```
-- Driver.mesa, last modified by Sweet, Aug 29, 1978 11:44 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [Address, BYTE, wordlength],
  Code: FROM "code" USING [acstack, actenable, catchcount, catchoutrecord, CodePassInconsistancy, codep
**tr, curbodyretlabel, curctxlvl, fileindex, firstcaseselread, framesz, mainBody, StackNotEmptyAtStatem
**ent, stking, tempcontext, tempstart, xtracting, xtractsei, ZEROlexeme],
  CodeDefs: FROM "codedefs" USING [AddressNotify, BDOIndex, BDONull, CallsNotify, CCIndex, CCItem, CCNu
**ll, ChunkBase, ChunkIndex, CodeCCIndex, CodeChunkType, ExpressionNotify, FinalNotify, FlowExpressionN
**otify, FlowNotify, FullBitAddress, JumpCCIndex, JumpCCNull, JumpsNotify, JumpType, LabelCCIndex, Labe
**lCCNull, Lexeme, MaxParmsInStack, NULLfileindex, OutCodeNotify, PeepholeNotify, StackNotify, Statemen
**tNotify, StoreNotify, topostack],
  ComData: FROM "comdata" USING [bodyIndex, bodyRoot, mainBody, nErrors, objectBytes, objectFrameSize,
**stopping, textIndex, typeSTRING],
  ControlDefs: FROM "controldefs" USING [codebaseOffset, framelink, globalbase, localbase, MaxAllocSlot
**, returnOffset],
  ErrorDefs: FROM "errordefs" USING [errorsei],
  FOpCodes: FROM "fopcodes" USING [qALLOC, qDUP, qFREE, qJ, qJREL, qLADRB, qLI, qLINKB, qLL, qME, qMEL,
** qMXD, qMXDL, qPUSH, qRET, qSG],
  InlineDefs: FROM "inlinedefs",
  LitDefs: FROM "litdefs" USING [FindLiteral, LiteralValue],
  OpTableDefs: FROM "optabledefs" USING [instlength],
  P5ADefs: FROM "p5adefs" USING [AddressInit, adjustacstack, chkacstack, chkrandsonstack, clearstack, c
**opyBDOItem, CRassign, Csyserror, dumpstack, freetemplist, gentemplex, incrstack, insertlabel, loadtso
**naddress, LogHeapFree, makeTOSaddrBDOItem, NumberOfParams, P5Error, pop, purgependtemplist, PushEffec
**t, putrandsonstack, releaseBDOItem, sCassign, slCassign, StackFinal, StackInit, stackoff, stackon, tr
**ansferconstruct],
  P5BDefs: FROM "p5bdefs" USING [Cfixup, Cstatement, endcodefile, outbinary, ProcessGlobalStrings, Proc
**essLocalStrings, pushlex, pushlitval, startcodefile],
  P5StmtExprDefs: FROM "p5stmtexprdefs",
  SymDefs: FROM "symdefs" USING [BitAddress, bodytype, BTIndex, BTNull, ByteIndex, CBTIndex, ContextLev
**el, CSEIndex, CSENull, CTXIndex, ctxtype, HTIndex, HTNull, ISEIndex, ISENull, lG, lL, MDIndex, record
**CSEIndex, recordCSENull, SEIndex, SENull, setype, typeTYPE],
  SymTabDefs: FROM "symtabdefs" USING [FnField, NextSe, UnderType, WordsForType],
  TableDefs: FROM "tabledefs" USING [FreeChunk, GetChunk, TableBase, TableNotifier],
  TreeDefs: FROM "treedefs" USING [empty, freenode, scanlist, TreeIndex, TreeLink, treetype];

DEFINITIONS FROM CodeDefs;

Driver: PROGRAM
    IMPORTS MPtr: ComData, CPtr: Code, CodeDefs, ErrorDefs, LitDefs, OpTableDefs, P5ADefs, P5BDefs, Sym
**TabDefs, TableDefs, TreeDefs
    EXPORTS CodeDefs, P5ADefs, P5StmtExprDefs =
  BEGIN
  OPEN SymTabDefs, P5ADefs, P5BDefs;


  -- imported definitions

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

  codebaseOffset: CARDINAL = ControlDefs.codebaseOffset;
  framelink: CARDINAL = ControlDefs.framelink;
  returnOffset: CARDINAL = ControlDefs.returnOffset;
  localbase: CARDINAL = ControlDefs.localbase;
  globalbase: CARDINAL = ControlDefs.globalbase;

  BitAddress: TYPE = SymDefs.BitAddress;
  BTIndex: TYPE = SymDefs.BTIndex;
  CBTIndex: TYPE = SymDefs.CBTIndex;
  BTNull: BTIndex = SymDefs.BTNull;
  ContextLevel: TYPE = SymDefs.ContextLevel;
  CSEIndex: TYPE = SymDefs.CSEIndex;
  CSENull: CSEIndex = SymDefs.CSENull;
  CTXIndex: TYPE = SymDefs.CTXIndex;
  HTIndex: TYPE = SymDefs.HTIndex;
  HTNull: HTIndex = SymDefs.HTNull;
  ISEIndex: TYPE = SymDefs.ISEIndex;
  ISENull: ISEIndex = SymDefs.ISENull;
  lG: ContextLevel = SymDefs.lG;
  lL: ContextLevel = SymDefs.lL;
  MDIndex: TYPE = SymDefs.MDIndex;
  recordCSEIndex: TYPE = SymDefs.recordCSEIndex;
  recordCSENull: recordCSEIndex = SymDefs.recordCSENull;
```

```
  SEIndex: TYPE = SymDefs.SEIndex;
  SENull: SEIndex = SymDefs.SENull;
  typeTYPE: CSEIndex = SymDefs.typeTYPE;

  empty: TreeLink = TreeDefs.empty;
  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)
  bb: TableDefs.TableBase;              -- body entry base (local copy)
  cb: ChunkBase;                -- code base (local copy)

  DriverNotify: PUBLIC TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever table area is repacked
    seb ← base[SymDefs.setype];
    ctxb ← base[SymDefs.ctxtype];
    bb ← base[SymDefs.bodytype];
    tb ← base[TreeDefs.treetype];
    cb ← LOOPHOLE[tb];
    AddressNotify[base];
    ExpressionNotify[base];
    FlowExpressionNotify[base];
    FlowNotify[base];
    StackNotify[base];
    StatementNotify[base];
    StoreNotify[base];
    CallsNotify[base];
    OutCodeNotify[base];
    FinalNotify[base];
    JumpsNotify[base];
    PeepholeNotify[base];
    RETURN
    END;


  codestart: CCIndex;
  endofcurbody: LabelCCIndex;
  bodyinrecord, bodyoutrecord: recordCSEIndex;
  codeindex: SymDefs.ByteIndex;
  mlock: TreeLink;
  longlock: BOOLEAN;

  Cmodule: PUBLIC PROCEDURE =
    BEGIN -- main driver for code generation
    bti, prev: BTIndex;

    CPtr.acstack ← 0;
    bodyinrecord ← bodyoutrecord ← recordCSENull;
    CPtr.ZEROlexeme ← Lexeme[literal[word[LitDefs.FindLiteral[0]]]];
    AddressInit[];
    StackInit[];
    stackoff[];
    CPtr.xtracting ← FALSE;
    CPtr.firstcaseselread ← FALSE;
    codeindex ← CPtr.fileindex ← 0;
    CPtr.catchoutrecord ← recordCSENull;
    CPtr.catchcount ← 0;
    CPtr.actenable ← LabelCCNull;
    CPtr.codeptr← codestart ← CCNull;
    startcodefile[];
    bti ← MPtr.bodyRoot;
    DO
      WITH (bb+bti) SELECT FROM
        Callable => Cbody[LOOPHOLE[bti]];
        ENDCASE;
      IF (bb+bti).firstSon # BTNull
        THEN bti ← (bb+bti).firstSon
        ELSE
          DO
          prev ← bti;  bti ← (bb+bti).link.index;
          IF bti = BTNull THEN GO TO Done;
          IF (bb+prev).link.which # parent THEN EXIT;
          ENDLOOP;
```

```
    REPEAT
      Done => NULL;
    ENDLOOP;
  MPtr.objectBytes ← endcodefile[];
  StackFinal[];
  RETURN;
  END;


Cbody: PROCEDURE [bti: CBTIndex] =
  BEGIN -- produces code for body
  psei: CSEIndex ← UnderType[(bb+bti).ioType];
  bodynode: TreeIndex;
  retryentry: LabelCCIndex;
  lockaddrsize: CARDINAL;

 CPtr.mainBody ← bti = MPtr.mainBody;
 MPtr.bodyIndex ← bti;

 WITH bi: (bb+bti).info SELECT FROM
  Internal =>
  BEGIN
  MPtr.textIndex ← bi.sourceIndex;
  bodynode ← bi.bodyTree;
  CPtr.curctxlvl ← (bb+bti).level;

  -- set up input and output contexts
  WITH (seb+psei) SELECT FROM
    transfer =>
      BEGIN
      bodyinrecord ← LOOPHOLE[UnderType[inrecord], recordCSEIndex];
      IF bodyinrecord # recordCSENull THEN
        (ctxb+(seb+bodyinrecord).fieldctx).ctxlevel ← CPtr.curctxlvl;
      bodyoutrecord ← LOOPHOLE[UnderType[outrecord], recordCSEIndex];
      IF bodyoutrecord # recordCSENull THEN
        (ctxb+(seb+bodyoutrecord).fieldctx).ctxlevel ← CPtr.curctxlvl;
      END;
    ENDCASE;

  IF CPtr.mainBody THEN
    BEGIN
    MPtr.objectFrameSize ← bi.frameSize;
    bi.frameSize ← localbase;
    CPtr.curctxlvl ← 1L;
    END;
  CPtr.tempstart ← CPtr.framesz ← bi.frameSize;
  codeindex ← CPtr.fileindex ← bi.sourceIndex;

  -- init the code stream and put down bracketing labels

  CPtr.curbodyretlabel ← labelalloc[];
  endofcurbody ← labelalloc[];
  CPtr.codeptr ← CCNull;
  codestart ← createlabel[];

  -- init data for creating temporaries

  (ctxb+CPtr.tempcontext).ctxlevel ← CPtr.curctxlvl;

  -- tuck parameters away into the frame

  IF CPtr.acstack # 0 THEN SIGNAL CPtr.StackNotEmptyAtStatement;
  WITH (bb+bti) SELECT FROM
   Inner => BEGIN
            CPtr.acstack ← 1;
            Ciout1[FOpCodes.qLINKB, frameOffset-localbase];
            END;
   ENDCASE;
  stackon[];
  popinvals[bodyinrecord, FALSE];
  purgependtemplist[];

      -- do string literals

  IF CPtr.mainBody THEN
    MPtr.objectFrameSize ← ProcessGlobalStrings[MPtr.objectFrameSize];
```

```
    CPtr.tempstart ← ProcessLocalStrings[CPtr.tempstart, bi.stOrigin];
    bi.frameSize ← CPtr.framesz ← MAX [CPtr.framesz, CPtr.tempstart];

        -- do initialization code and main body

    IF CPtr.mainBody AND MPtr.stopping THEN
      BEGIN OPEN FOpCodes;
      Ciout1[qLADRB, 0];
      Ciout1[qSG, globalbase];
      END;

    IF (tb+bodynode).attr1 THEN
      BEGIN
      insertlabel[retryentry ← labelalloc[]];
      lockaddrsize ← loadtsonaddress[(mlock ← (tb+bodynode).son4)];
      longlock ← lockaddrsize > wordlength;
      Ciout0[IF longlock THEN FOpCodes.qMEL
        ELSE FOpCodes.qME];
      Ciout1[FOpCodes.qLI, 0];
      Coutjump[JumpE, retryentry];
      END
    ELSE mlock ← empty;

    (tb+bodynode).son2 ← Cstatement[(tb+bodynode).son2];
    (tb+bodynode).son3 ← Cstatement[(tb+bodynode).son3];
    (tb+bodynode).son1 ← TreeDefs.empty;
    insertlabel[endofcurbody];
    IF CPtr.acstack # 0 THEN SIGNAL CPtr.StackNotEmptyAtStatement;

    -- push the return values onto the stack

    IF mlock # empty THEN
      BEGIN [] ← loadtsonaddress[mlock];
      Ciout0[IF longlock THEN FOpCodes.qMXDL ELSE FOpCodes.qMXD];
      END;
    pushretvals[];
    clearstack[];
    CPtr.acstack ← 0;
    insertlabel[CPtr.curbodyretlabel];
    IF CPtr.mainBody AND MPtr.stopping THEN
        BEGIN Ciout1[FOpCodes.qLI, 0]; Ciout1[FOpCodes.qSG, globalbase]; END;
    stackoff[];
    Ciout0[FOpCodes.qRET];
    purgependtemplist[];

    -- write frame size into bodyitem

    bi.frameSize ← CPtr.framesz;
    WITH (bb+bti) SELECT FROM
      Inner => IF bi.frameSize > framevec[LENGTH[framevec]-1]
        THEN ErrorDefs.errorsei[addressOverflow, id];
      ENDCASE;

    -- fixup jumps

    IF MPtr.nErrors = 0 THEN Cfixup[codestart];

        -- output the object code

    codeindex ← CPtr.fileindex ← NULLfileindex;
    TreeDefs.freenode[bodynode];
    IF MPtr.nErrors = 0 THEN outbinary[bti, codestart]
    ELSE
      BEGIN
      c, next: CCIndex;
      FOR c ← codestart, next WHILE c # CCNull DO
        next ← cb[c].flink;
        deletecell[c];
        ENDLOOP;
      END;
    freetemplist[];
    END;
    ENDCASE;
    RETURN
    END;
```

```
popparams: PROCEDURE [sei: ISEIndex] =
  BEGIN -- recursive routine to store params from acstack into
        -- frame in lifo order
  IF sei = ISENull THEN RETURN;
  popparams[nextvar[NextSe[sei]]];
  sCassign[sei];
  RETURN
  END;


popinvals: PUBLIC PROCEDURE [irecord: recordCSEIndex, isenable: BOOLEAN] =
  BEGIN -- sets up input parameters if number of parms exceeds acstack
  l: bdo Lexeme;
  nparms: CARDINAL;
  r: BDOIndex;
  b: BitAddress;
  sei: ISEIndex;
  tlex: se Lexeme ← topostack;
  dup: BOOLEAN;
  nwds: CARDINAL;

  IF irecord = CSENull THEN RETURN;
  nparms ← wordsforsei[irecord];
  IF isenable THEN
    IF nparms <= 1 THEN RETURN
    ELSE Ciout1[FOpCodes.qLL,localbase+1];
  sei ← nextvar[(ctxb+(seb+irecord).fieldctx).selist];
  IF nparms > MaxParmsInStack OR (isenable AND nparms > 1) THEN
    BEGIN
    IF ~isenable THEN
      BEGIN CPtr.acstack ← 1; incrstack[1] END;
    l ← Lexeme[bdo[]];
    UNTIL sei = ISENull DO
      r ← l.lexbdoi ← makeTOSaddrBDOItem[wordlength];
      [b, cb[r].offset.size] ← FnField[sei];
      cb[r].offset.posn ← FullBitAddress[wd: b.wd, bd: b.bd];
      nwds ← cb[r].offset.size/wordlength;
      dup ←  ~isenable OR nextvar[NextSe[sei]] # ISENull;
      IF dup THEN
        IF nwds <= 2 THEN Ciout0[FOpCodes.qDUP]
        ELSE IF tlex = topostack THEN
          BEGIN
          tlex ← gentemplex[1];
          sCassign[tlex.lexsei];
          Ciout0[FOpCodes.qPUSH];
          END;
      slCassign[sei, l, FALSE, nwds];
      sei ← nextvar[NextSe[sei]];
      IF dup AND nwds > 2 THEN pushlex[tlex];
      ENDLOOP;
    IF ~isenable THEN Ciout0[FOpCodes.qFREE];
    END
  ELSE
    BEGIN CPtr.acstack ← nparms; incrstack[nparms]; popparams[sei]; END;
  CPtr.acstack ← 0;
  RETURN
  END;


pushretvals: PROCEDURE =
  BEGIN -- pushes the return vals from a body onto the stack
  sei: ISEIndex;
  l: se Lexeme;
  r, rr: BDOIndex;
  b: BitAddress;
  nretvals: CARDINAL;

  IF bodyoutrecord = CSENull THEN RETURN;
  nretvals ← wordsforsei[bodyoutrecord];
  sei ← (ctxb+(seb+bodyoutrecord).fieldctx).selist;
  IF (seb+nextvar[sei]).htptr = HTNull THEN -- anonymous RETURNS list
    BEGIN
    Csyserror[];
```

```
    RETURN
    END;
  IF nretvals > MaxParmsInStack THEN
    BEGIN
    pushlitval[computeframesize[nretvals]];
    Ciout0[FOpCodes.qALLOC];
    r ← makeTOSaddrBDOItem[wordlength];
    UNTIL (sei ← nextvar[sei]) = ISENull DO
      rr ← copyBDOItem[r];
      [b, cb[rr].offset.size] ← FnField[sei];
      cb[rr].offset.posn ← FullBitAddress[wd: b.wd, bd: b.bd];
      CRassign[rr, empty, sei, TRUE];
      sei ← NextSe[sei];
      ENDLOOP;
    chkrandsonstack[1];
    releaseBDOItem[r];
    RETURN
    END;
  l ← Lexeme[se[]];
  UNTIL (sei ← nextvar[sei]) = ISENull DO
    l.lexsei ← sei; pushlex[l]; sei ← NextSe[sei]; ENDLOOP;
  RETURN
  END;


wordsforsei: PUBLIC PROCEDURE [sei: SEIndex] RETURNS [CARDINAL] =
  BEGIN
  RETURN [IF sei = SENull THEN 0 ELSE WordsForType[UnderType[sei]]];
  END;


wordsforoperand: PUBLIC PROCEDURE [t: TreeLink] RETURNS [n: CARDINAL] =
  BEGIN -- compute number of words for storing value of tree
  WITH t SELECT FROM
    literal => n ← 1;
    symbol => n ← wordsforsei[(seb+index).idtype];
    subtree => n ← WordsForType[operandtype[t]];
    ENDCASE;
  RETURN
  END;


bitsfortype: PUBLIC PROCEDURE [sei: SEIndex] RETURNS [CARDINAL] =
  BEGIN
  csei: CSEIndex ← UnderType[sei];

  WITH (seb+csei) SELECT FROM
    record => RETURN[length];
    ENDCASE => RETURN[WordsForType[sei]*wordlength]
  END;


bitsforoperand: PUBLIC PROCEDURE [t: TreeLink] RETURNS [CARDINAL] =
  BEGIN
  RETURN[bitsfortype[operandtype[t]]]
  END;


operandtype: PUBLIC PROCEDURE [t: TreeLink] RETURNS [sei: CSEIndex] =
  BEGIN -- compute number of words for storing value of tree
  WITH e:t SELECT FROM
    literal =>
      WITH e.info SELECT FROM
        string => sei ← MPtr.typeSTRING;
        ENDCASE => SIGNAL CPtr.CodePassInconsistancy;
    symbol => sei ← UnderType[(seb+e.index).idtype];
    subtree =>
      IF e = empty THEN
        IF CPtr.xtracting THEN
          sei ← UnderType[(seb+CPtr.xtractsei).idtype]
        ELSE ERROR
      ELSE sei ← (tb+e.index).info;
    ENDCASE;
  RETURN
  END;
```

```
ReleaseLock: PUBLIC PROCEDURE =
  BEGIN
  RequireStack[0];
  [] ← loadtsonaddress[mlock];
  Ciout0[IF longlock THEN FOpCodes.qMXDL ELSE FOpCodes.qMXD];
  RETURN
  END;


sCreturn: PROCEDURE [node: TreeIndex, isresume: BOOLEAN] =
  BEGIN -- generate code for RETURN and RESUME
  savacstack: CARDINAL ← CPtr.acstack;
  nretvals: CARDINAL;
  rsei: CSEIndex;
  monitored: BOOLEAN;

  monitored ← ~isresume AND (tb+node).attr1;
  IF isresume OR ~commonret[(tb+node).son1] THEN
    BEGIN
    IF monitored AND (tb+node).attr2 THEN
      BEGIN ReleaseLock[]; monitored ← FALSE; END;
    rsei ← IF isresume THEN CPtr.catchoutrecord ELSE bodyoutrecord;
    nretvals ← IF rsei = CSENull THEN 0 ELSE WordsForType[UnderType[rsei]];
    IF nretvals > MaxParmsInStack OR (isresume AND nretvals > 1) THEN
      BEGIN
      pushlitval[computeframesize[nretvals]];
      Ciout0[FOpCodes.qALLOC];
      transferconstruct[makeTOSaddrBDOItem[wordlength], (tb+node).son1, rsei];
      nretvals ← 1;
      END
    ELSE transferconstruct[BDONull, (tb+node).son1, rsei];
    IF monitored THEN ReleaseLock[];
    chkrandsonstack[nretvals];
    IF isresume THEN
      BEGIN pushlitval[1]; adjustacstack[-1]; pop[]; Ciout0[FOpCodes.qRET];
      Coutjump[JumpRet, LabelCCNull];
      END
    ELSE Coutjump[Jump, CPtr.curbodyretlabel];
    CPtr.acstack ← savacstack;
    END
  ELSE Coutjump[Jump, endofcurbody];
  RETURN
  END;


Creturn: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN -- produce code for RETURN
  sCreturn[node, FALSE !LogHeapFree => RESUME[FALSE, topostack]]; RETURN
  END;


Cresume: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN -- produce code for RESUME
  sCreturn[node, TRUE !LogHeapFree => RESUME[FALSE, topostack]]; RETURN
  END;


commonret: PROCEDURE [t: TreeLink] RETURNS [common: BOOLEAN] =
  BEGIN -- test if the returns list duplicats the returns declaration
  sei: ISEIndex;
  scr: PROCEDURE [t: TreeLink] =
    BEGIN
    IF ~common THEN RETURN;
    WITH t SELECT FROM
      literal => common ← FALSE;
      symbol => common ← sei = index;
      subtree => common ← FALSE;
      ENDCASE;
    IF sei # SENull THEN sei ← nextvar[NextSe[sei]];
    RETURN
    END;

  common ← TRUE;
  IF t = empty THEN RETURN;
  IF bodyoutrecord # CSENull THEN
```

```
    sei ← nextvar[(ctxb+(seb+bodyoutrecord).fieldctx).selist]
  ELSE RETURN [FALSE];
  TreeDefs.scanlist[t, scr];
  RETURN
  END;


nextvar: PUBLIC PROCEDURE [sei: ISEIndex] RETURNS [ISEIndex] =
  BEGIN -- starting at sei returns first variable on ctx-list
  IF sei = ISENull THEN RETURN [ISENull];
  DO
    IF (seb+sei).idtype # typeTYPE THEN RETURN [sei];
    IF (sei ← NextSe[sei]) = ISENull THEN EXIT;
    ENDLOOP;
  RETURN [ISENull];
  END;


prevvar: PUBLIC PROCEDURE [ssei, sei : ISEIndex] RETURNS [ISEIndex] =
  BEGIN -- returns vars in reverse order as those returned by  nextvar
  psei: ISEIndex ← nextvar[ssei];
  rsei: ISEIndex;

  IF psei = sei THEN RETURN [psei];
  UNTIL psei = sei DO
    rsei ← psei; psei ← nextvar[NextSe[psei]]; ENDLOOP;
  RETURN [rsei];
  END;


Ciout0: PUBLIC PROCEDURE [i: BYTE] =
  BEGIN -- outputs an parameter-less instruction
  c: CodeCCIndex;
  pusheffect: INTEGER = PushEffect[i];

  chkacstack[i];
  IF NumberOfParams[i] # 0 THEN P5ADefs.P5Error[257];
  codeindex ← MAX[CPtr.fileindex, codeindex];
  c ← AllocCodeCCItem[0];
  cb[c].inst ← i;
  cb[c].minimalStack ← CPtr.acstack = pusheffect;
  RETURN
  END;


Ciout1: PUBLIC PROCEDURE [i: BYTE, p1: WORD] =
  BEGIN -- outputs an one-parameter instruction
  c: CodeCCIndex;
  pusheffect: INTEGER = PushEffect[i];

  chkacstack[i];
  IF NumberOfParams[i] # 1 THEN P5ADefs.P5Error[258];
  codeindex ← MAX[CPtr.fileindex, codeindex];
  c ← AllocCodeCCItem[1];
  cb[c].inst ← i;
  cb[c].parameters[1] ← p1;
  cb[c].minimalStack ← CPtr.acstack = pusheffect;
  RETURN
  END;


Ciout2: PUBLIC PROCEDURE [i: BYTE, p1, p2: WORD] =
  BEGIN -- outputs an two-parameter instruction
  c: CodeCCIndex;
  pusheffect: INTEGER = PushEffect[i];

  chkacstack[i];
  IF NumberOfParams[i] # 2 THEN P5ADefs.P5Error[259];
  codeindex ← MAX[CPtr.fileindex, codeindex];
  c ← AllocCodeCCItem[2];
  cb[c].inst ← i;
  cb[c].parameters[1] ← p1;
  cb[c].parameters[2] ← p2;
  cb[c].minimalStack ← CPtr.acstack = pusheffect;
  RETURN
  END;
```

```
  Ciout3: PUBLIC PROCEDURE [i: BYTE, p1, p2, p3: WORD] =
    BEGIN -- outputs an three-parameter instruction
    c: CodeCCIndex;
    pusheffect: INTEGER = PushEffect[i];

    chkacstack[i];
    IF NumberOfParams[i] # 3 THEN P5ADefs.P5Error[260];
    codeindex ← MAX[CPtr.fileindex, codeindex];
    c ← AllocCodeCCItem[3];
    cb[c].inst ← i;
    cb[c].parameters[1] ← p1;
    cb[c].parameters[2] ← p2;
    cb[c].parameters[3] ← p3;
    cb[c].minimalStack ← CPtr.acstack = pusheffect;
    RETURN
    END;


  treeliteral: PUBLIC PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
    BEGIN
    node: TreeIndex;
      DO
      WITH t SELECT FROM
        literal => RETURN[info.litTag = word];
        subtree =>
          BEGIN  node ← index;
          SELECT (tb+node).name FROM
            cast, mwconst =>  t ← (tb+node).son1;
            ENDCASE => RETURN [FALSE];
          END;
        ENDCASE => RETURN[FALSE]
      ENDLOOP
    END;

  treeliteralvalue: PUBLIC PROCEDURE [t: TreeLink] RETURNS [WORD] =
    BEGIN
    node: TreeIndex;
      DO
      WITH e:t SELECT FROM
        literal =>
          WITH e.info SELECT FROM
            word => RETURN [LitDefs.LiteralValue[index]];
            ENDCASE => EXIT;
        subtree =>
          BEGIN  node ← e.index;
          SELECT (tb+node).name FROM
            cast, mwconst =>  t ← (tb+node).son1;
            ENDCASE => EXIT;
          END;
        ENDCASE => EXIT
      ENDLOOP;
    P5ADefs.P5Error[261];
    END;

  maketreeliteral: PUBLIC PROCEDURE [val: WORD] RETURNS [TreeLink] =
    BEGIN
    RETURN [TreeLink[literal[[word[index: LitDefs.FindLiteral[val]]]]]]
    END;

  labelalloc: PUBLIC PROCEDURE RETURNS [c: LabelCCIndex] =
    BEGIN -- gets a chunk for a label but does not insert it in stream
    c ← GetChunk[SIZE[label CCItem]];
    cb[c] ←
        CCItem[free: FALSE, pad:0, flink: , blink: , ccvalue: label[labelseen: FALSE, jumplist: JumpCCN
**ull]];
    RETURN
    END;


  createlabel: PUBLIC PROCEDURE RETURNS [c: LabelCCIndex] =
    BEGIN -- allocates and inserts a label at codeptr
    c ← labelalloc[];
    insertlabel[c];
    RETURN
```

```
    END;


  ccellalloc: PUBLIC PROCEDURE [t: CodeChunkType] =
    BEGIN -- allocates a cell for code or jump
    c: CCIndex;
    nwords: CARDINAL;

    codeindex ← MAX[CPtr.fileindex, codeindex];
    SELECT t FROM
      code => P5ADefs.P5Error[262];
      label => P5ADefs.P5Error[263];
      jump => nwords ← SIZE[jump CCItem];
      other => nwords ← SIZE[other CCItem];
      ENDCASE;
    c ← GetChunk[nwords];
    SELECT t FROM
      jump =>
        cb[c] ←
          CCItem[free: FALSE, pad:0, flink: , blink: , ccvalue: jump[,,,,,,,,]];
      other =>
        cb[c] ←
          CCItem[free: FALSE, pad:0, flink: , blink: , ccvalue: other[]];
      ENDCASE;
    linkCCItem[c];
    RETURN
    END;

  ParamCount: PUBLIC PROCEDURE [c: CodeCCIndex] RETURNS [CARDINAL] =
    BEGIN
    RETURN[IF cb[c].isize # 0 THEN cb[c].isize-1
        ELSE IF cb[c].realinst THEN OpTableDefs.instlength[cb[c].inst]-1
           ELSE NumberOfParams[cb[c].inst]]
    END;

  AllocCodeCCItem: PUBLIC PROCEDURE [n: [0..3]] RETURNS [c: CodeCCIndex] =
    BEGIN
    c ← GetChunk[SIZE[code CCItem] + n];
    cb[c] ←
          CCItem[free: FALSE, pad:0, flink: CCNull, blink: CCNull, ccvalue:
                code[inst: 0, realinst: FALSE, minimalStack: FALSE,
                sourcefileindex: NULLfileindex,
                isize: 0, aligned: FALSE, fill: 0, parameters: ]];
    IF CPtr.stking THEN cb[c].sourcefileindex ← codeindex;
    linkCCItem[c];
    RETURN
    END;

  linkCCItem: PROCEDURE[c: CCIndex] =
    BEGIN -- inserts a CCItem in list @ codeptr
    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;

  RequireStack: PUBLIC PROCEDURE [n: INTEGER] =
    BEGIN
    IF CPtr.acstack # n THEN
      BEGIN
      dumpstack[];
      IF n # 0 THEN putrandsonstack[n];
      END;
    RETURN
    END;

  Coutjump: PUBLIC PROCEDURE [jt: JumpType, l: LabelCCIndex] =
    BEGIN -- outputs a jump-type code ceel into the code stream
    SELECT jt FROM
```

```
        Jump, JumpA, JumpC, JumpCA, JumpRet => chkacstack[FOpCodes.qJ];
        ENDCASE => chkacstack[FOpCodes.qJREL];
    ccellalloc[jump];
    WITH cb[CPtr.codeptr] SELECT FROM
      jump =>
        BEGIN
        fixedup ← FALSE;
        completed ← FALSE;
        jtype ← jt;
        destlabel ← l;
        IF l # LabelCCNull THEN
          BEGIN
          thread ← cb[l].jumplist;
          cb[l].jumplist ← LOOPHOLE[CPtr.codeptr, JumpCCIndex];
          END
        ELSE thread ← JumpCCNull;
        RETURN
        END;
      ENDCASE
    END;


  deletecell: PUBLIC PROCEDURE [c: CCIndex] =
    BEGIN -- deletes cell from code stream
    nwords: CARDINAL;

    IF cb[c].blink # CCNull THEN
      cb[cb[c].blink].flink ← cb[c].flink;
    IF cb[c].flink # CCNull THEN
      cb[cb[c].flink].blink ← cb[c].blink;
    WITH cb[c] SELECT FROM
      code => nwords ← ParamCount[LOOPHOLE[c]] + SIZE[code CCItem];
      label => nwords ← SIZE[label CCItem];
      jump => nwords ← SIZE[jump CCItem];
      other => nwords ← SIZE[other CCItem];
      ENDCASE;
    FreeChunk[c, nwords];
    RETURN
    END;

  FreeChunk: PUBLIC PROCEDURE [i: CodeDefs.ChunkIndex, size: CARDINAL] =
    BEGIN
    TableDefs.FreeChunk[LOOPHOLE[i],size];
    END;

  GetChunk: PUBLIC PROCEDURE [size: CARDINAL] RETURNS [CodeDefs.ChunkIndex] =
    BEGIN
    RETURN [LOOPHOLE[TableDefs.GetChunk[size]]];
    END;


  framevec: ARRAY [0..ControlDefs.MaxAllocSlot) OF CARDINAL = [
    7,11,15,19,23,27,31,39,47,55,67,79,95,111,127,147,171,199,231];


  computeframesize: PUBLIC PROCEDURE [fs: CARDINAL] RETURNS [CARDINAL] =
    BEGIN -- finds alloc-vector index for frame of size fs
    fx: CARDINAL;

    FOR fx IN [0..ControlDefs.MaxAllocSlot) DO
      IF fs <= framevec[fx] THEN RETURN [fx] ENDLOOP;
    RETURN [fs];
    END;


END...
```