```
-- file Pass4S.Mesa
-- last modified by Satterthwaite, July 31, 1978  9:30 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  ComData: FROM "comdata",
  ControlDefs: FROM "controldefs",
  ErrorDefs: FROM "errordefs",
  LitDefs: FROM "litdefs",
  P4Defs: FROM "p4defs",
  Pass4: FROM "pass4",
  SymDefs: FROM "symdefs",
  SymTabDefs: FROM "symtabdefs",
  TableDefs: FROM "tabledefs",
  TreeDefs: FROM "treedefs";

Pass4S: PROGRAM
    IMPORTS
        ErrorDefs, LitDefs, P4Defs, SymTabDefs, TreeDefs,
        dataPtr: ComData, passPtr: Pass4
    EXPORTS P4Defs =
  BEGIN
  OPEN TreeDefs, SymTabDefs, SymDefs;

  tb: TableDefs.TableBase;      -- tree base address (local copy)
  seb: TableDefs.TableBase;     -- se table base address (local copy)
  ctxb: TableDefs.TableBase;    -- ctx table base address (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];  bb ← base[bodytype];  RETURN
    END;

  WordLength: CARDINAL = AltoDefs.wordlength;
  Repr: TYPE = P4Defs.Repr;
    none: Repr = P4Defs.none;

 -- bodies and blocks

  frameBase, frameBound: CARDINAL;
  CatchFrameBase: CARDINAL = (ControlDefs.localbase+1)*WordLength;
  catchFrameBound: CARDINAL;


  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 => BodyList[(bb+bti).firstSon];
        IF (bb+bti).link.which = parent THEN EXIT;
        bti ← (bb+bti).link.index;
        ENDLOOP;
    RETURN
    END;

  Body: PROCEDURE [bti: CBTIndex] =
    BEGIN
    oldBodyIndex: CBTIndex = dataPtr.bodyIndex;
    saveIndex: CARDINAL = dataPtr.textIndex;
    saveBase: CARDINAL = frameBase;
    saveBound: CARDINAL = frameBound;
    saveCatchBound: CARDINAL = catchFrameBound;
    saveRecord: recordCSEIndex = passPtr.returnRecord;
    node: TreeIndex;
    sei: CSEIndex;
    initTree: TreeLink;
    n: CARDINAL;
    dataPtr.bodyIndex ← bti;
    WITH (bb+bti).info SELECT FROM
      Internal =>  BEGIN node ← bodyTree; dataPtr.textIndex ← sourceIndex END;
```

```
    ENDCASE => ERROR;
  sei ← UnderType[(bb+bti).ioType];
  passPtr.returnRecord ← TransferTypes[sei].typeOut;
  catchFrameBound ← CatchFrameBase + WordLength;
  [] ← LitDefs.ResetLocalStrings[];
  IF (tb+node).son4 # empty
    THEN
      BEGIN
      (tb+node).son4 ← P4Defs.Exp[(tb+node).son4, none]; P4Defs.VPop[];
      END;
  (tb+node).son1 ← updatelist[(tb+node).son1, OpenItem];
  scanlist[(tb+node).son2, P4Defs.DeclItem];
  IF ~dataPtr.definitionsOnly
    THEN
      frameBase ← SELECT (bb+bti).level FROM
        1G => P4Defs.LayoutGlobals[bti],
        ENDCASE => P4Defs.LayoutLocals[bti]
    ELSE
      BEGIN
      n ← P4Defs.LayoutInterface[bti]; frameBase ← 0;
      WITH (seb+sei) SELECT FROM
        definition =>
          nGfi ← IF n=0
                  THEN 1
                  ELSE LOOPHOLE[n-1, CARDINAL]/ControlDefs.EPRange + 1;
        ENDCASE => [] ← P4Defs.LayoutLocals[bti];
      END;
  initTree ← empty;
  SELECT (bb+bti).level FROM
    1G =>
      IF dataPtr.monitored AND (tb+passPtr.lockNode).attr1
        THEN
          BEGIN
          mlpush[(tb+passPtr.lockNode).son2];
          pushlittree[LitDefs.FindLiteral[100000B]]; pushtree[cast, 1];
          setinfo[dataPtr.typeLOCK];
          pushtree[assign, 2]; setattr[1, FALSE]; initTree ← mlpop[];
          END;
    ENDCASE =>
      IF (bb+bti).firstSon # BTNull
        THEN
          BEGIN
          frameBase ← P4Defs.AssignLocalDescriptors[(bb+bti).firstSon, frameBase];
          initTree ← BodyInitList[(bb+bti).firstSon];
          END;
  frameBound ← frameBase;
  (tb+node).son3 ← updatelist[(tb+node).son3, Stmt];
  WITH (bb+bti).info SELECT FROM
    Internal =>
      BEGIN
      frameSize ← (frameBound + (WordLength-1)) / WordLength;
      stOrigin ← LitDefs.ResetLocalStrings[];
      END;
    ENDCASE;
  IF (bb+bti).firstSon # BTNull
    THEN BodyList[(bb+bti).firstSon]
    ELSE (tb+node).son1 ← reverseupdatelist[(tb+node).son1, CloseItem];
  (tb+node).son2 ← updatelist[(tb+node).son2, P4Defs.DeclUpdate];
  IF initTree # empty
    THEN
      BEGIN mlpush[initTree];
      IF (tb+node).son2 # empty
        THEN BEGIN mlpush[(tb+node).son2]; pushlist[2] END;
      (tb+node).son2 ← mlpop[];
      END;
  frameBase ← saveBase; frameBound ← saveBound;
  catchFrameBound ← saveCatchBound;
  dataPtr.bodyIndex ← oldBodyIndex; dataPtr.textIndex ← saveIndex;
  passPtr.returnRecord ← saveRecord; RETURN
  END;

BodyInitList: PROCEDURE [firstBti: BTIndex] RETURNS [TreeLink] =
  BEGIN
  bti: BTIndex;
  n: CARDINAL;
  n ← 0;
```

```
    IF (bti ← firstBti) # BTNull
      THEN
        DO
        WITH (bb+bti) SELECT FROM
          Callable =>
            BEGIN  pushtree[procinit, 0];  setinfo[bti];  n ← n+1  END;
          ENDCASE => NULL;
        IF (bb+bti).link.which = parent THEN EXIT;
        bti ← (bb+bti).link.index;
        ENDLOOP;
    RETURN [makelist[n]]
    END;


  Block: PROCEDURE [node: TreeIndex] RETURNS [TreeLink] =
    BEGIN  OPEN (tb+node);
    initTree: TreeLink;
    bti: BTIndex = info;
    saveBase: CARDINAL = frameBase;
    saveBound: CARDINAL = frameBound;
    saveIndex: CARDINAL = dataPtr.textIndex;
    WITH (bb+bti).info SELECT FROM
      Internal =>  dataPtr.textIndex ← sourceIndex;
      ENDCASE;
    scanlist[(tb+node).son1, P4Defs.DeclItem];
    frameBase ← P4Defs.LayoutBlock[bti, frameBase];
    initTree ← empty;
    IF (bb+bti).level # 1G AND (bb+bti).firstSon # BTNull
      THEN
        BEGIN
        frameBase ← P4Defs.AssignLocalDescriptors[(bb+bti).firstSon, frameBase];
        initTree ← BodyInitList[(bb+bti).firstSon];
        END;
    frameBound ← frameBase;
    (tb+node).son2 ← updatelist[(tb+node).son2, Stmt];
    WITH (bb+bti).info SELECT FROM
      Internal =>  frameSize ← (frameBound + (WordLength-1)) / WordLength;
      ENDCASE;
    (tb+node).son1 ← updatelist[(tb+node).son1, P4Defs.DeclUpdate];
    IF initTree # empty
      THEN
        BEGIN  mlpush[initTree];
        IF (tb+node).son1 # empty
          THEN  BEGIN  mlpush[(tb+node).son1];  pushlist[2]  END;
        (tb+node).son1 ← mlpop[];
        END;
    frameBase ← saveBase;  frameBound ← MAX [frameBound, saveBound];
    dataPtr.textIndex ← saveIndex;
    RETURN [TreeLink[subtree[index: node]]]
    END;


  -- main dispatch

  Stmt: PROCEDURE [stmt: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    node: TreeIndex;
    saveIndex: CARDINAL = dataPtr.textIndex;
    val ← stmt;          -- the default case
    WITH stmt SELECT FROM
      subtree =>
        BEGIN  node ← index;
        IF node # nullTreeIndex
          THEN
            BEGIN  OPEN (tb+node);
            dataPtr.textIndex ← info;
            SELECT name FROM
              assign =>
                BEGIN  val ← P4Defs.Assignment[node];  P4Defs.VPop[]  END;
              extract =>  Extract[node];
              call, portcall, signal, error, xerror, start, join =>
                BEGIN  val ← P4Defs.Call[node];  P4Defs.VPop[]  END;
              block =>  val ← Block[node];
              ifstmt =>  val ← IfStmt[node];
              casestmt =>  val ← CaseDriver[node, Stmt];
              bindstmt =>  val ← Binding[node, casestmt, BindStmt];
```

```
            dostmt =>  val ← DoStmt[node];
            return =>
              son1 ← P4Defs.MakeArgRecord[passPtr.returnRecord, son1];
            label =>
              BEGIN
              son1 ← Stmt[son1];
              son2 ← updatelist[son2, Stmt];
              END;
            goto, exit, loop, nullstmt =>  NULL;
            restart =>
              BEGIN
              son1 ← P4Defs.NeutralExp[son1];
              IF nsons > 2 THEN CatchNest[son3];
              END;
            stop =>  CatchNest[son1];
            wait =>
              BEGIN
              son1 ← P4Defs.Exp[son1, none];  P4Defs.VPop[];
              son2 ← P4Defs.Exp[son2, none];  P4Defs.VPop[];
              IF nsons > 2 THEN CatchNest[son3];
              END;
            notify, broadcast, unlock =>
              BEGIN  son1 ← P4Defs.Exp[son1, none];  P4Defs.VPop[]  END;
            syserror =>  NULL;
            openstmt =>
              BEGIN
              son1 ← updatelist[son1, OpenItem];
              son2 ← updatelist[son2, Stmt];
              END;
            enable =>  Enable[node];
            resume =>
              son1 ← P4Defs.MakeArgRecord[passPtr.resumeRecord, son1];
            continue, retry =>  NULL;
            catchmark =>  son1 ← Stmt[son1];
            dst, lst, lstf =>
              BEGIN
              son1 ← P4Defs.Exp[son1, none];
              IF P4Defs.WordsForType[P4Defs.OperandType[son1]] #
                  SIZE[ControlDefs.StateVector]
                THEN ErrorDefs.errortree[sizeClash, son1];
              P4Defs.VPop[];
              END;
            item =>  son2 ← Stmt[son2];
            list =>  val ← updatelist[stmt, Stmt];
            ENDCASE =>  ErrorDefs.error[unimplemented];
          END;
      END;
    ENDCASE =>  ERROR;
  dataPtr.textIndex ← saveIndex;  RETURN
  END;

-- extraction

 Extract: PROCEDURE [node: TreeIndex] =
   BEGIN  OPEN (tb+node);

   AssignItem: TreeMap =
     BEGIN
     type: CSEIndex;
     saveType: CSEIndex = passPtr.implicitType;
     saveBias: INTEGER = passPtr.implicitBias;
     saveRep: Repr = passPtr.implicitRep;
     IF t = empty
       THEN v ← empty
       ELSE
         BEGIN  type ← UnderType[(seb+sei).idtype];
         passPtr.implicitType ← type;
         passPtr.implicitBias ← P4Defs.BiasForType[type];
         passPtr.implicitRep ← P4Defs.RepForType[type];
         v ← P4Defs.Assignment[GetNode[t]];  P4Defs.VPop[];
         END;
     sei ← NextSe[sei];
     passPtr.implicitRep ← saveRep;  passPtr.implicitBias ← saveBias;
     passPtr.implicitType ← saveType;  RETURN
     END;
```

```
    subNode: TreeIndex = GetNode[son1];
    rType: recordCSEIndex = (tb+subNode).info;
    sei: ISEIndex;
    (seb+rType).lengthUsed ← TRUE;
    sei ← firstvisiblese[(seb+rType).fieldctx];
    son1 ← updatelist[son1, AssignItem];
    son2 ← P4Defs.Exp[son2, none];  P4Defs.VPop[];
    RETURN
    END;


  -- conditionals

  IfStmt: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    son1 ← P4Defs.NeutralExp[son1];
    son2 ← Stmt[son2];  son3 ← Stmt[son3];
    IF ~P4Defs.TreeLiteral[son1]
      THEN  val ← TreeLink[subtree[index: node]]
      ELSE
        BEGIN
        IF son1 # passPtr.tFALSE
          THEN  BEGIN  val ← son2;  son2 ← empty  END
          ELSE  BEGIN  val ← son3;  son3 ← empty  END;
        freenode[node];
        END;
    RETURN
    END;


  BindStmt: TreeMap =
    BEGIN
    RETURN [CaseDriver[GetNode[t], Stmt]]
    END;


  -- drivers for processing selections

  Binding: PUBLIC PROCEDURE [node: TreeIndex, op: NodeName, eval: TreeMap]
      RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    subNode: TreeIndex;
    mlpush[son2];  son2 ← empty;
    mlpush[son3];  son3 ← empty;
    mlpush[son4];  son4 ← empty;
    mlpush[OpenItem[son1]];  son1 ← empty;
    pushtree[op, 4];  setinfo[info];  setattr[1, FALSE];
    val ← eval[mlpop[]];  subNode ← GetNode[val];
    (tb+subNode).son4 ← CloseItem[(tb+subNode).son4];
    freenode[node];  RETURN
    END;


  CaseDriver: PUBLIC PROCEDURE [node: TreeIndex, selection: TreeMap]
      RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    type: CSEIndex = P4Defs.OperandType[son1];
    son1 ← P4Defs.Exp[son1, none];
    IF type = dataPtr.typeBOOLEAN AND attr1 AND P4Defs.TreeLiteral[son1]
      THEN
        BEGIN

        CaseItem: TreeScan =
          BEGIN
          subNode: TreeIndex = GetNode[t];
          started: BOOLEAN;

          PushTest: TreeScan =
            BEGIN
            tNode: TreeIndex = GetNode[t];
            mlpush[(tb+tNode).son2];  (tb+tNode).son2 ← empty;
            IF son1 = passPtr.tFALSE THEN pushtree[not, 1];
            IF started THEN pushtree[or, 2];
            started ← TRUE;  RETURN
            END;
```

```
      mlpush[(tb+subNode).son2];  (tb+subNode).son2 ← empty;
      started ← FALSE;  scanlist[(tb+subNode).son1, PushTest];
      IF selection = Stmt
        THEN  BEGIN pushtree[ifstmt, -3]; setinfo[(tb+subNode).info] END
        ELSE  BEGIN pushtree[ifexp, -3]; setinfo[(tb+node).info] END;
      RETURN
      END;

    son1 ← P4Defs.AdjustBias[son1, -P4Defs.VBias[]];  P4Defs.VPop[];
    mlpush[son3];  son3 ← empty;
    reversescanlist[son2, CaseItem];
    freenode[node];
    val ← selection[mlpop[]];
    END
  ELSE
    BEGIN
    nSons: CARDINAL = listlength[son2];
    i, first, last, copied, newSons: CARDINAL;
    min, max: INTEGER;
    rep: P4Defs.Repr;
    subNode: TreeIndex;
    switchable, copying: BOOLEAN;
    multiword: BOOLEAN = P4Defs.WordsForType[type] # 1;
    count: CARDINAL;

    SwitchValue: TreeMap =
      BEGIN
      val: INTEGER;
      tNode: TreeIndex = GetNode[t];
      (tb+tNode).son2 ←
            P4Defs.RValue[(tb+tNode).son2, passPtr.implicitBias, rep];
      P4Defs.VPop[];
      val ← P4Defs.TreeLiteralValue[(tb+tNode).son2];
      IF count = 0
        THEN  BEGIN  first ← i;  min ← max ← val  END
        ELSE
          BEGIN
          IF P4Defs.Compare[val, min, rep] < 0 THEN min ← val;
          IF P4Defs.Compare[val, max, rep] > 0 THEN max ← val;
          END;
      count ← count + 1;
      RETURN [t]
      END;

    p, q: POINTER [0..TableDefs.TableLimit) TO RECORD [soni: TreeLink];
    saveType: CSEIndex = passPtr.implicitType;
    saveBias: INTEGER = passPtr.implicitBias;
    saveRep: Repr = passPtr.implicitRep;
    passPtr.implicitType ← type;  passPtr.implicitBias ← P4Defs.VBias[];
    passPtr.implicitRep ← rep ← P4Defs.VRep[];  P4Defs.VPop[];
    newSons ← nSons;
    i ← 1;  copying ← FALSE;  copied ← 0;
    p ← q ← LOOPHOLE[GetNode[son2] + TreeNodeSize];
    UNTIL i > nSons
      DO
      WHILE i <= nSons
        DO
        subNode ← GetNode[(tb+p).soni];
        IF (tb+subNode).attr1 AND ~multiword THEN EXIT;
        (tb+subNode).son1 ← updatelist[(tb+subNode).son1, P4Defs.NeutralExp];
        (tb+subNode).son2 ← selection[(tb+subNode).son2];
        i ← i+1;  p ← p+1;
        ENDLOOP;
      switchable ← FALSE;  count ← 0;
      WHILE i <= nSons
        DO  -- N.B. implicitbias is never changed by this loop
        subNode ← GetNode[(tb+p).soni];
        IF ~(tb+subNode).attr1 OR multiword THEN EXIT;
        (tb+subNode).son1 ← updatelist[(tb+subNode).son1, SwitchValue];
        (tb+subNode).son2 ← selection[(tb+subNode).son2];
        switchable ← TRUE;  last ← i;
        i ← i+1;  p ← p+1;
        ENDLOOP;
      IF switchable AND SwitchWorthy[count, max-min]
        THEN
```

```
            BEGIN  copying ← TRUE;
            THROUGH (copied .. first)
              DO  mlpush[(tb+q).soni];  q ← q+1  ENDLOOP;
            mlpush[P4Defs.AdjustBias[empty, min]];
            mlpush[P4Defs.MakeTreeLiteral[max-min+1]];
            THROUGH [first .. last]
              DO  mlpush[SwitchTree[(tb+q).soni, min]];  q ← q+1  ENDLOOP;
            pushproperlist[last-first+1];
            mlpush[maketree[caseswitch, 3]];
            copied ← last;  newSons ← newSons - (last-first);
            END;
        ENDLOOP;
      IF copying
        THEN
          BEGIN
          THROUGH (copied .. nSons] DO mlpush[(tb+q).soni]; q ← q+1 ENDLOOP;
          pushproperlist[newSons];  son2 ← mlpop[];
          END;
      son3 ← selection[son3];
      val ← TreeLink[subtree[index: node]];
      passPtr.implicitRep ← saveRep;  passPtr.implicitBias ← saveBias;
      passPtr.implicitType ← saveType;
      END;
  RETURN
  END;

-- auxiliary routines for CaseDriver

  SwitchWorthy: PROCEDURE [entries, delta: CARDINAL] RETURNS [BOOLEAN] =
    -- the decision function for using a switch
    BEGIN  RETURN [delta < 77777B AND delta+6 < 3*entries]
    END;

  SwitchTree: PROCEDURE [t: TreeLink, offset: INTEGER] RETURNS [TreeLink] =
    BEGIN
    node: TreeIndex = GetNode[t];
    count: CARDINAL;

    PushSwitchEntry: TreeScan =
      BEGIN
      subNode: TreeIndex = GetNode[t];
      count ← count+1;
      mlpush[P4Defs.MakeTreeLiteral[
              P4Defs.TreeLiteralValue[(tb+subNode).son2]-offset]];
      RETURN
      END;

    count ← 0;  scanlist[(tb+node).son1, PushSwitchEntry];
    pushlist[count];  mlpush[(tb+node).son2];
    (tb+node).son2 ← empty;  freenode[node];
    RETURN [maketree[casetest, 2]]
    END;


-- iterative statements

 DoStmt: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
   BEGIN  OPEN (tb+node);
   subNode: TreeIndex;
   delete: BOOLEAN ← FALSE;
   IF son1 # empty
     THEN
       BEGIN  subNode ← GetNode[son1];
         BEGIN  -- process a for-clause
         OPEN (tb+subNode);
         idBias: INTEGER;
         idRep, target: Repr;
         idType: CSEIndex;

         EvalSeqItem: TreeMap =
           BEGIN
           v ← P4Defs.RValue[t, idBias, target];  P4Defs.VPop[];
           IF ~P4Defs.AssignableRanges[idType, P4Defs.OperandType[v]]
             THEN  ErrorDefs.errortree[sizeClash, v];
           RETURN
           END;
```

```
        IF son1 = empty
          THEN
            BEGIN
            idBias ← 0;  idRep ← P4Defs.both;  target ← P4Defs.none;
            idType ← dataPtr.typeINTEGER;
            END
          ELSE
            BEGIN
            son1 ← P4Defs.Exp[son1, none];
            idBias ← P4Defs.VBias[];  idRep ← P4Defs.VRep[];  P4Defs.VPop[];
            target ← P4Defs.TargetRep[idRep];
            idType ← P4Defs.OperandType[son1];
            END;
        SELECT name FROM
          forseq =>
            BEGIN
            son2 ← EvalSeqItem[son2];  son3 ← EvalSeqItem[son3];
            END;
          upthru, downthru =>
            BEGIN
            son2 ← P4Defs.Range[t:son2, bias:idBias, rep:idRep, target:target
                  !P4Defs.EmptyInterval =>
                     BEGIN  delete ← TRUE;  RESUME  END;
                   P4Defs.MixedRepresentation =>
                     BEGIN
                     ErrorDefs.errortree[mixedRepresentation, son2]; RESUME
                     END];
            IF P4Defs.WordsForType[idType] = 0 AND ~delete
              THEN ErrorDefs.errortree[sizeClash, son1];
            P4Defs.VPop[];
            END;
          ENDCASE =>  ERROR;
        END;
      END;
    IF son2 # empty
      THEN
        BEGIN  son2 ← P4Defs.NeutralExp[son2];
        SELECT son2 FROM
          passPtr.tTRUE => son2 ← freetree[son2];
          passPtr.tFALSE => delete ← TRUE;
          ENDCASE;
        END;
    son3 ← updatelist[son3, OpenItem];
    son4 ← updatelist[son4, Stmt];
    son5 ← updatelist[son5, Stmt];
    son6 ← updatelist[son6, Stmt];
    son3 ← reverseupdatelist[son3, CloseItem];
    IF ~delete
      THEN  val ← TreeLink[subtree[index: node]]
      ELSE  BEGIN  freenode[node];  val ← empty  END;
    RETURN
    END;


-- basing

 OpenItem: TreeMap =
    BEGIN
    node: TreeIndex = GetNode[t];
    IF ~testtree[(tb+node).son2, openexp]
      THEN  v ← empty
      ELSE
        BEGIN
        v ← P4Defs.NeutralExp[(tb+node).son2];  (tb+node).son2 ← empty;
        END;
    freenode[node];
    RETURN
    END;

 CloseItem: TreeMap =
    BEGIN
    node: TreeIndex;
    IF ~testtree[t, openexp]
      THEN  v ← t
      ELSE
```

```
      BEGIN
      setshared[t, FALSE];  node ← GetNode[t];
      v ← (tb+node).son1;  (tb+node).son1 ← empty;  freenode[node];
      END;
    RETURN
    END;


-- catch phrases

  Enable: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    saveCatchBound: CARDINAL = catchFrameBound;
    CatchPhrase[son1];
    son2 ← Stmt[son2];
    catchFrameBound ← saveCatchBound;  RETURN
    END;

  CatchNest: PUBLIC PROCEDURE [t: TreeLink] =
    BEGIN
    saveCatchBound: CARDINAL = catchFrameBound;
    IF t # empty THEN CatchPhrase[t];
    catchFrameBound ← saveCatchBound;  RETURN
    END;

  CatchPhrase: PROCEDURE [t: TreeLink] =
    BEGIN
    node: TreeIndex = GetNode[t];
    saveBase: CARDINAL = frameBase;
    saveBound: CARDINAL = frameBound;
    frameBound ← catchFrameBound;
    catchFrameBound ← CatchFrameBase + WordLength;
    scanlist[(tb+node).son1, CatchItem];
    IF (tb+node).nsons > 1 THEN
      BEGIN
      frameBase ← CatchFrameBase;  (tb+node).son2 ← Stmt[(tb+node).son2];
      END;
    (tb+node).info ← (frameBound + (WordLength-1))/WordLength;
    catchFrameBound ← frameBound;
    frameBase ← saveBase;  frameBound ← saveBound;
    RETURN
    END;

  CatchItem: TreeScan =
    BEGIN
    node: TreeIndex = GetNode[t];
    type: CSEIndex = (tb+node).info;
    saveRecord: recordCSEIndex = passPtr.resumeRecord;

    CatchTest: TreeMap =
      BEGIN
      mlpush[empty];  mlpush[P4Defs.Exp[t, none]];  P4Defs.VPop[];
      RETURN [maketree[relE, 2]]
      END;

    frameBase ← CatchFrameBase;
    (tb+node).son1 ← updatelist[(tb+node).son1, CatchTest];
    IF type = SENull
      THEN  passPtr.resumeRecord ← recordCSENull
      ELSE
        WITH (seb+type) SELECT FROM
          transfer =>
            BEGIN  passPtr.resumeRecord ← outrecord;
            frameBase ← frameBase + P4Defs.ArgLength[type];
            END;
          ENDCASE =>  ERROR;
    (tb+node).son2 ← Stmt[(tb+node).son2];
    IF frameBase > ControlDefs.MaxSmallFrameSize*WordLength
      THEN ErrorDefs.errorsei[addressOverflow, dataPtr.seAnon];
    frameBound ← MAX[frameBase, frameBound];
    passPtr.resumeRecord ← saveRecord;
    RETURN
    END;


  END.
```