```
-- Flow.mesa last modified by Sweet, July 18, 1978  9:55 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [BYTE, charlength, wordlength],
  Code: FROM "code" USING [acstack, catchcount, CodeNotImplemented, codeptr, curctxlvl, firstcaseselrea
**d, mwcaseseltlex],
  CodeDefs: FROM "codedefs" USING [BDOIndex, CCNull, ChunkBase, CompareClass, EXLabelRecord, EXLRIndex,
** EXLRNull, FreeChunk, GetChunk, JumpType, LabelCCIndex, Lexeme, topostack],
  ComData: FROM "comdata",
  ControlDefs: FROM "controldefs" USING [localbase, returnOffset],
  FOpCodes: FROM "fopcodes" USING [qDCOMP, qLL, qLP, qPUSH, qSFC],
  P5ADefs: FROM "p5adefs" USING [adjustacstack, Ciout0, Ciout1, Cload, copyBDOItem, Coutjump, Csyscall,
** Csyscalln, dumpstack, easilyaddressed, freeheaplex, genanonlex, genBDOItem, gentemplex, GetFrame, in
**crstack, labelalloc, loadlexaddress, loadtsonaddress, loadtsonchars, LogHeapFree, LongTreeAddress, ma
**rkstack, operandtype, pop, RequireStack, rmakeBDOItem, sCassign, treeliteral, wordsforoperand],
  P5BDefs: FROM "p5bdefs" USING [Cexp, Cstatement, MWConstant, pushlex, pushlitval, pushrhs],
  P5StmtExprDefs: FROM "p5stmtexprdefs",
  SDDefs: FROM "sddefs" USING [sBLTE, sBLTEC, sBLTECL, sBLTEL, sBYTBLTE, sBYTBLTEC, sBYTBLTECL, sBYTBLT
**EL, sFCOMP],
  SymDefs: FROM "symdefs" USING [ContextLevel, CSEIndex, CTXIndex, HTIndex, HTNull, ISEIndex, SEIndex,
**setype],
  SymTabDefs: FROM "symtabdefs" USING [BitsForType, Cardinality],
  TableDefs: FROM "tabledefs" USING [TableBase, TableLimit, TableNotifier],
  TreeDefs: FROM "treedefs" USING [empty, NodeName, scanlist, TreeIndex, TreeLink, treetype];

DEFINITIONS FROM CodeDefs;

Flow: PROGRAM
    IMPORTS CPtr: Code, CodeDefs, P5ADefs, P5BDefs, TreeDefs, SymTabDefs
    EXPORTS CodeDefs, P5ADefs, P5StmtExprDefs =
  BEGIN
  OPEN P5ADefs, P5BDefs;

  -- imported definitions

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

  ContextLevel: TYPE = SymDefs.ContextLevel;
  CTXIndex: TYPE = SymDefs.CTXIndex;
  HTIndex: TYPE = SymDefs.HTIndex;
  HTNull: HTIndex = SymDefs.HTNull;
  ISEIndex: TYPE = SymDefs.ISEIndex;
  SEIndex: TYPE = SymDefs.SEIndex;

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

  CRLabelRecord: TYPE = RECORD [
        free: BOOLEAN,
        retrylabel, contlabel: LabelCCIndex,
        crcc: CARDINAL];

  CRLRIndex: TYPE = ChunkBase RELATIVE POINTER [0..TableDefs.TableLimit) TO CRLabelRecord;
  CRLRNull: CRLRIndex = LOOPHOLE[TableDefs.TableLimit-1];

  labelstack: EXLRIndex ← EXLRNull;

  CRlabel: CRLRIndex ← CRLRNull;

  UndeclaredLabel: SIGNAL[HTIndex] = CODE;

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

  FlowNotify: PUBLIC TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever table area is repacked
    seb ← base[SymDefs.setype];
    tb ← base[TreeDefs.treetype];
    cb ← LOOPHOLE[tb];
    RETURN
    END;
```

```
  JumpNN: PACKED ARRAY NodeName[relE..relLE] OF JumpType ← [
        JumpE, JumpN, JumpL, JumpGE, JumpG, JumpLE];

  UJumpNN: PACKED ARRAY NodeName[relE..relLE] OF JumpType ← [
        JumpE, JumpN, UJumpL, UJumpGE, UJumpG, UJumpLE];

  RNN: ARRAY NodeName[relE..relLE] OF NodeName = [
        relE, relN, relG, relLE, relL, relGE];

  CNN: ARRAY NodeName[relE..relLE] OF NodeName = [
        relN, relE, relGE, relL, relLE, relG];

  PushOnly: PROCEDURE [t: TreeLink] =
    BEGIN
    IF t # empty THEN BEGIN RequireStack[0]; pushrhs[t] END
    ELSE BEGIN pushrhs[t]; RequireStack[1] END;
    RETURN
    END;

  Cflow: PUBLIC PROCEDURE [t: TreeLink, tf: BOOLEAN, label: LabelCCIndex] =
    BEGIN -- produces code to jump to label on condition tf
    node: TreeIndex;
    l: se Lexeme;
    label1: LabelCCIndex;
    sw: BOOLEAN;

    WITH t SELECT FROM
      symbol =>
        BEGIN
        RequireStack[0];
        l ← Lexeme[lexvalue: se[index]];
        pushlex[l];
        pushlitval[0];
        Coutjump[IF tf THEN JumpN ELSE JumpE, label];
        RETURN
        END;
      subtree =>
        BEGIN
        node ← index;
        SELECT (tb+node).name FROM
          and, or =>
            BEGIN ENABLE LogHeapFree => RESUME[FALSE, topostack];
            sw ← IF (tb+node).name = and THEN tf ELSE ~tf;
            IF sw THEN
              BEGIN
              label1←labelalloc[];
              Cflow[(tb+node).son1, ~tf, label1];
              Cflow[(tb+node).son2, tf, label];
              insertlabel[label1];
              END
            ELSE
              BEGIN
              Cflow[(tb+node).son1, tf, label];
              Cflow[(tb+node).son2, tf, label];
              END;
            END;
          not => Cflow[(tb+node).son1, ~tf, label];
          in => Cfin[t, tf, label];
          notin => Cfin[t, ~tf, label];
          relE, relN, relL, relGE, relG, relLE =>
            sCfrel[node, tf, label];
          ENDCASE =>
            BEGIN
            PushOnly[t];
            pushlitval[0];
            Coutjump[IF tf THEN JumpN ELSE JumpE, label];
            END;
        END;
      ENDCASE;
    RETURN
    END;

  CompareOps: ARRAY CompareClass OF ARRAY BOOLEAN OF
    PACKED ARRAY [1..2] OF BYTE ← -- try to make this better *********
      [[[SDDefs.sBLTE,SDDefs.sBLTEL],
```

```
        [SDDefs.sBLTEC,SDDefs.sBLTECL]],
       [[SDDefs.sBYTBLTE,SDDefs.sBYTBLTEL],
        [SDDefs.sBYTBLTEC,SDDefs.sBYTBLTECL]]];

CompareFn: PUBLIC PROCEDURE [class: CompareClass, code: BOOLEAN, length: [1..2]] RETURNS [BYTE] =
  BEGIN
  RETURN[CompareOps[class][code][length]];
  END;


sCfrel: PROCEDURE [node: TreeIndex, tf: BOOLEAN, label: LabelCCIndex] =
  BEGIN -- main subroutine of Cflow for handling relationals
  t1: TreeLink ← (tb+node).son1;
  t2, tt: TreeLink;
  sei: SymDefs.CSEIndex;
  n: NodeName ← (tb+node).name;
  nwords: INTEGER;
  lex1, lex2: se Lexeme ← topostack;
  tlex: se Lexeme;
  function: CompareClass ← word;
  plength: [1..2];
  code: BOOLEAN ← FALSE;
  real: BOOLEAN;
  t1addrsize, t2addrsize: CARDINAL;

  IF treeliteral[t1] THEN
    BEGIN
    n ← RNN[n];
    t2 ← t1; t1 ← (tb+node).son2;
    END
  ELSE t2 ← (tb+node).son2;
  IF ~tf THEN n ← CNN[n];

    BEGIN
    IF t2.tag = literal THEN GO TO notpacked;
    sei ← operandtype[t2];
    WITH (seb+sei) SELECT FROM
      array =>
        BEGIN
        IF ~packed OR SymTabDefs.BitsForType[componenttype] > 8 THEN GO TO notpacked;
        nwords ← SymTabDefs.Cardinality[indextype];
        IF nwords <= 4 THEN
          BEGIN
          IF t1 # empty THEN RequireStack[0];
          loadtsonchars[t1, nwords];
          IF t1 = empty THEN RequireStack[(nwords+1)/2];
          loadtsonchars[t2, nwords];
          IF nwords <= 2 THEN Coutjump[UJumpNN[n], label]
          ELSE
            BEGIN
            Ciout0[FOpCodes.qDCOMP];
            pushlitval[0];
            Coutjump[JumpNN[n], label];
            END;
          RETURN
          END
        ELSE function ← byte;
        END;
      ENDCASE => GO TO notpacked;
    EXITS
      notpacked => nwords ← wordsforoperand[t2];
    END;
  IF nwords > 1 THEN
    BEGIN
    IF nwords = 2 THEN
      BEGIN
      RequireStack[0];
      IF (real ← (tb+node).attr1 AND (tb+node).attr2) THEN markstack[];
      IF t1 = empty THEN
        BEGIN
        CPtr.firstcaseselread←FALSE;
        pushlex[CPtr.mwcaseseltlex]
        END
      ELSE pushrhs[t1];
      RequireStack[2];
      pushrhs[t2];
```

```
      IF real THEN Csyscalln[SDDefs.sFCOMP,1] ELSE Ciout0[FOpCodes.qDCOMP];
      pushlitval[0];
      Coutjump[JumpNN[n], label];
      RETURN
      END;
    dumpstack[]; markstack[];
    WITH t1 SELECT FROM
      subtree =>
        IF (tb+index).name = mwconst THEN
          BEGIN tt ← t1; t1 ← t2; t2 ← tt END;
      ENDCASE;
    WITH t1 SELECT FROM
      subtree =>
        IF (tb+index).name = mwconst THEN SIGNAL CPtr.CodeNotImplemented;
      ENDCASE;
    t1addrsize ← IF t1 = empty THEN loadlexaddress[CPtr.mwcaseseltlex]
      ELSE loadtsonaddress[t1
        !LogHeapFree => IF calltree = t1 THEN RESUME[TRUE, lex1 ← genanonlex[1]]
      ];
    IF t1addrsize = wordlength AND LongTreeAddress[t2] THEN
      Ciout0[FOpCodes.qLP];
    pushlitval[nwords];
    t2addrsize ← loadtsonaddress[t2
      !LogHeapFree =>
        IF calltree = t2 THEN RESUME[TRUE, lex2 ← genanonlex[1]];
      MWConstant =>
        BEGIN
        code ← TRUE;
        pushlitval[cOffset];
        t2addrsize ← t1addrsize;
        CONTINUE;
        END
      ];
    IF t1addrsize # t2addrsize THEN
      BEGIN
      IF t1addrsize > wordlength THEN Ciout0[FOpCodes.qLP];
      plength ← 2;
      END
    ELSE plength ← t1addrsize/wordlength;
    Csyscall[CompareFn[function, code, plength]];
    incrstack[1]; CPtr.acstack ← 1;
    IF lex1 # topostack OR lex2 # topostack THEN
      BEGIN
      tlex ← gentemplex[1];
      sCassign[tlex.lexsei];
      IF lex1 # topostack THEN freeheaplex[lex1];
      IF lex2 # topostack THEN freeheaplex[lex2];
      pushlex[tlex];
      END;
    pushlitval[0];
    Coutjump[IF n # relE THEN JumpE ELSE JumpN, label];
    RETURN
    END;
  PushOnly[t1];
  pushrhs[t2];
  Coutjump[IF (tb+node).attr2 THEN UJumpNN[n] ELSE JumpNN[n], label];
  RETURN
  END;


Cfin: PUBLIC PROCEDURE [t: TreeLink, tf: BOOLEAN, label: LabelCCIndex] =
  BEGIN -- generates code for IN expression in flow context
  node: TreeIndex;
  n: NodeName;
  l: LabelCCIndex ← labelalloc[];
  jumpNN: POINTER TO PACKED ARRAY NodeName[relE..relLE] OF JumpType;
  double, real: BOOLEAN;
  r: BDOIndex;
  t1: TreeLink;
  tlex: se Lexeme;

  WITH t SELECT FROM
    subtree =>
      BEGIN
      node ← index;
      t1 ← (tb+node).son1;
```

```
      double ← (tb+node).attr1;
      IF double THEN
        BEGIN
        RequireStack[0];
        IF (real ← double AND (tb+node).attr2) THEN markstack[];
        IF t1 = empty THEN
          BEGIN
          CPtr.firstcaseselread←FALSE;
          pushlex[CPtr.mwcaseseltlex];
          r ← rmakeBDOItem[CPtr.mwcaseseltlex];
          END
        ELSE
          BEGIN
          r ← rmakeBDOItem[Cexp[t1]];
          IF ~easilyaddressed[r] THEN
            BEGIN
            Cload[r];
            tlex ← gentemplex[2];
            sCassign[tlex.lexsel];
            r ← rmakeBDOItem[tlex];
            END;
          Cload[copyBDOItem[r]];
          END;
        END
      ELSE PushOnly[t1];
      WITH (tb+node).son2 SELECT FROM
        subtree =>
          BEGIN ENABLE LogHeapFree => RESUME[FALSE, topostack];
          node ← index;
          jumpNN ← IF ~double AND (tb+node).attr2 THEN @UJumpNN ELSE @JumpNN;
          n ← (tb+node).name;
          pushrhs[(tb+node).son1];
          IF double THEN
            BEGIN
            IF real THEN Csyscall[SDDefs.sFCOMP,1]
            ELSE Ciout0[FOpCodes.qDCOMP];
            pushlitval[0];
            END;
          SELECT n FROM
            intOO,intOC => Coutjump[jumpNN[relLE],IF tf THEN 1 ELSE label];
            intCO,intCC => Coutjump[jumpNN[relL],IF tf THEN 1 ELSE label];
            ENDCASE;
          IF double THEN
            BEGIN
            IF real THEN markstack[];
            Cload[r];
            END
          ELSE Ciout0[FOpCodes.qPUSH];
          pushrhs[(tb+node).son2];
          IF double THEN
            BEGIN
            IF real THEN Csyscall[SDDefs.sFCOMP,1]
            ELSE Ciout0[FOpCodes.qDCOMP];
            pushlitval[0];
            END;
          SELECT n FROM
            intOO,intCO => Coutjump[IF tf THEN jumpNN[relL] ELSE jumpNN[relGE],label];
            intOC,intCC => Coutjump[IF tf THEN jumpNN[relLE] ELSE jumpNN[relG],label];
            ENDCASE;
          insertlabel[l];
          RETURN
          END;
        ENDCASE
      END;
    ENDCASE
  END;

Ccatchmark: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN -- process a CONTINUEd or RETRYed statement
  savCRlabel: CRLRIndex ← CRlabel;
  l: CRLRIndex ← CodeDefs.GetChunk[SIZE[CRLabelRecord]];
  elabel: LabelCCIndex;

  CRlabel ← l;
  cb[l].free ← FALSE;
```

```
    insertlabel[cb[l].retrylabel ← labelalloc[]];
    elabel ← cb[l].contlabel ← labelalloc[];
    cb[l].crcc ← CPtr.catchcount;
    (tb+node).son1 ← Cstatement[(tb+node).son1];
    insertlabel[elabel];
    CRlabel ← savCRlabel;
    CodeDefs.FreeChunk[l, SIZE[CRLabelRecord]];
    RETURN
    END;


  Clabel: PUBLIC PROCEDURE [node: TreeIndex] =
    BEGIN -- process an exitable block
    elabel: LabelCCIndex ← labelalloc[];
    labelmark: EXLRIndex ← labelstack;

    TreeDefs.scanlist[(tb+node).son2, Clabelcreate];
    (tb+node).son1 ← Cstatement[(tb+node).son1];
    Coutjump[Jump, elabel];
    Clabellist[(tb+node).son2, elabel];
    insertlabel[elabel];
    poplabels[labelmark];
    RETURN
    END;


  getlabelmark: PUBLIC PROCEDURE RETURNS [EXLRIndex] =
    BEGIN RETURN[labelstack] END;


  poplabels: PUBLIC PROCEDURE [labelmark: EXLRIndex] =
    BEGIN
    oldl: EXLRIndex;

    UNTIL labelstack = labelmark DO
      oldl ← labelstack;
      labelstack ← cb[labelstack].thread;
      CodeDefs.FreeChunk[oldl, SIZE[EXLabelRecord]];
      ENDLOOP;
    RETURN
    END;


  Clabellist: PUBLIC PROCEDURE [t: TreeLink, elabel: LabelCCIndex] =
    BEGIN -- generates code for labels
    Clabelitem: PROCEDURE [t: TreeLink] =
      BEGIN -- generates code for a labelitem
      WITH t SELECT FROM
        subtree =>
          BEGIN
          TreeDefs.scanlist[(tb+index).son1, putlabel];
          (tb+index).son2 ← Cstatement[(tb+index).son2];
          Coutjump[Jump, elabel];
          RETURN
          END;
        ENDCASE
    END;

    TreeDefs.scanlist[t, Clabelitem];
    RETURN
    END;


  putlabel: PROCEDURE [t: TreeLink] =
    BEGIN
    WITH t SELECT FROM
      hash => insertlabel[cb[findlabel[index]].labelcci];
      ENDCASE;
    RETURN
    END;


  Clabelcreate: PUBLIC PROCEDURE [t: TreeLink] =
    BEGIN -- sets up label cells for labels
    WITH t SELECT FROM
```

```
    subtree => TreeDefs.scanlist[(tb+index).son1, pushlabel];
    ENDCASE;
  RETURN
  END;


pushlabel: PROCEDURE [t: TreeLink] =
  BEGIN -- stacks a label for an EXIT clause
  l: EXLRIndex ← CodeDefs.GetChunk[SIZE[EXLabelRecord]];

  WITH t SELECT FROM
    hash =>
      BEGIN
      cb[l] ←
        EXLabelRecord[free: FALSE, thread: labelstack, labelhti: index,
              labelcc: CPtr.catchcount, labelcci: labelalloc[]];
      labelstack ← l;
      RETURN;
      END;
    ENDCASE
  END;


makeEXITlabel: PUBLIC PROCEDURE RETURNS [exit, loop: LabelCCIndex] =
  BEGIN -- sets up anonymous label for EXITs
  l: EXLRIndex ← CodeDefs.GetChunk[SIZE[EXLabelRecord]];

  exit ← labelalloc[];
  loop ← labelalloc[];
  cb[l] ←
      EXLabelRecord[free: FALSE, thread: labelstack, labelhti: HTNull,
              labelcc: CPtr.catchcount, labelcci: loop];
  labelstack ← l;
  l ← CodeDefs.GetChunk[SIZE[EXLabelRecord]];
  cb[l] ←
      EXLabelRecord[free: FALSE, thread: labelstack, labelhti: HTNull,
              labelcc: CPtr.catchcount, labelcci: exit];
  labelstack ← l;
  RETURN
  END;


findlabel: PROCEDURE [hti: HTIndex] RETURNS [c: EXLRIndex] =
  BEGIN -- searches down label stack for label hti
  FOR c ← labelstack, cb[c].thread UNTIL c = EXLRNull DO
    IF cb[c].labelhti = hti THEN RETURN
    ENDLOOP;
  SIGNAL UndeclaredLabel[hti];
  RETURN
  END;


Cretry: PUBLIC PROCEDURE =
  BEGIN -- process RETRY statement
  retcontexit[cb[CRlabel].crcc, cb[CRlabel].retrylabel];
  RETURN
  END;


Ccontinue: PUBLIC PROCEDURE =
  BEGIN -- process CONTINUE statement
  retcontexit[cb[CRlabel].crcc, cb[CRlabel].contlabel];
  RETURN
  END;


Cexit: PUBLIC PROCEDURE =
  BEGIN  -- generate code for EXIT
  l: EXLRIndex ← findlabel[HTNull];

  retcontexit[cb[l].labelcc, cb[l].labelcci];
  RETURN
  END;

Cloop: PUBLIC PROCEDURE =
  BEGIN  -- generate code for EXIT
```

```
    l: EXLRIndex ← findlabel[HTNull];

    l ← cb[l].thread;
    retcontexit[cb[l].labelcc, cb[l].labelcci];
    RETURN
    END;


  Cgoto: PUBLIC PROCEDURE [node: TreeIndex] =
    BEGIN  -- generate code for GOTO
    l: EXLRIndex;

    WITH (tb+node).son1 SELECT FROM
      hash => l ← findlabel[index];
      ENDCASE;
    retcontexit[cb[l].labelcc, cb[l].labelcci];
    RETURN
    END;


  retcontexit: PROCEDURE [cc: CARDINAL, lc: LabelCCIndex] =
    BEGIN -- process EXIT/REPEAT statement
    r: BDOIndex;

    IF CPtr.catchcount = cc THEN Coutjump[Jump, lc]
    ELSE
      BEGIN
      markstack[];
      r ← genBDOItem[];
      cb[r].tag ← o;
      cb[r].offset ←
                [level: CPtr.curctxlvl-(CPtr.catchcount-cc-1),
                 posn:[wd: ControlDefs.localbase, bd: 0],
                 size: AltoDefs.wordlength];
      GetFrame[r];
      pushlex[[bdo[r]]];
      pushlitval[-1];
      Ciout1[FOpCodes.qLL, ControlDefs.returnOffset];
      Ciout0[FOpCodes.qSFC];
      adjustacstack[-2];
      pop[];  pop[];
      Coutjump[Jump, lc];
      END;
    RETURN
    END;


  insertlabel: PUBLIC PROCEDURE [c: LabelCCIndex] =
    BEGIN -- puts a label chunk in the code stream
    IF CPtr.codeptr # CCNull THEN
      BEGIN
      cb[c].flink ← cb[CPtr.codeptr].flink;
      IF cb[CPtr.codeptr].flink # CCNull THEN
        cb[cb[CPtr.codeptr].flink].blink ← c;
      cb[CPtr.codeptr].flink ← c;
      END
    ELSE cb[c].flink ← CCNull;
    cb[c].blink ← CPtr.codeptr;
    CPtr.codeptr ← c;
    RETURN
    END;


END...
```