```
-- Statement.mesa, modified by Sweet, Aug 29, 1978 11:37 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [BYTE, wordlength],
  Code: FROM "code" USING [acstack, actenable, catchcount, catchoutrecord, cfs, CodeNotImplemented, cod
**eptr, curctxlvl, fileindex, firstcaseselread, framesz, mwcaseseltlex, StackNotEmptyAtStatement, xtrac
**ting],
  CodeDefs: FROM "codedefs" USING [BDOIndex, CCIndex, CCItem, ChunkBase, CodeCCIndex, EXLRIndex, FreeCh
**unk, LabelCCIndex, LabelCCNull, Lexeme, OtherCCIndex, StkIndex, TempStateRecord, topostack],
  ComData: FROM "comdata" USING [textIndex],
  ControlDefs: FROM "controldefs" USING [localbase],
  ErrorDefs: FROM "errordefs" USING [error],
  FOpCodes: FROM "fopcodes" USING [qBCAST, qBCASTL, qBLT, qBLTL, qCATCH, qDADD, qDCOMP, qDSUB, qDEC, qD
**ST, qINC, qLL, qLP, qLST, qLSTF, qNOTIFY, qNOTIFYL, qPOP, qPUSH, qRET, qSL],
  P5ADefs: FROM "p5adefs" USING [adjustacstack, ccellalloc, Cflow, Ciout0, Ciout1, clearstack, computef
**ramesize, Coutjump, createlabel, Csyserror, genanonlex, genheaplex, getlabelmark, incrstack, insertla
**bel, labelalloc, loadlexaddress, loadtsonaddress, LogHeapFree, makeEXITlabel, markstack, newstack, P5
**Error, pop, popinvals, poplabels, poptempstate, purgeheaplist, purgependtemplist, pushheaplist, pusht
**empstate, releaseBDOItem, releasetemplex, RequireStack, resettomark, restoreoldstack, rmakeBDOItem, s
**Cassign, stackoff, stackon, treeliteral, treeliteralvalue, unmarkstack, wordsforoperand],
  P5BDefs: FROM "p5bdefs" USING [Cexp, MWConstant, pushlex, pushlitval, pushrhs],
  P5StmtExprDefs: FROM "p5stmtexprdefs" USING [Cassign, Ccall, Ccatchmark, Cconstruct, Ccontinue, Cexit
**, Cextract, Cgoto, Cjoin, Clabel, Clabelcreate, Clabellist, Cloop, Cprocinit, Crestart, Cresume, Cret
**ry, Creturn, Crowcons, Csigerr, Cstart, Cstop, Cunlock, Cvconstruct, Cwait, Cxerror],
  SymDefs: FROM "symdefs" USING [bodytype, BTIndex, ContextLevel, CTXIndex, CTXNull, ctxtype, HTIndex,
**ISEIndex, recordCSEIndex, recordCSENull, SEIndex, SENull, SERecord, setype],
  SymTabDefs: FROM "symtabdefs" USING [WordsForType],
  SystemDefs: FROM "systemdefs" USING [AllocateHeapNode, FreeHeapNode],
  TableDefs: FROM "tabledefs" USING [TableBase, TableLimit, TableNotifier],
  TreeDefs: FROM "treedefs" USING [empty, freetree, listlength, NodeName, reverseupdatelist, scanlist,
**setshared, TreeIndex, TreeLink, treetype, updatelist];

DEFINITIONS FROM FOpCodes, CodeDefs;

Statement: PROGRAM
    IMPORTS MPtr: ComData, CPtr: Code, CodeDefs, P5ADefs, P5BDefs, P5StmtExprDefs, SymTabDefs, SystemDe
**fs, TreeDefs, ErrorDefs
    EXPORTS CodeDefs, P5BDefs =
  BEGIN
  OPEN P5ADefs, P5BDefs, P5StmtExprDefs;


  -- imported definitions

  BYTE: TYPE = AltoDefs.BYTE;
  wordlength: CARDINAL = AltoDefs.wordlength;

  ContextLevel: TYPE = SymDefs.ContextLevel;
  CTXIndex: TYPE = SymDefs.CTXIndex;
  CTXNull: CTXIndex = SymDefs.CTXNull;
  HTIndex: TYPE = SymDefs.HTIndex;
  ISEIndex: TYPE = SymDefs.ISEIndex;
  recordCSEIndex: TYPE = SymDefs.recordCSEIndex;
  recordCSENull: recordCSEIndex = SymDefs.recordCSENull;
  SEIndex: TYPE = SymDefs.SEIndex;
  SENull: SEIndex = SymDefs.SENull;
  SERecord: TYPE = SymDefs.SERecord;
  BTIndex: TYPE = SymDefs.BTIndex;

  empty: TreeLink = TreeDefs.empty;
  NodeName: TYPE = TreeDefs.NodeName;
  TreeIndex: TYPE = TreeDefs.TreeIndex;
  TreeLink: TYPE = TreeDefs.TreeLink;


  tb: TableDefs.TableBase;              -- tree base (local copy)
  seb: TableDefs.TableBase;             -- semantic entry base (local copy)
  ctxb: TableDefs.TableBase;            -- context entry base (local copy)
  cb: ChunkBase;                -- code base (local copy)
  bb: TableDefs.TableBase;              -- body base (local copy)

  StatementNotify: PUBLIC TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever table area is repacked
    seb ← base[SymDefs.setype];
    ctxb ← base[SymDefs.ctxtype];
    tb ← base[TreeDefs.treetype];
```

```
    cb ← LOOPHOLE[tb];
    bb ← base[SymDefs.bodytype];
    RETURN
    END;

  CatchFrameTooLarge: SIGNAL = CODE;


  Cstatement: PUBLIC PROCEDURE [t: TreeLink] RETURNS[TreeLink] =
    BEGIN -- generates code for Mesa statements
    node: TreeIndex;
    savheaplist: ISEIndex;
    saveIndex: CARDINAL = MPtr.textIndex;

    clearstack[];
    IF t = empty THEN RETURN[empty];
    BEGIN
    ENABLE
      BEGIN
      LogHeapFree => RESUME[TRUE, genheaplex[]];
      CPtr.CodeNotImplemented => GO TO unimplementedConstruct
      END;
    savheaplist ← pushheaplist[];
    WITH t SELECT FROM
      subtree =>
        BEGIN
        node ← index;
        CPtr.fileindex ← MPtr.textIndex ← (tb+node).info;
        IF CPtr.acstack # 0 THEN
          BEGIN SIGNAL CPtr.StackNotEmptyAtStatement; CPtr.acstack ← 0; END;
        SELECT (tb+node).name FROM
          block => Cblock[node];
          start => Cstart[node];
          restart => Crestart[node];
          stop => Cstop[node];
          dst => Cdst[node];
          lst => Clst[node];
          lstf => Clstf[node];
          call, portcall => Ccall[node];
          signal,error => Csigerr[node];
          syserror => Csyserror[];
          label => Clabel[node];
          assign => Cassign[node];
          extract => Cextract[node];
          ifstmt => Cifstmt[node];
          casestmt => [] ← Ccasestmtexp[node, FALSE];
          dostmt => Cdostmt[node];
          exit => Cexit[];
          loop => Cloop[];
          retry => Cretry[];
          construct => Cconstruct[node];
          vconstruct => Cvconstruct[node];
          continue => Ccontinue[];
          goto => Cgoto[node];
          catchmark => Ccatchmark[node];
          rowcons => Crowcons[node];
          return => Creturn[node];
          resume => Cresume[node];
          openstmt => Copen[node];
          enable => Cenable[node];
          procinit => Cprocinit[node];
          wait => Cwait[node];
          notify => Cnotify[node];
          broadcast => Cbroadcast[node];
          join => Cjoin[node];
          unlock => Cunlock[node];
          xerror => Cxerror[node];
          nullstmt => NULL;
          list => t ← TreeDefs.updatelist[t, Cstatement];
          ENDCASE => GO TO unimplementedConstruct;
        END;
      ENDCASE;
    purgeheaplist[savheaplist];
    purgependtemplist[];
    EXITS
      unimplementedConstruct =>
```

```
      BEGIN
      ErrorDefs.error[unimplemented];
      CPtr.acstack ← 0;
      clearstack[];
      END;
    END;
    MPtr.textIndex ← saveIndex;
    [] ← TreeDefs.freetree[t];
    RETURN[empty]
    END;


  Copen: PROCEDURE [node: TreeIndex] =
    BEGIN
    OPEN TreeDefs;
    sCopen: PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
      BEGIN
      setshared[t, FALSE];
      RETURN[freetree[t]]
      END;

    (tb+node).son2 ← Cstatement[(tb+node).son2];
    (tb+node).son1 ← reverseupdatelist[(tb+node).son1, sCopen];
    RETURN
    END;


  Cdst: PROCEDURE [node: TreeIndex] =
    BEGIN -- generates dumpstate
    dlstate[node, qDST];
    RETURN
    END;


  Clst: PROCEDURE [node: TreeIndex] =
    BEGIN -- generates loadstate
    dlstate[node, qLST];
    RETURN
    END;


  Clstf: PROCEDURE [node: TreeIndex] =
    BEGIN -- generates loadstateandfree
    dlstate[node, qLSTF];
    Coutjump[JumpRet, LabelCCNull];
    RETURN
    END;


  InvalidStateStorageLocation: SIGNAL = CODE;


  dlstate: PROCEDURE [node: TreeIndex, opc: BYTE] =
    BEGIN -- does state move after checking for small currentcontext address
    r: BDOIndex;

    r ← rmakeBDOItem[Cexp[(tb+node).son1]];
    IF cb[r].tag # o OR cb[r].offset.level # CPtr.curctxlvl
     OR cb[r].offset.posn.wd ~IN[0..377B] THEN
      SIGNAL InvalidStateStorageLocation;
    Ciout1[opc, cb[r].offset.posn.wd];
    releaseBDOItem[r];
    RETURN
    END;

  Cblock: PROCEDURE [node: TreeIndex] =
    BEGIN
    bti: BTIndex ← (tb+node).info;
    WITH (bb+bti).info SELECT FROM
      Internal => CPtr.fileindex ← MPtr.textIndex ← sourceIndex;
      ENDCASE;
    ccellalloc[other];
    cb[LOOPHOLE[CPtr.codeptr, OtherCCIndex]].obody ←
        startbody[index: bti];
    (tb+node).son1 ← TreeDefs.updatelist[(tb+node).son1, Cstatement];
    (tb+node).son2 ← TreeDefs.updatelist[(tb+node).son2, Cstatement];
```

```
    ccellalloc[other];
    cb[LOOPHOLE[CPtr.codeptr, OtherCCIndex]].obody ←
        endbody[index: bti];
    END;


  Cifstmt: PROCEDURE [node: TreeIndex] =
    BEGIN -- generates code for an IF statement
    ilabel,elabel: LabelCCIndex;

    elabel←labelalloc[];
    Cflow[(tb+node).son1, FALSE, elabel];
    (tb+node).son2 ← Cstatement[(tb+node).son2];
    IF (tb+node).son3 # empty THEN
      BEGIN
      Coutjump[Jump, ilabel←labelalloc[]];
      insertlabel[elabel];
      (tb+node).son3 ← Cstatement[(tb+node).son3];
      insertlabel[ilabel];
      END
    ELSE insertlabel[elabel];
    RETURN
    END;


  Ccasestmtexp: PUBLIC PROCEDURE [node: TreeIndex, iscasexp: BOOLEAN] RETURNS [valpsize: CARDINAL] =
    BEGIN -- generate code for CASE statment and expression
    casendlabel: LabelCCIndex ← labelalloc[];
    caselpendlabel: LabelCCIndex ← labelalloc[];
    mw: BOOLEAN ← FALSE;
    nswords: CARDINAL ← wordsforoperand[(tb+node).son1];
    nwords: CARDINAL ←
        IF iscasexp THEN SymTabDefs.WordsForType[(tb+node).info] ELSE 0;
    savmwcaseseltlex: se Lexeme ← CPtr.mwcaseseltlex;
    savxtracting: BOOLEAN ← CPtr.xtracting;
    savfirstcaseselread: BOOLEAN ← CPtr.firstcaseselread;
    selpsize, endcasepsize: CARDINAL;
    tlex: se Lexeme ← topostack;
    sCitem: PROCEDURE [t: TreeLink] =
      BEGIN
      WITH t SELECT FROM
        subtree => valpsize ← MAX[Ccaseitem[index, iscasexp, FALSE, nwords, casendlabel, caselpendlabel
**], valpsize];
        ENDCASE;
      IF iscasexp THEN resettomark[];
      RETURN
      END;

    CPtr.xtracting ← CPtr.firstcaseselread ← FALSE;
    RequireStack[0];
    IF iscasexp THEN markstack[];
    IF nswords = 2 THEN
      BEGIN
      CPtr.mwcaseseltlex ← genanonlex[2];
      pushrhs[(tb+node).son1];
      sCassign[CPtr.mwcaseseltlex.lexsei];
      CPtr.firstcaseselread ← TRUE;
      mw ← TRUE;
      END
    ELSE IF nswords > 2 THEN
      BEGIN
      CPtr.mwcaseseltlex ← genanonlex[nswords];
      selpsize ← loadtsonaddress[(tb+node).son1];
      pushlitval[nswords];
      [] ← loadlexaddress[CPtr.mwcaseseltlex];
      IF selpsize # wordlength THEN
        BEGIN Ciout0[qLP]; Ciout0[qBLTL] END
      ELSE Ciout0[qBLT];
      mw ← TRUE;
      END
    ELSE BEGIN pushrhs[(tb+node).son1]; CPtr.firstcaseselread ← TRUE; END;
    valpsize ← wordlength;
    BEGIN ENABLE
      BEGIN
      LogHeapFree => IF iscasexp THEN RESUME[FALSE, topostack];
      MWConstant => IF iscasexp THEN
```

```
        BEGIN
        IF tlex = topostack THEN tlex ← genanonlex[nwords];
        RESUME[tlex];
        END
      END;
      TreeDefs.scanlist[(tb+node).son2, sCitem];
      IF iscasexp THEN
        BEGIN
        IF nwords > 2 THEN
          BEGIN
          endcasepsize ← loadtsonaddress[(tb+node).son3];
          Coutjump[Jump, IF endcasepsize > wordlength THEN caselpendlabel ELSE casendlabel];
          END
        ELSE pushrhs[(tb+node).son3];
        unmarkstack[];
        END
      ELSE (tb+node).son3 ← Cstatement[(tb+node).son3];
    END;
    insertlabel[casendlabel];
    IF valpsize > wordlength THEN Ciout0[qLP];
    insertlabel[caselpendlabel];
    IF mw THEN releasetemplex[CPtr.mwcaseseltlex];
    CPtr.mwcaseseltlex ← savmwcaseseltlex;
    CPtr.firstcaseselread ← savfirstcaseselread;
    CPtr.xtracting ← savxtracting;
    (tb+node).son1 ← TreeDefs.freetree[(tb+node).son1];
    (tb+node).son2 ← TreeDefs.freetree[(tb+node).son2];
    (tb+node).son3 ← TreeDefs.freetree[(tb+node).son3];
    IF (tb+node).nsons > 3 THEN TreeDefs.setshared[(tb+node).son4, FALSE];
    IF tlex # topostack THEN releasetemplex[tlex];
    RETURN
    END;


  newbranches: PROCEDURE [t: TreeLink, itemlabel, faillabel: LabelCCIndex,
                          bt: DESCRIPTOR FOR ARRAY OF LabelCCIndex]
                 RETURNS [new: BOOLEAN] =
    BEGIN -- sees if any new branches need to be added to branch table
    i: CARDINAL;
    snb: PROCEDURE [t: TreeLink] =
      BEGIN
      i ← treeliteralvalue[t];
      IF bt[i] = faillabel THEN
        BEGIN bt[i] ← itemlabel; new ← TRUE; END;
      RETURN
      END;

    new ← FALSE;
    TreeDefs.scanlist[t, snb];
    RETURN
    END;


  Cbranch: PROCEDURE [node: TreeIndex, isexp: BOOLEAN, nwords: CARDINAL, casendlabel, caselpendlabel: L
**abelCCIndex] RETURNS [valpsize: CARDINAL] =
    BEGIN -- generate code for case switch if range is densely packed
    range, i: CARDINAL;
    btcp, savcodeptr: CCIndex;
    faillabel: LabelCCIndex;
    bt: DESCRIPTOR FOR ARRAY OF LabelCCIndex;
    scb: PROCEDURE [t: TreeLink] =
      BEGIN
      bnode: TreeIndex;
      itemlabel: LabelCCIndex;
      vpsize: CARDINAL ← wordlength;

      WITH t SELECT FROM
        subtree =>
          BEGIN
          bnode ← index;
          itemlabel ← labelalloc[];
          IF newbranches[(tb+bnode).son1, itemlabel, faillabel, bt] THEN
            BEGIN
            insertlabel[itemlabel];
            IF isexp THEN
              BEGIN
```

```
            IF nwords > 2 THEN
              BEGIN vpsize ← loadtsonaddress[(tb+bnode).son2];
              adjustacstack[-(vpsize/wordlength)];
              END
            ELSE
              BEGIN pushrhs[(tb+bnode).son2]; adjustacstack[-nwords]; END;
            resettomark[];
            END
          ELSE (tb+bnode).son2 ← Cstatement[(tb+bnode).son2];
          Coutjump[Jump, IF vpsize > wordlength THEN casendlabel
              ELSE caselpendlabel];
          END
        ELSE CodeDefs.FreeChunk[itemlabel, SIZE[label CCItem]];
        IF vpsize # wordlength THEN valpsize ← vpsize;
        RETURN
        END;
      ENDCASE
    END;

  valpsize ← wordlength;
  range ← treeliteralvalue[(tb+node).son2];
  faillabel ← labelalloc[];
  pushrhs[(tb+node).son1];
  RequireStack[1];
  pushlitval[range];
  adjustacstack[-2];
  pop[]; pop[];
  ccellalloc[other];
  cb[LOOPHOLE[CPtr.codeptr, OtherCCIndex]].obody ←
      table[btab: , tablecodebytes: 3, taboffset: ];
  btcp ← CPtr.codeptr;
  Coutjump[JumpCA, faillabel];
  bt ← DESCRIPTOR[SystemDefs.AllocateHeapNode[range], range];
  FOR i IN [0..range) DO bt[i] ← faillabel ENDLOOP;
  TreeDefs.scanlist[(tb+node).son3, scb];
  savcodeptr ← CPtr.codeptr;
  CPtr.codeptr ← btcp;
  FOR i IN [0..range) DO Coutjump[JumpC, bt[i]] ENDLOOP;
  CPtr.codeptr ← savcodeptr;
  insertlabel[faillabel];
  SystemDefs.FreeHeapNode[BASE[bt]];
  RETURN
  END;


  Ccaseitem: PROCEDURE [node: TreeIndex, isexp,isenable: BOOLEAN, nwords: CARDINAL, casendlabel, caselp
**endlabel: LabelCCIndex] RETURNS [valpsize: CARDINAL] =
  BEGIN -- generate code for a CASE item
  itemlabel, faillabel: LabelCCIndex;
  irecord, savcatchoutrecord: recordCSEIndex;
  sei: POINTER [0..TableDefs.TableLimit) TO transfer constructor SERecord;
  savinctxlevel, savoutctxlevel: ContextLevel;
  ictx, octx: CTXIndex ← CTXNull;
  lasson: CARDINAL;
  thisson: CARDINAL ← 0;
  sci: PROCEDURE [t: TreeLink] =
    BEGIN
    IF thisson # lasson THEN
      BEGIN
      Cflow[t, TRUE, itemlabel];
      thisson ← thisson+1;
      END
    ELSE
      BEGIN
      Cflow[t, FALSE, faillabel];
      insertlabel[itemlabel];
      END;
    RETURN
    END;

  IF (tb+node).name = caseswitch THEN
    RETURN[Cbranch[node, isexp, nwords, casendlabel, caselpendlabel]];
  valpsize ← wordlength;
  faillabel ← labelalloc[];
  WITH t1: (tb+node).son1 SELECT FROM
    subtree =>
```

```
      BEGIN
      itemlabel ← labelalloc[];
      IF (tb+t1.index).name # list THEN lasson ← 0
      ELSE lasson ← TreeDefs.listlength[t1]-1;
      TreeDefs.scanlist[t1, sci];
      END;
    ENDCASE => Cflow[t1, FALSE, faillabel];
  IF isexp THEN
    IF nwords > 2 THEN
      BEGIN
      valpsize ← loadtsonaddress[(tb+node).son2];
      adjustacstack[-(valpsize/wordlength)];
      END
    ELSE
      BEGIN pushrhs[(tb+node).son2]; adjustacstack[-nwords]; END
  ELSE
    IF isenable THEN
      BEGIN
      savcatchoutrecord ← CPtr.catchoutrecord;
      sei ← (tb+node).info;
      IF sei # SENull THEN
        BEGIN
        irecord ← (seb+sei).inrecord;
        CPtr.catchoutrecord ← (seb+sei).outrecord;
        IF irecord # recordCSENull THEN
          BEGIN
          ictx ← (seb+irecord).fieldctx;
          savinctxlevel ← (ctxb+ictx).ctxlevel;
          (ctxb+ictx).ctxlevel ← CPtr.curctxlvl;
          END;
        IF CPtr.catchoutrecord # recordCSENull THEN
          BEGIN
          octx ← (seb+CPtr.catchoutrecord).fieldctx;
          savoutctxlevel ← (ctxb+octx).ctxlevel;
          (ctxb+octx).ctxlevel ← CPtr.curctxlvl;
          END;
        END
      ELSE irecord ← CPtr.catchoutrecord ← recordCSENull;
      popinvals[irecord, TRUE];
      (tb+node).son2 ← Cstatement[(tb+node).son2];
      IF ictx # CTXNull THEN (ctxb+ictx).ctxlevel ← savinctxlevel;
      IF octx # CTXNull THEN (ctxb+octx).ctxlevel ← savoutctxlevel;
      CPtr.catchoutrecord ← savcatchoutrecord;
      END
    ELSE (tb+node).son2 ← Cstatement[(tb+node).son2];
  Coutjump[Jump, IF valpsize = wordlength THEN casendlabel
      ELSE caselpendlabel];
  insertlabel[faillabel];
  RETURN
  END;


Cdostmt: PROCEDURE [rootnode: TreeIndex] =
  BEGIN --  generates code for all the loop statments
  steploop, tempindex, tempend, uploop, forseqloop, unsigned, long: BOOLEAN ← FALSE;
  t, Sson, Eson: TreeLink;
  node, node2: TreeIndex;
  inttype: NodeName;
  indexlex: se Lexeme;
  endlex: Lexeme;
  toplabel: LabelCCIndex ← labelalloc[];
  startlabel: LabelCCIndex;
  finlabel: LabelCCIndex ← labelalloc[];
  endlabel, looplabel: LabelCCIndex;
  labelmark: EXLRIndex ← getlabelmark[];
  updateCV: PROCEDURE [loadlong: BOOLEAN] =
    BEGIN
    IF long THEN
      BEGIN
      IF loadlong THEN
        BEGIN RequireStack[0]; pushlex[indexlex]; END
      ELSE RequireStack[2];
      pushlitval[1]; pushlitval[0];
      CioutO[IF uploop THEN qDADD ELSE qDSUB];
      sCassign[indexlex.lexsei];
      END
```

```
      ELSE Ciout0[IF uploop THEN qINC ELSE qDEC];
      END;

    -- set up for EXIT clause

    [exit: endlabel, loop: looplabel] ← makeEXITlabel[];
    TreeDefs.scanlist[(tb+rootnode).son5, Clabelcreate];

    -- handle initialization node

    t ← (tb+rootnode).son1;
    WITH t SELECT FROM
      subtree =>
       IF t # empty THEN
        BEGIN
        node ← index;
        SELECT (tb+node).name FROM
          forseq =>
            BEGIN
            forseqloop ← TRUE;
            pushrhs[(tb+node).son2];
            insertlabel[toplabel];
            WITH (tb+node).son1 SELECT FROM
              symbol => sCassign[index];
              ENDCASE;
            END;
          upthru, downthru =>
            BEGIN
            steploop ← TRUE;
            uploop ← (tb+node).name = upthru;
            WITH (tb+node).son2 SELECT FROM
              subtree =>
                BEGIN
                node2 ← index;
                inttype ← (tb+node2).name;
                IF (tb+node2).attr1 THEN
                  BEGIN
                  long ← TRUE;
                  IF (tb+node2).attr2 THEN SIGNAL CPtr.CodeNotImplemented;
                  END
                ELSE unsigned ← (tb+node2).attr2;
                END;
              ENDCASE;
            WITH (tb+node).son1 SELECT FROM
              subtree => -- son1 is empty
                BEGIN
                indexlex ← genanonlex[IF long THEN 2 ELSE 1];
                tempindex ← TRUE;
                END;
              symbol => indexlex ← Lexeme[se[index]];
              ENDCASE;
            WITH (tb+node).son2 SELECT FROM
              subtree =>
                BEGIN
                IF uploop THEN
                  BEGIN Sson ← (tb+node2).son1; Eson ← (tb+node2).son2; END
                ELSE
                  BEGIN
                  SELECT inttype FROM
                    intCO => inttype ← intOC;
                    intOC => inttype ← intCO;
                    ENDCASE;
                  Sson ← (tb+node2).son2;
                  Eson ← (tb+node2).son1;
                  END;
                WITH e: Eson SELECT FROM
                  literal =>
                    WITH e.info SELECT FROM
                      word => endlex ←
                         Lexeme[literal[word[index]]];
                      ENDCASE => P5ADefs.P5Error[769];
                  ENDCASE =>
                    BEGIN
                    pushrhs[e]; tempend ← TRUE;
                    sCassign[
                      (endlex ← genanonlex[IF long THEN 2 ELSE 1]).lexsei];
```

```
            END;
          startlabel ← labelalloc[];
          IF long THEN RequireStack[0];
          pushrhs[Sson];
          IF long THEN
            IF inttype = intOO THEN
              BEGIN adjustacstack[-2]; pop[]; pop[] END
            ELSE sCassign[indexlex.lexsei];
          IF inttype = intCC AND (tempend OR ~treeliteral[Sson]) THEN
              BEGIN -- earlier passes check for empty intervals
              IF long THEN BEGIN Ciout0[qPUSH]; Ciout0[qPUSH] END;
              pushlex[endlex];
              IF long THEN BEGIN Ciout0[qDCOMP]; pushlitval[0] END;
              Coutjump[
                  IF unsigned THEN IF uploop THEN UJumpG ELSE UJumpL
                  ELSE IF uploop THEN JumpG ELSE JumpL, finlabel];
              IF ~long THEN Ciout0[qPUSH];
              END;
          IF ~long THEN BEGIN adjustacstack[-1]; pop[]; END;
          Coutjump[Jump, startlabel];
          insertlabel[toplabel];
          IF ~long THEN Ciout0[qPUSH];
          SELECT inttype FROM
            intCC => BEGIN updateCV[TRUE]; insertlabel[startlabel]; END;
            intOC => updateCV[TRUE];
            intCO, intOO => NULL;
            ENDCASE;
          IF ~long THEN sCassign[indexlex.lexsei];
          END;
        ENDCASE;
      END;
    ENDCASE;
  END
 ELSE insertlabel[toplabel];
ENDCASE;

-- now the pre-body test

IF (tb+rootnode).son2 # empty THEN
  Cflow[(tb+rootnode).son2, FALSE, finlabel];

-- ignore the opens

-- (tb+node).son3;

-- now the body

(tb+rootnode).son4 ← Cstatement[(tb+rootnode).son4];

-- now (update and) test the control variable

insertlabel[looplabel];
IF steploop THEN
  BEGIN
  IF long AND inttype = intOC THEN insertlabel[startlabel];
  pushlex[indexlex];
  SELECT inttype FROM
    intCC => NULL;
    intCO => BEGIN updateCV[FALSE]; insertlabel[startlabel]; END;
    intOC => IF ~long THEN insertlabel[startlabel];
    intOO => BEGIN insertlabel[startlabel]; updateCV[FALSE]; END;
    ENDCASE;
  IF long THEN SELECT inttype FROM
    intCO, intOO => BEGIN Ciout0[qPUSH]; Ciout0[qPUSH] END;
    ENDCASE;
  pushlex[endlex];
  IF long THEN BEGIN Ciout0[qDCOMP]; pushlitval[0] END;
  Coutjump[
        IF unsigned THEN IF uploop THEN UJumpL ELSE UJumpG
        ELSE IF uploop THEN JumpL ELSE JumpG, toplabel];
  Coutjump[Jump, finlabel];
  IF tempend THEN releasetemplex[LOOPHOLE[endlex, se Lexeme]];
  IF tempindex THEN releasetemplex[indexlex];
  END
ELSE
  BEGIN
```

```
    IF forseqloop THEN
      BEGIN
      WITH (tb+rootnode).son1 SELECT FROM
        subtree => pushrhs[(tb+index).son3 ! LogHeapFree =>
          RESUME[FALSE, topostack]];
        ENDCASE;
      END;
    Coutjump[Jump, toplabel];
    END;
  clearstack[]; CPtr.acstack ← 0;

  -- now the labelled EXITs

  Clabellist[(tb+rootnode).son5, endlabel];
  poplabels[labelmark];

  -- finally the FINISHED clause

  insertlabel[finlabel];
  (tb+rootnode).son6 ← Cstatement[(tb+rootnode).son6];
  insertlabel[endlabel];
  RETURN
  END;


Ccatchphrase: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN -- process a catchphrase at procedure call
  aroundlabel: LabelCCIndex ← labelalloc[];
  savcfs: CARDINAL ← CPtr.cfs;
  r: CodeCCIndex;

  CPtr.catchcount ← CPtr.catchcount + 1;
  Ciout1[qCATCH, 0];
  r ← LOOPHOLE[CPtr.codeptr, CodeCCIndex];
  Coutjump[JumpA, aroundlabel];
  sCcatchphrase[node];
  cb[r].parameters[1] ← CPtr.cfs;
  insertlabel[aroundlabel];
  CPtr.catchcount ← CPtr.catchcount - 1;
  CPtr.cfs ← savcfs;
  RETURN
  END;


Cenable: PROCEDURE [node: TreeIndex] =
  BEGIN -- generate code for an ENABLE
  aroundlabel: LabelCCIndex ← labelalloc[];
  enablelabel: LabelCCIndex;
  savactenable: LabelCCIndex ← CPtr.actenable;
  savcfs: CARDINAL ← CPtr.cfs;

  CPtr.catchcount ← CPtr.catchcount + 1;
  Coutjump[JumpA,aroundlabel];
  enablelabel ← createlabel[];
  WITH (tb+node).son1 SELECT FROM
    subtree => sCcatchphrase[index];
    ENDCASE;
  insertlabel[aroundlabel];
  CPtr.actenable ← enablelabel;
  CPtr.catchcount ← CPtr.catchcount -1;
  (tb+node).son2 ← Cstatement[(tb+node).son2];
  CPtr.actenable ← savactenable;
  CPtr.cfs ← savcfs;
  RETURN
  END;


sCcatchphrase: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN -- main subr for catchphrases and ENABLEs
  savfirstcaseselread: BOOLEAN ← CPtr.firstcaseselread;
  endlabel: LabelCCIndex ← labelalloc[];
  oldstkptr: StkIndex ← newstack[];
  msgtemp, sigtemp: se Lexeme;
  savactenable: LabelCCIndex;
  tempstate: TempStateRecord;
  sscc: PROCEDURE [t: TreeLink] =
```

```
      BEGIN
      WITH t SELECT FROM
        subtree => [] ← Ccaseitem[
          node:index, isexp:FALSE, isenable:TRUE,
          nwords:0, casendlabel:endlabel, caselpendlabel:endlabel];
        ENDCASE;
      RETURN
      END;

    CPtr.curctxlvl ← CPtr.curctxlvl + 1;
    pushtempstate[@tempstate, (tb+node).info];
    IF CPtr.actenable # LabelCCNull THEN
      BEGIN
      sigtemp ← genanonlex[1];
      msgtemp ← genanonlex[1];
      adjustacstack[1]; incrstack[2];
      stackoff[]; Ciout1[qLL,ControlDefs.localbase+1]; stackon[];
      sCassign[msgtemp.lexsei];
      sCassign[sigtemp.lexsei];
      pushlex[sigtemp];
      adjustacstack[-1];
      pop[];
      END;
    CPtr.firstcaseselread ← TRUE;
    adjustacstack[1];
    incrstack[1];
    savactenable ← CPtr.actenable;
    CPtr.actenable ← LabelCCNull;
    TreeDefs.scanlist[(tb+node).son1, sscc];
    stackoff[];
    IF (tb+node).son2 # empty THEN
      BEGIN
      IF (tb+node).son1 = empty THEN Ciout0[qPOP];
      stackon[];
      (tb+node).son2 ← Cstatement[(tb+node).son2];
      END;
    CPtr.actenable ← savactenable;
    insertlabel[endlabel];
    stackoff[];
    IF CPtr.actenable # LabelCCNull THEN
      BEGIN
      pushlex[sigtemp];
      pushlex[msgtemp];
      Ciout1[qSL,ControlDefs.localbase+1];
      adjustacstack[-1];
      Coutjump[Jump,CPtr.actenable];
      releasetemplex[msgtemp];
      releasetemplex[sigtemp];
      END
    ELSE
      BEGIN
      pushlitval[0];
      adjustacstack[-1];
      Ciout0[qRET];
      Coutjump[JumpRet,LabelCCNull];
      END;
    stackon[];
    CPtr.curctxlvl ← CPtr.curctxlvl-1;
    CPtr.firstcaseselread ← savfirstcaseselread;
    CPtr.cfs ← CPtr.framesz;
    poptempstate[@tempstate];
    CPtr.cfs ← computeframesize[CPtr.cfs];
    IF CPtr.cfs > 377B THEN SIGNAL CatchFrameTooLarge;
    restoreoldstack[oldstkptr];
    RETURN
    END;

  Cnotify: PROCEDURE [node: TreeIndex] =
    BEGIN
    Ciout0[IF loadtsonaddress[(tb+node).son1] = wordlength THEN qNOTIFY
        ELSE qNOTIFYL];
    END;

  Cbroadcast: PROCEDURE [node: TreeIndex] =
    BEGIN
    Ciout0[IF loadtsonaddress[(tb+node).son1] = wordlength THEN qBCAST
```

```
        ELSE qBCASTL];
    END;

  END...
```