb+node).son1 ← empty;  freenode[node];  RETURN
    END;

  Wait: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN
    subNode: TreeIndex;
    IF ~passPtr.lockHeld THEN  ErrorDefs.error[misplacedMonitorRef];
    val ← ForceApplication[(tb+node).son1];
    subNode ← GetNode[val];
    Apply[subNode, typeANY, TRUE];
    SELECT (tb+subNode).name FROM
      wait => NULL;
      apply => NULL;
      ENDCASE => ErrorDefs.errortree[typeClash, (tb+subNode).son1];
    (tb+node).son1 ← empty;  freenode[node];
    IF ~OperandLhs[(tb+subNode).son1]
      THEN ErrorDefs.errortree[nonLHS, (tb+subNode).son1];
    [] ← freetree[(tb+subNode).son2];
    (tb+subNode).son2 ← (tb+subNode).son1;  (tb+subNode).son1 ← CopyLock[];
    RETURN
    END;


  -- monitors

  LockVar: PROCEDURE [t: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    type, nType: CSEIndex;
    desc: StringDefs.SubStringDescriptor;
    sei: ISEIndex;
    nDerefs: CARDINAL;
    long, b: BOOLEAN;

    Dereference: PROCEDURE [type: CSEIndex] =
      BEGIN
      mlpush[val];  pushtree[uparrow, 1];  setinfo[type];  setattr[1, long];
      val ← mlpop[];
      RETURN
      END;

    val ← Exp[t, typeANY];  long ← LongPath[val];
    type ← RType[];  RPop[];  nDerefs ← 0;
      DO
      IF type = dataPtr.typeLOCK
        THEN
          BEGIN
          IF nDerefs # 0 THEN Dereference[type];
          GO TO success
          END;
      type ← TypeRoot[type];  nType ← NormalType[type];
      WITH (seb+nType) SELECT FROM
        record =>
          BEGIN
          IF monitored
            THEN
              BEGIN
              desc ← ["LOCK"L, 0, ("LOCK"L).length];
              [b, sei] ← SearchCtxList[EnterString[@desc], fieldctx];
              IF ~b THEN ERROR;
              mlpush[val];  pushsymtree[sei];
              pushtree[IF nDerefs = 0 THEN dollar ELSE dot, 2];
              setinfo[dataPtr.typeLOCK];  setattr[1, long];  val ← mlpop[];
              GO TO success;
              END;
```

```
        GO TO failure;
        END;
      pointer =>
        BEGIN
        IF (nDerefs ← nDerefs + 1) > 255 THEN GO TO failure;
        IF nDerefs > 1 THEN Dereference[type];
        long ← (seb+type).typetag = long;
        dereferenced ← TRUE;  type ← UnderType[pointedtotype];
        END;
      ENDCASE => GO TO failure;
    REPEAT
      success => NULL;
      failure => ErrorDefs.errortree[typeClash, val];
    ENDLOOP;
  IF ~OperandLhs[val] THEN ErrorDefs.errortree[nonLHS, val];
  RETURN
  END;

FindLockParams: PROCEDURE RETURNS [formal, actual: ISEIndex] =
  BEGIN
  node: TreeIndex = GetNode[(tb+passPtr.lockNode).son1];
  found: BOOLEAN;
  IF node = nullTreeIndex
    THEN  formal ← actual ← ISENull
    ELSE
      BEGIN
      WITH (tb+node).son1 SELECT FROM
        symbol => formal ← index;
        ENDCASE => ERROR;
      IF current.inputRecord = SENull
        THEN  found ← FALSE
        ELSE  [found, actual] ← SearchCtxList[
                  (seb+formal).htptr,
                  (seb+current.inputRecord).fieldctx];
      IF ~found THEN actual ← ISENull;
      END;
  RETURN
  END;

CopyLock: PROCEDURE RETURNS [val: TreeLink] =
  BEGIN
  formal, actual: ISEIndex;
  SELECT TRUE FROM
    passPtr.lockNode = nullTreeIndex =>
      val ← empty;
    (tb+current.bodyNode).son4 # empty =>
      val ← LambdaApply[(tb+current.bodyNode).son4, ISENull, ISENull];
    ENDCASE =>
      BEGIN
      [formal:formal, actual:actual] ← FindLockParams[];
      IF formal # SENull
        THEN
          BEGIN
          IF actual = SENull
            THEN
              BEGIN
              ErrorDefs.errorsei[missingLock, formal];
              actual ← dataPtr.seAnon;
              END;
          IF ~TypePackDefs.AssignableTypes[
                  [dataPtr.ownSymbols, UnderType[(seb+formal).idtype]],
                  [dataPtr.ownSymbols, UnderType[(seb+actual).idtype]]]
            THEN ErrorDefs.errortree[typeClash, [symbol[index: actual]]];
          END;
      val ← LambdaApply[(tb+passPtr.lockNode).son2, formal, actual];
      END;
  RETURN
  END;


-- basing

OpenItem: TreeScan =
  BEGIN
  node: TreeIndex = GetNode[t];
  saveIndex: CARDINAL = dataPtr.textIndex;
```

```
    dataPtr.textIndex ← (tb+node).info;
    WITH (tb+node).son1 SELECT FROM
      hash =>
        (tb+node).son2 ← OpenBase[(tb+node).son2, index
         ! InsertCatchLabel => BEGIN ErrorDefs.error[catchLabel]; RESUME END];
      ENDCASE => ERROR;
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  CloseItem: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    WITH (tb+node).son1 SELECT FROM
      hash => CloseBase[(tb+node).son2, index];
      ENDCASE => ERROR;
    RETURN
    END;


-- signals

  Signal: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN
    subNode: TreeIndex;
    nodeTag: NodeName = (tb+node).name;
    IF nodeTag = xerror AND current.catchDepth # 0
      THEN ErrorDefs.error[misplacedReturn];
    val ← ForceApplication[(tb+node).son1];
    subNode ← GetNode[val];
    Apply[subNode, typeANY, TRUE];
    SELECT (tb+subNode).name FROM
      signal => NULL;
      error =>
        IF nodeTag # error AND nodeTag # xerror
          THEN ErrorDefs.errortree[typeClash, (tb+subNode).son1];
      apply => NULL;
      ENDCASE => ErrorDefs.errortree[typeClash, (tb+subNode).son1];
    (tb+subNode).name ← nodeTag;
    IF nodeTag = xerror
      THEN
        BEGIN
        (tb+subNode).attr1 ← current.entry;
        IF current.entry
          THEN (tb+subNode).attr2 ← CheckLocals[(tb+subNode).son2];
        END;
    (tb+node).son1 ← empty;  freenode[node];  RETURN
    END;

  ForceApplication: PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
    BEGIN
    IF testtree[t, apply] THEN  RETURN [t];
    mlpush[t];  mlpush[empty];
    RETURN [maketree[apply, 2]]
    END;


-- catch phrases


  CatchPhrase: PUBLIC PROCEDURE [t: TreeLink] RETURNS [unwindCaught: BOOLEAN] =
    BEGIN

    CatchItem: TreeScan =
      BEGIN
      node: TreeIndex = GetNode[t];
      type: CSEIndex;
      mixed: BOOLEAN;
      saveIndex: CARDINAL = dataPtr.textIndex;

      CatchLabel: TreeMap =
        BEGIN
        subType: CSEIndex;
        v ← Exp[t, typeANY];
        subType ← CanonicalType[RType[]];  RPop[];
        SELECT XferMode[subType] FROM
          signal, error =>
```

```
          IF type = typeANY
            THEN type ← subType
            ELSE IF type # subType THEN mixed ← TRUE;
        ENDCASE =>
          IF subType # typeANY THEN ErrorDefs.errortree[typeClash, t];
      RETURN
      END;

    dataPtr.textIndex ← (tb+node).info;
    type ← typeANY;  mixed ← FALSE;
    (tb+node).son1 ← updatelist[(tb+node).son1, CatchLabel];
    IF mixed THEN type ← typeANY;
    (tb+node).son2 ← CatchBody[(tb+node).son2, type];
    IF (tb+node).son1 = TreeLink[symbol[index: dataPtr.idUNWIND]]
      THEN
        BEGIN
        unwindCaught ← TRUE;
        IF current.entry AND ~current.unwindEnabled
         AND current.catchDepth = 0
          THEN
            BEGIN
            mlpush[(tb+node).son2];  mlpush[CopyLock[]];
            pushtree[unlock, 1];  setinfo[dataPtr.textIndex];
            (tb+node).son2 ← makelist[2];
            END;
        END;
    (tb+node).info ← IF type # typeANY THEN type ELSE SENull;
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  setLabel: BOOLEAN;
  node: TreeIndex = GetNode[t];
  setLabel ← unwindCaught ← FALSE;
    BEGIN
    ENABLE InsertCatchLabel =>
      IF ~catchSeen
        THEN
          BEGIN
          setLabel ← TRUE; SIGNAL InsertCatchLabel[catchSeen: TRUE]; RESUME
          END;
    scanlist[(tb+node).son1, CatchItem];
    IF (tb+node).nsons > 1
      THEN (tb+node).son2 ← CatchBody[(tb+node).son2, typeANY];
    END;
  IF setLabel THEN markCatch ← TRUE;
  RETURN
  END;

CatchBody: PROCEDURE [body: TreeLink, type: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  saveRecord:  recordCSEIndex = current.resumeRecord;
  saveFlag: BOOLEAN = current.resumeFlag;
  current.catchDepth ← current.catchDepth + 1;
  WITH (seb+type) SELECT FROM
    transfer =>
      BEGIN  current.resumeFlag ← mode = signal;
      PushArgCtx[inrecord];
      PushArgCtx[current.resumeRecord ← outrecord];
      END;
    ENDCASE =>
      BEGIN  current.resumeFlag ← FALSE;
      current.resumeRecord ← recordCSENull;
      END;
  val ← updatelist[body, Stmt
        ! InsertCatchLabel => IF catchSeen THEN RESUME];
  WITH (seb+type) SELECT FROM
    transfer =>
      BEGIN  PopArgCtx[outrecord];  PopArgCtx[inrecord];
      END;
    ENDCASE;
  current.catchDepth ← current.catchDepth - 1;
  current.resumeRecord ← saveRecord;  current.resumeFlag ← saveFlag;
  RETURN
  END;
```

```
Resume: PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  rSei: recordCSEIndex = current.resumeRecord;
  n: CARDINAL;
  sei: ISEIndex;
  IF ~current.resumeFlag THEN ErrorDefs.error[misplacedResume];
  IF rSei = SENull OR son1 # empty
    THEN  son1 ← MatchFields[rSei, son1, FALSE]
    ELSE
      BEGIN  n ← 0;
      BumpFieldRefs[rSei];
      FOR sei ← (ctxb+(seb+rSei).fieldctx).selist, NextSe[sei] UNTIL sei = SENull
        DO
        n ← n+1;
        IF n=1 AND (seb+sei).htptr = HTNull
          THEN ErrorDefs.error[illDefinedReturn];
        pushsymtree[sei];
        ENDLOOP;
      son1 ← makelist[n];
      END;
  RETURN
  END;

END.
```