```
-- file Pass4D.Mesa
-- last modified by Satterthwaite, August 29, 1978  10:02 AM

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

Pass4D: PROGRAM
    IMPORTS
        ErrorDefs, LitDefs, P4Defs, SymSegDefs, SymTabDefs, TreeDefs,
        dataPtr: ComData
    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;    -- context table base address (local copy)
  bb: TableDefs.TableBase;      -- body table base address (local copy)

  DeclNotify: 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;


  DeclItem: PUBLIC PROCEDURE [item: TreeLink] =
    BEGIN
    node: TreeIndex = GetNode[item];
    type: CSEIndex;
    expNode: TreeIndex;
    initFlag, eqFlag: BOOLEAN;

    ExpInit: PROCEDURE =
      BEGIN  OPEN (tb+node);
      val, info: UNSPECIFIED;
      t: TreeLink;
      son3 ← P4Defs.RValue[son3, BiasForType[type],
                P4Defs.TargetRep[RepForType[type]]];
      IF ~AssignableRanges[type, P4Defs.OperandType[son3]]
        THEN  son3 ← P4Defs.ResolveSizes[son3, type];
      IF eqFlag
        THEN
          BEGIN
          t ← son3;
          WHILE testtree[t, cast]
            DO
            WITH t SELECT FROM
              subtree =>  t ← (tb+index).son1;
              ENDCASE;
            ENDLOOP;
          IF P4Defs.TreeLiteral[t]
            THEN
              BEGIN
              val ← P4Defs.TreeLiteralValue[t]; info ← BTNull; GO TO define
              END;
          IF testtree[t, mwconst]
            THEN
              BEGIN
              WITH t SELECT FROM
                subtree =>  (tb+index).info ← type;
                ENDCASE;
              AugmentSEValue[son1, t, FALSE];  son3 ← empty;
              val ← 0;  info ← BTNull;  GO TO define
              END;
```

```
        IF (seb+type).typetag = transfer
          THEN
            WITH t SELECT FROM
              symbol =>
                BEGIN
                sei: ISEIndex = index;
                IF (seb+sei).constant
                  THEN
                    BEGIN
                    IF (seb+sei).extended
                      THEN  AugmentSEValue[son1,
                              SymSegDefs.FindExtension[sei], TRUE];
                    val ← (seb+sei).idvalue;  info ← (seb+sei).idinfo;
                    GO TO define
                    END;
                END;
              ENDCASE;
        EXITS
          define =>
            BEGIN
            DefineSEValue[son1, val, info];
            son3 ← freetree[son3];  initFlag ← FALSE;
            END;
        END;
    SELECT (seb+NormalType[type]).typetag FROM
      pointer, arraydesc, relative =>
        IF listlength[son1] # 1 AND son3 # empty
         AND ~P4Defs.TreeLiteral[son3] AND ~testtree[son3, mwconst]
          THEN ErrorDefs.Warning[pointerInit];
      ENDCASE;
    P4Defs.VPop[];  RETURN
    END;

  saveIndex: CARDINAL = dataPtr.textIndex;
  IF (tb+node).mark = P4Defs.P4Mark THEN RETURN;      -- already processed
  (tb+node).mark ← P4Defs.P4Mark;
  dataPtr.textIndex ← (tb+node).info;
  initFlag ← (tb+node).son3 # empty;
  IF testtree[(tb+node).son2, modeTC]
    THEN  TypeExp[typeExp:(tb+node).son3, body:FALSE]
    ELSE
      BEGIN  OPEN (tb+node);
      IF ~initFlag
        THEN
          BEGIN
          TypeExp[typeExp:son2, body:FALSE];
          type ← UnderType[TypeForTree[son2]];
          WITH (seb+type) SELECT FROM
            record =>
              IF FrameVars[son1] AND
               (type = dataPtr.typeLOCK OR type = dataPtr.typeCONDITION)
                THEN  son3 ← ProcessInit[type];
            transfer =>
              IF mode = port
                THEN
                  BEGIN
                  pushtree[portinit, 0]; setinfo[type]; son3 ← mlpop[];
                  END;
            ENDCASE;
          END
        ELSE
          BEGIN  eqFlag ← attr1;
          TypeExp[son2, testtree[son3, body]];
          type ← UnderType[TypeForTree[son2]];
          WITH son3 SELECT FROM
            symbol, literal =>  ExpInit[];
            subtree =>
              BEGIN  expNode ← index;
              SELECT (tb+expNode).name FROM
                body =>
                  BEGIN
                  bti: CBTIndex = (tb+expNode).info;
                  IF eqFlag
                    THEN
                      BEGIN
                      DefineSEValue[
```

```
                                son1,
                                P4Defs.MakeEPLink[(bb+bti).entryIndex, 0],
                                bti];
                        son3 ← empty;
                        END
                      ELSE
                        BEGIN
                        pushtree[body, 0];  setinfo[bti];  son3 ← mlpop[];
                        END;
                    END;
                  signalinit =>
                    IF eqFlag
                      THEN
                        BEGIN
                        DefineSEValue[
                                son1,
                                P4Defs.MakeEPLink[(tb+expNode).info, 0],
                                BTNull];
                        son3 ← freetree[son3];
                        END;
                  stringinit =>
                    BEGIN  OPEN exp: (tb+expNode);
                    IF listlength[son1] # 1
                      THEN ErrorDefs.Warning[pointerInit];
                    exp.son2 ← P4Defs.RValue[exp.son2, 0, P4Defs.unsigned];
--                  IF P4Defs.TreeLiteralValue[exp.son2] < 0
--                    THEN ErrorDefs.error[stringLength];
                    P4Defs.VPop[];
                    END;
                  inline =>
                    BEGIN
                    (tb+expNode).son1 ←
                        updatelist[(tb+expNode).son1, InlineOp];
                    DefineSEValue[son1, 0, BTNull];
                    AugmentSEValue[son1, son3, FALSE];
                    son3 ← empty;  initFlag ← FALSE;
                    END;
                  ENDCASE =>  ExpInit[];
                END;
              ENDCASE;
            END;
        END;
    MarkAndCheckSE[(tb+node).son1, initFlag];
    dataPtr.textIndex ← saveIndex;  RETURN
    END;


  FrameVars: PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
    BEGIN
    s: TreeLink = listhead[t];
    RETURN [WITH s SELECT FROM
      symbol => (ctxb+(seb+index).ctxnum).ctxlevel # 1Z,
      ENDCASE => FALSE]
    END;

  ProcessInit: PROCEDURE [type: CSEIndex] RETURNS [TreeLink] =
    BEGIN
    condInit: ARRAY [0..2) OF WORD ← [0, 100];
    SELECT type FROM
      dataPtr.typeLOCK =>
        BEGIN
        pushlittree[LitDefs.FindLiteral[100000B]];  pushtree[cast, 1];
        END;
      dataPtr.typeCONDITION =>
        BEGIN
        pushlittree[LitDefs.FindLitDescriptor[DESCRIPTOR[condInit]]];
        pushtree[mwconst, 1];
        END;
      ENDCASE => ERROR;
    setinfo[type];  RETURN [mlpop[]]
    END;

  InlineOp: TreeMap =
    BEGIN
    RETURN [updatelist[t, P4Defs.NeutralExp]]
    END;
```

```
MarkAndCheckSE: PROCEDURE [t: TreeLink, initialized: BOOLEAN] =
  BEGIN

  UpdateSE: TreeScan =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        (seb+sei).mark4 ← TRUE;
        IF dataPtr.definitionsOnly THEN CheckDefinition[sei, initialized];
        END;
      ENDCASE =>  ERROR;
    RETURN
    END;

  scanlist[t, UpdateSE];  RETURN
  END;

CheckDefinition: PROCEDURE [sei: ISEIndex, initialized: BOOLEAN] =
  BEGIN
  SELECT (seb+sei).ctxnum FROM
    dataPtr.mainCtx =>
      SELECT XferMode[(seb+sei).idtype] FROM
        procedure, signal, error, program =>  IF ~initialized THEN RETURN;
        ENDCASE =>  IF (seb+sei).constant THEN RETURN;
    ENDCASE =>  IF ~initialized OR (seb+sei).constant THEN RETURN;
  ErrorDefs.errorsei[nonDefinition, sei];  RETURN
  END;


DeclUpdate: PUBLIC PROCEDURE [item: TreeLink] RETURNS [update: TreeLink] =
  BEGIN
  node: TreeIndex = GetNode[item];
  IF testtree[(tb+node).son2, modeTC] OR (tb+node).son3 = empty
    THEN update ← empty
    ELSE
      BEGIN  OPEN (tb+node);
      P4Defs.PushAssignment[son1, son3, UnderType[TypeForTree[son2]]];
      setinfo[info];  update ← mlpop[];  son3 ← empty;
      END;
  freenode[node];
  RETURN
  END;


TypeExp: PUBLIC PROCEDURE [typeExp: TreeLink, body: BOOLEAN] =
  BEGIN  -- body => arg records subsumed by frame
  node, subNode: TreeIndex;
  iSei: ISEIndex;
  sei, tSei: CSEIndex;
  rSei: recordCSEIndex;
  origin, newOrigin: CARDINAL;
  WordLength: CARDINAL = AltoDefs.wordlength;
  ByteLength: CARDINAL = AltoDefs.charlength;
  WITH typeExp SELECT FROM
    symbol =>
      BEGIN  iSei ← index;
      IF ~(seb+iSei).mark4
        THEN DeclItem[TreeLink[subtree[index: (seb+iSei).idvalue]]];
      END;
    subtree =>
      BEGIN  node ← index;
      SELECT (tb+node).name FROM
        discrimTC =>  TypeExp[typeExp:(tb+node).son1, body:FALSE];
        cdot =>  TypeExp[typeExp:(tb+node).son2, body:FALSE];
        frameTC =>  NULL;
        ENDCASE =>
          BEGIN  OPEN (tb+node);
          sei ← info;
          WITH type: (seb+sei) SELECT FROM
            enumerated =>  NULL;
            record =>
              BEGIN
```

```
    scanlist[son1, DeclItem];
    WITH type SELECT FROM
      notlinked =>
        P4Defs.LayoutFields[LOOPHOLE[sei, recordCSEIndex], 0];
      ENDCASE;
    ExtractFieldAttributes[LOOPHOLE[sei, recordCSEIndex]];
    END;
  pointer =>
    IF TypeConstructor[son1]
      THEN TypeExp[typeExp:son1, body:FALSE];
  array =>
    BEGIN
    IF son1 # empty THEN TypeExp[typeExp:son1, body:FALSE];
    TypeExp[typeExp:son2, body:FALSE];
    type.comparable ← ComparableType[UnderType[type.componenttype]];
    END;
  arraydesc =>
    IF TypeConstructor[son1]
      THEN TypeExp[typeExp:son1, body:FALSE];
  transfer =>
    BEGIN
    origin ← SELECT type.mode FROM
      program => ControlDefs.globalbase,
      signal, error => ControlDefs.localbase+1,
      procedure => ControlDefs.localbase+1,
      ENDCASE => 0;
    scanlist[son1, DeclItem];
    rSei ← type.inrecord;
    IF rSei # SENull
      THEN
        BEGIN
        newOrigin ← P4Defs.LayoutArgs[rSei, origin, body];
        (seb+rSei).length ← (newOrigin - origin)*WordLength;
        (seb+rSei).mark4 ← TRUE;
        origin ← newOrigin;
        END;
    scanlist[son2, DeclItem];
    rSei ← type.outrecord;
    IF rSei # SENull
      THEN
        BEGIN
        (seb+rSei).length ←
         (P4Defs.LayoutArgs[rSei, origin, body] - origin)*WordLength;
        (seb+rSei).mark4 ← TRUE;
        END;
    END;
  definition =>  NULL;
  union =>
    BEGIN
    DeclItem[son1];
    IF BiasForType[UnderType[(seb+type.tagsei).idtype]] # 0
      THEN ErrorDefs.errorsei[nonTagType, type.tagsei];
    scanlist[son2, DeclItem];
    END;
  relative =>
    BEGIN
    IF TypeConstructor[son1]
      THEN TypeExp[typeExp:son1, body:FALSE];
    IF TypeConstructor[son2]
      THEN TypeExp[typeExp:son2, body:FALSE];
    END;
  subrange =>
    BEGIN
    TypeExp[typeExp:son1, body:FALSE];
    subNode ← GetNode[son2];
    IF ~P4Defs.Interval[subNode, 0, P4Defs.both] THEN ERROR;
    IF P4Defs.VRep[] = P4Defs.none
      THEN ErrorDefs.errortree[mixedRepresentation, son2];
    [type.origin, type.range] ← P4Defs.ConstantInterval[subNode
      ! P4Defs.EmptyInterval =>
          BEGIN  type.empty ← TRUE; RESUME  END];
    P4Defs.VPop[];
    type.filled ← TRUE;
    tSei ← UnderType[type.rangetype];
    WITH cover: (seb+tSei) SELECT FROM
      subrange =>   -- incomplete test
```

```
                IF  type.origin < cover.origin
                 OR (~type.empty AND type.range > cover.range)
                  THEN ErrorDefs.error[subrangeNesting];
              ENDCASE => NULL;
            son2 ← freetree[son2];
            END;
          long => TypeExp[typeExp:son1, body:FALSE];
          ENDCASE => ERROR;
        (seb+sei).mark4 ← TRUE;
        END;
    END;
  ENDCASE => ERROR;
RETURN
END;

TypeConstructor: PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
  BEGIN
  RETURN [WITH t SELECT FROM
    subtree =>
      SELECT (tb+index).name FROM
        dot, cdot, discrimTC => FALSE,
        ENDCASE => TRUE,
    ENDCASE => FALSE]
  END;

ExtractFieldAttributes: PROCEDURE [rType: recordCSEIndex] =
  BEGIN
  -- compatibility version
  sei: ISEIndex;
  type: CSEIndex;
  comparable, privateFields: BOOLEAN;
  comparable ← TRUE;   privateFields ← FALSE;
  FOR sei ← (ctxb+(seb+rType).fieldctx).selist, NextSe[sei] UNTIL sei = SENull
    DO
    IF ~(seb+sei).public THEN privateFields ← TRUE;
    type ← UnderType[(seb+sei).idtype];
    WITH t: (seb+type) SELECT FROM
      record =>
        IF ~t.comparable AND ~ComparableType[type] THEN comparable ← FALSE;
      array =>
        IF ~t.comparable AND ~ComparableType[type] THEN comparable ← FALSE;
      union =>
        IF ~t.equalLengths THEN comparable ← FALSE;
      ENDCASE;
    ENDLOOP;
  (seb+rType).comparable ← comparable;
  (seb+rType).privateFields ← privateFields;
  RETURN
  END;

TypeForTree: PUBLIC PROCEDURE [t: TreeLink] RETURNS [SEIndex] =
  BEGIN
  RETURN [WITH t SELECT FROM
    symbol => index,
    subtree => (tb+index).info,
    ENDCASE => typeANY]
  END;


DefineSEValue: PROCEDURE [t: TreeLink, value, info: UNSPECIFIED] =
  BEGIN

  UpdateSE: TreeScan =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        (seb+sei).constant ← TRUE;
        (seb+sei).idvalue ← value;  (seb+sei).idinfo ← info;
        END;
      ENDCASE => ERROR;
    RETURN
    END;

  scanlist[t, UpdateSE];  RETURN
```

```
    END;

  AugmentSEValue: PROCEDURE [t, extension: TreeLink, copy: BOOLEAN] =
    BEGIN

    UpdateSE: TreeScan =
      BEGIN
      WITH t SELECT FROM
        symbol =>
          SymSegDefs.EnterExtension[index,
                IF copy THEN IdentityMap[extension] ELSE extension];
        ENDCASE =>  ERROR;
      copy ← TRUE;  RETURN
      END;

    scanlist[t, UpdateSE];  RETURN
    END;


  BiasForType: PUBLIC PROCEDURE [type: CSEIndex] RETURNS [INTEGER] =
    BEGIN
    ctx: CTXIndex;
    IF type = SENull THEN RETURN [0];
    DO
      WITH (seb+type) SELECT FROM
        subrange =>  RETURN [origin];
        record =>
          BEGIN  ctx ← fieldctx;
          IF ~unifield OR CtxEntries[ctx] # 1 THEN RETURN [0];
          type ← UnderType[(seb+(ctxb+ctx).selist).idtype];
          END;
        ENDCASE =>  RETURN [0]
      ENDLOOP;
    END;

  RepForType: PUBLIC PROCEDURE [type: CSEIndex] RETURNS [P4Defs.Repr] =
    BEGIN
    ctx: CTXIndex;
    IF type = SENull THEN RETURN [P4Defs.none];
    DO
      WITH (seb+type) SELECT FROM
        basic =>
          RETURN [SELECT code FROM
            codeANY => P4Defs.both + P4Defs.other,
            codeINTEGER => P4Defs.signed,
            codeCHARACTER => P4Defs.both,
            ENDCASE => P4Defs.other];
        enumerated => RETURN [P4Defs.both];
        pointer => RETURN [P4Defs.unsigned];
        record =>
          BEGIN  ctx ← fieldctx;
          IF ~unifield OR CtxEntries[ctx] # 1 THEN RETURN [P4Defs.other];
          type ← UnderType[(seb+(ctxb+ctx).selist).idtype];
          END;
        relative =>  type ← UnderType[offsetType];
        subrange =>
          RETURN [IF origin >= 0
            THEN
              (IF CARDINAL[origin] + range > 77777B
                THEN P4Defs.unsigned ELSE P4Defs.both)
            ELSE
              (IF range <= 77777B THEN P4Defs.signed ELSE P4Defs.none)];
        ENDCASE =>  RETURN [P4Defs.other]
      ENDLOOP;
    END;


  WordsForType: PUBLIC PROCEDURE [type: CSEIndex] RETURNS [nW: CARDINAL] =
    BEGIN
    WordLength: CARDINAL = AltoDefs.wordlength;
    IF ~(seb+type).mark4
      THEN
        nW ← ((P4Defs.BitsForType[type]+(WordLength-1))/WordLength)*WordLength
      ELSE
        BEGIN
```

```
      WITH (seb+type) SELECT FROM
        record => lengthUsed ← TRUE;
        array => lengthUsed ← TRUE;
        ENDCASE => NULL;
      nW ← SymTabDefs.WordsForType[type];
      END;
  RETURN
  END;

AssignableRanges: PUBLIC PROCEDURE [lType, rType: CSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  nw: CARDINAL;
  WITH l: (seb+lType) SELECT FROM
    record =>
      BEGIN  l.lengthUsed ← TRUE;
      WITH r: (seb+rType) SELECT FROM
        record =>
          BEGIN  r.lengthUsed ← TRUE;  RETURN[l.length = r.length]  END;
        ENDCASE;
      END;
    ENDCASE;
  nw ← P4Defs.WordsForType[lType];
  RETURN [(nw # 0) AND (P4Defs.WordsForType[rType] = nw)]
  END;

ComparableRanges: PUBLIC PROCEDURE [type1, type2: CSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  -- compatibility version
  nw: CARDINAL = P4Defs.WordsForType[type1];
  IF nw = 0 OR P4Defs.WordsForType[type2] # nw THEN RETURN [FALSE];
  RETURN [ComparableType[type1]]
  END;

ComparableType: PROCEDURE [type: CSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  -- compatibility version
  RETURN [WITH (seb+type) SELECT FROM
    record => comparable OR (~variant OR ComparableUnion[LOOPHOLE[type]]),
    array => comparable OR ComparableType[UnderType[componenttype]],
    ENDCASE => TRUE]
  END;

ComparableUnion: PROCEDURE [rType: recordCSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  sei: ISEIndex;
  type: CSEIndex;
  FOR sei ← (ctxb+(seb+rType).fieldctx).selist, NextSe[sei] UNTIL sei = SENull
    DO
    type ← UnderType[(seb+sei).idtype];
    WITH (seb+type) SELECT FROM
      union => RETURN [equalLengths];
      ENDCASE;
    ENDLOOP;
  RETURN [FALSE]
  END;


END.
```