```
-- file Pass3Xa.Mesa
-- last modified by Satterthwaite, July 16, 1978  10:11 AM

DIRECTORY
  ComData: FROM "comdata"
    USING [
      idCARDINAL, ownSymbols, seAnon,
      typeCHARACTER, typeCONDITION, typeINTEGER, typeSTRING, xref],
  ErrorDefs: FROM "errordefs" USING [error, errorhti, errorn, errortree],
  P3Defs: FROM "p3defs"
    USING [
      ArrangeKeys, BumpCount, Bundling, CanonicalType, CatchPhrase,
      CompleteRecord, DefinedId, Exp, FieldId, ForceType,
      MakeLongType, MakePointerType, OperandType, PopCtx, PushCtx,
      RConst, RecordReference, Rhs, RPop, RPush, RType,
      TargetType, TypeExp, TypeForTree, Unbundle, VariantUnionType, VoidExp,
      XferBody, XferForFrame,
      CheckExprLoop],
  Pass3: FROM "pass3" USING [implicitRecord, implicitType, lockHeld],
  SymDefs: FROM "symdefs"
    USING [bodytype, ctxtype, setype,
      SERecord,
      HTIndex, SEIndex, ISEIndex, CSEIndex, recordCSEIndex, CTXIndex, CBTIndex,
      HTNull, SENull, ISENull, CSENull, BTNull,
      1G, typeANY, typeTYPE],
  SymTabDefs: FROM "symtabdefs"
    USING [
      ConstantId, firstvisiblese, makenonctxse, NextSe, NormalType,
      TypeForm, TypeRoot, UnderType, visiblectxentries],
  TableDefs: FROM "tabledefs" USING [TableBase, TableNotifier],
  TreeDefs: FROM "treedefs"
    USING [treetype,
      NodeName, TreeIndex, TreeLink, TreeMap,
      empty, nullTreeIndex,
      freetree, GetNode, IdentityMap, listhead, listlength, listtail,
      makelist, maketree, mlpop, mlpush, pushproperlist, pushtree,
      setattr, setinfo, testtree, updatelist],
  TypePackDefs: FROM "typepackdefs"
    USING [SymbolTableBase, AssignableTypes, EquivalentTypes];

Pass3Xa: PROGRAM
    IMPORTS
        ErrorDefs, P3Defs, SymTabDefs, TreeDefs, TypePackDefs,
        dataPtr: ComData, passPtr: Pass3
    EXPORTS P3Defs =
  BEGIN
  OPEN SymTabDefs, TreeDefs, P3Defs;

 -- pervasive definitions from SymDefs

  SEIndex: TYPE = SymDefs.SEIndex;
  ISEIndex: TYPE = SymDefs.ISEIndex;
  CSEIndex: TYPE = SymDefs.CSEIndex;
  RecordSEIndex: TYPE = SymDefs.recordCSEIndex;
  SENull: SymDefs.SEIndex = SymDefs.SENull;
  typeANY: SymDefs.CSEIndex = SymDefs.typeANY;

  CTXIndex: TYPE = SymDefs.CTXIndex;


  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)

  own: TypePackDefs.SymbolTableBase;

  ExpANotify: 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[treetype];
    own ← dataPtr.ownSymbols;  RETURN
    END;
```

```
-- tree manipulation utilities

 OperandLhs: PUBLIC PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
   BEGIN
   node: TreeIndex;
     DO
     WITH t SELECT FROM
       symbol =>
         BEGIN
         IF dataPtr.xref THEN RecordReference[index, lhs];
         RETURN [~(seb+index).writeonce]
         END;
       subtree =>
         BEGIN  node ← index;
         IF node = nullTreeIndex THEN RETURN [FALSE];
         SELECT (tb+node).name FROM
           dot =>
             RETURN [WITH (tb+node).son2 SELECT FROM
               symbol => ~(seb+index).writeonce,
               ENDCASE => FALSE];
           dollar =>
             WITH (tb+node).son2 SELECT FROM
               symbol =>
                 BEGIN
                 IF dataPtr.xref THEN RecordReference[index, lhs];
                 IF ~(seb+index).writeonce
                   THEN t ← (tb+node).son1
                   ELSE RETURN [FALSE];
                 END;
               ENDCASE =>  RETURN [FALSE];
           index, loophole, cast, openexp, align =>  t ← (tb+node).son1;
           uparrow, dindex, seqindex, reloc, memory, register =>
             RETURN [TRUE];
           apply =>  RETURN [listlength[(tb+node).son1] = 1];
           ENDCASE =>  RETURN [FALSE];
         END;
       ENDCASE =>  RETURN [FALSE];
     ENDLOOP;
   END;

 LongPath: PUBLIC PROCEDURE [t: TreeLink] RETURNS [long: BOOLEAN] =
   BEGIN
   node: TreeIndex;
   WITH t SELECT FROM
     subtree =>
       BEGIN  node ← index;
       IF node = nullTreeIndex
         THEN  long ← FALSE
         ELSE  SELECT (tb+node).name FROM
           loophole, cast, openexp, align =>
             long ← LongPath[(tb+node).son1];
           ENDCASE
           -- dot, uparrow, dindex, reloc, seqindex, dollar, index -- =>
             long ← (tb+node).attr1;
       END;
     ENDCASE =>  long ← FALSE;
   RETURN
   END;

 OperandInternal: PUBLIC PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
   BEGIN
   node: TreeIndex;
   WITH t SELECT FROM
     symbol =>
       BEGIN
       sei: ISEIndex = index;
       subNode: TreeIndex;
       bti: SymDefs.CBTIndex;
       IF ~(seb+sei).constant THEN RETURN [FALSE];
       IF (seb+sei).mark4
         THEN
           BEGIN
           bti ← (seb+sei).idinfo;
           RETURN [bti # SymDefs.BTNull AND (bb+bti).internal]
           END;
       subNode ← (seb+sei).idvalue;
```

```
      RETURN [WITH (tb+subNode).son3 SELECT FROM
        subtree => (tb+index).name = body AND (tb+index).attr2,
        ENDCASE => FALSE]
      END;
    subtree =>
      BEGIN  node ← index;
      RETURN [SELECT (tb+node).name FROM
        dot, cdot, assignx => OperandInternal[(tb+node).son2],
        ifexp =>
          OperandInternal[(tb+node).son2] OR OperandInternal[(tb+node).son3],
        ENDCASE => FALSE]     -- should check caseexp, bindexp also
      END;
    ENDCASE =>  RETURN [FALSE];
  END;


-- type manipulation

 DiscriminatedType: PROCEDURE [baseType: CSEIndex, t: TreeLink] RETURNS [CSEIndex] =
   BEGIN
   node: TreeIndex;
   type: CSEIndex;
   temp: TreeLink;
   IF t = empty THEN RETURN [passPtr.implicitRecord];
   WITH t SELECT FROM
     subtree =>
       BEGIN  node ← index;
       SELECT (tb+node).name FROM
         unionx =>
           BEGIN
           WITH (tb+node).son1 SELECT FROM
             symbol =>  type ← UnderType[index];
             ENDCASE =>  ERROR;
           WITH (seb+type) SELECT FROM
             record =>
               RETURN [IF variant AND (temp←listtail[(tb+node).son2]) # empty
                 THEN DiscriminatedType[type, temp]
                 ELSE type];
             ENDCASE =>  ERROR;
           END;
         dollar =>  RETURN [OperandType[(tb+node).son1]];
         dot =>
           BEGIN
           type ← NormalType[OperandType[(tb+node).son1]];
           WITH (seb+type) SELECT FROM
             pointer =>  RETURN[UnderType[pointedtotype]];
             ENDCASE =>  ERROR;
           END;
         assignx =>  RETURN [DiscriminatedType[baseType, (tb+node).son2]];
         ENDCASE;
       END;
     ENDCASE;
   RETURN [baseType]
   END;


-- expression list manipulation

 CheckLength: PROCEDURE [t: TreeLink, length: INTEGER] =
   BEGIN
   n: INTEGER = listlength[t];
   SELECT n FROM
     = length => NULL;
     > length => ErrorDefs.errorn[listLong, n-length];
     < length => ErrorDefs.errorn[listShort, length-n];
     ENDCASE;
   RETURN
   END;

 KeyedList: PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
   BEGIN
   RETURN [t # empty AND testtree[listhead[t], item]]
   END;

 ContextComplete: PROCEDURE [ctx: CTXIndex] RETURNS [BOOLEAN] =
   BEGIN
```

```
    RETURN [WITH (ctxb+ctx) SELECT FROM
      simple => TRUE,
      included => ctxcomplete,
      ENDCASE => FALSE]
    END;

  MatchFields: PUBLIC PROCEDURE [record: RecordSEIndex, expList: TreeLink, elisions: BOOLEAN]
      RETURNS [val: TreeLink] =
    BEGIN
    nFields: INTEGER;
    ctx: CTXIndex;
    sei: ISEIndex;

    EvaluateField: TreeMap =
      BEGIN
      IF t # empty
        THEN
          BEGIN
          v ← Rhs[t, IF sei = SENull
            THEN typeANY
            ELSE TargetType[UnderType[(seb+sei).idtype]]];
          RPop[];
          END
        ELSE
          BEGIN  v ← empty;
          IF ~elisions AND sei # SENull THEN ErrorDefs.error[elision];
          END;
      IF sei # SENull THEN sei ← NextSe[sei];
      RETURN
      END;

    KeyFillError: PROCEDURE [sei: ISEIndex] RETURNS [TreeLink] =
      BEGIN
      ErrorDefs.errorhti[omittedKey, (seb+sei).htptr];
      RETURN [TreeLink[symbol[index: dataPtr.seAnon]]];
      END;

    IF record = SENull
      THEN
        BEGIN  nFields ← 0;  sei ← SymDefs.ISENull;
        IF expList # empty
          THEN ErrorDefs.errorn[listLong, listlength[expList]];
        END
      ELSE
        BEGIN
        CompleteRecord[record];
        IF ~ContextComplete[(seb+record).fieldctx]
          THEN
            BEGIN  ErrorDefs.error[noAccess];
            nFields ← 0;  sei ← SymDefs.ISENull;
            END
          ELSE
            BEGIN  ctx ← (seb+record).fieldctx;
            IF KeyedList[expList]
              THEN
                BEGIN
                nFields ← ArrangeKeys[expList, ctx, KeyFillError];
                expList ← makelist[nFields];
                END
              ELSE
                BEGIN  nFields ← visiblectxentries[ctx];
                IF nFields # 1 OR expList # empty
                  THEN  CheckLength[expList, nFields];
                END;
            sei ← firstvisiblese[ctx];
            END;
        END;
    IF expList # empty
      THEN  val ← updatelist[expList, EvaluateField]
      ELSE
        BEGIN   -- resolve length 0/length 1 ambiguity
        IF nFields = 0
          THEN  val ← empty
          ELSE
            BEGIN
            IF ~elisions THEN ErrorDefs.error[elision];
```

```
          mlpush[empty];  pushproperlist[1];  val ← mlpop[];
          END;
      END;
  RETURN
  END;


BumpFieldRefs: PUBLIC PROCEDURE [record: RecordSEIndex] =
  BEGIN
  sei: ISEIndex;
  IF record # SENull
    THEN
      FOR sei ← (ctxb+(seb+record).fieldctx).selist, NextSe[sei] UNTIL sei = SENull
        DO  BumpCount[sei]  ENDLOOP;
  RETURN
  END;


-- operators

Dot: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  type, rType, nType: CSEIndex;
  sei: ISEIndex;
  fieldHti: SymDefs.HTIndex;
  op: NodeName;
  matched, const, long: BOOLEAN;
  nHits: CARDINAL;
  nDerefs: CARDINAL;
  son1 ← Exp[son1, typeANY];  type ← RType[];  RPop[];
  WITH son2 SELECT FROM
    hash =>  fieldHti ← index;
    ENDCASE =>  ERROR;
  op ← dollar;  nDerefs ← 0;  long ← LongPath[son1];
  -- N.B. failure is avoided only by EXITing the following loop
    DO
    nType ← NormalType[type];
    WITH (seb+nType) SELECT FROM
      record =>
        BEGIN
        [nHits, sei] ← FieldId[fieldHti, LOOPHOLE[nType, RecordSEIndex]];
        SELECT nHits FROM
          0 => NULL;
          1 => EXIT;
          ENDCASE => GO TO ambiguous;
        IF Bundling[nType] = 0 THEN GO TO nomatch;
        type ← Unbundle[LOOPHOLE[nType, RecordSEIndex]];
        son1 ← IF op = dot
                THEN Dereference[son1, type, long]
                ELSE ForceType[son1, type];
        op ← dollar;
        END;
      pointer =>
        BEGIN
        IF (nDerefs ← nDerefs+1) > 255 THEN GO TO nomatch;
        IF op = dot THEN son1 ← Dereference[son1, type, long];
        long ← (seb+type).typetag = long;
        op ← dot;  dereferenced ← TRUE;  type ← UnderType[pointedtotype];
        END;
      definition =>
        BEGIN
        [matched, sei] ← DefinedId[fieldHti, nType];
        IF matched THEN  BEGIN  op ← cdot;  EXIT  END;
        GO TO nomatch;
        END;
      ENDCASE =>  GO TO nomatch;
    REPEAT
      nomatch =>
        BEGIN
        IF fieldHti # SymDefs.HTNull
          THEN ErrorDefs.errorhti[unknownField, fieldHti];
        sei ← dataPtr.seAnon;
        END;
      ambiguous =>
        BEGIN
        ErrorDefs.errorhti[ambiguousId, fieldHti];  sei ← dataPtr.seAnon;
```

```
        END;
      ENDLOOP;
    son2 ← TreeLink[symbol[index: sei]];
    rType ← UnderType[(seb+sei).idtype];
    const ← ConstantId[sei];
    IF const
      THEN  name ← cdot
      ELSE  BEGIN  name ← op;  attr1 ← long  END;
    RPush[rType, const];  RETURN
    END;

  Dereference: PROCEDURE [t: TreeLink, type: CSEIndex, long: BOOLEAN] RETURNS [TreeLink] =
    BEGIN
    mlpush[t];  pushtree[uparrow, 1];  setinfo[type];  setattr[1, long];
    RETURN[mlpop[]]
    END;

  UpArrow: PUBLIC PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    type, nType: CSEIndex;
    son1 ← Exp[son1, typeANY];
    type ← RType[];  RPop[];
      DO
      nType ← NormalType[type];
      WITH (seb+nType) SELECT FROM
        pointer =>
          BEGIN
          dereferenced ← TRUE;  RPush[UnderType[pointedtotype], FALSE];
          attr1 ← (seb+type).typetag = long;  EXIT
          END;
        record =>
          BEGIN
          IF Bundling[nType] = 0 THEN GO TO fail;
          type ← Unbundle[LOOPHOLE[nType, RecordSEIndex]];
          END;
        ENDCASE => GO TO fail;
      REPEAT
        fail =>
          BEGIN
          IF type # typeANY THEN  ErrorDefs.errortree[typeClash, son1];
          RPush[type, FALSE];
          END;
      ENDLOOP;
    RETURN
    END;


  Apply: PUBLIC PROCEDURE [node: TreeIndex, target: CSEIndex, mustXfer: BOOLEAN] =
    BEGIN  OPEN (tb+node);
    opType, type, nType, subType: CSEIndex;
    nDerefs: CARDINAL;
    const, desc, long: BOOLEAN;

    ApplyError: PROCEDURE [warn: BOOLEAN] =
      BEGIN
      IF warn THEN ErrorDefs.errortree[noApplication, son1];
      son2 ← updatelist[son2, VoidExp];
      RPush[typeANY, FALSE];  RETURN
      END;

    UniOperand: PROCEDURE RETURNS [valid: BOOLEAN] =
      BEGIN
      IF ~(valid ← listlength[son2] = 1)
        THEN
          BEGIN
          CheckLength[son2, 1];
          son2 ← updatelist[son2, VoidExp];
          RPush[typeANY, FALSE];
          END
        ELSE IF KeyedList[son2] THEN ErrorDefs.error[keys];
      RETURN
      END;

    IF son1 # empty
      THEN
        BEGIN
```

```
      WITH (seb+target) SELECT FROM
        union =>
          BEGIN  PushCtx[casectx];
          son1 ← Exp[son1, typeANY];  PopCtx[];
          END;
        ENDCASE =>  son1 ← Exp[son1, typeANY];
      opType ← RType[];  const ← RConst[];  RPop[];
      IF opType = SymDefs.typeTYPE
        THEN type ← UnderType[TypeForTree[son1]];
      END
    ELSE
      BEGIN  opType ← SymDefs.typeTYPE;
      SELECT (seb+target).typetag FROM
        record =>  type ← TypeRoot[target];
        array =>  type ← target;
        ENDCASE =>
          BEGIN  type ← typeANY;
          ErrorDefs.errortree[noTarget, [subtree[node]]];
          END;
      END;
  nDerefs ← 0;  desc ← FALSE;  long ← LongPath[son1];
  -- dereferencing/deproceduring loop
    DO
    nType ← NormalType[opType];
    WITH (seb+nType) SELECT FROM
      mode =>
        BEGIN
        SELECT (seb+type).typetag FROM
          record =>  Construct[node, LOOPHOLE[type, RecordSEIndex]];
          array =>  RowCons[node, type];
          enumerated, subrange, basic =>
            IF UniOperand[]
              THEN
                BEGIN
                son1 ← Rhs[son2, TargetType[type]];
                son2 ← empty;  name ← loophole;
                const ← RConst[];  RPop[];
                RPush[type, const];
                END;
          ENDCASE =>  ApplyError[type#typeANY];
        EXIT
        END;
      transfer =>
        BEGIN
        SELECT mode FROM
          procedure =>
            IF ~passPtr.lockHeld AND OperandInternal[son1]
              THEN  ErrorDefs.errortree[internalCall, son1];
          program =>
            IF XferBody[son1] # SymDefs.BTNull
              THEN ErrorDefs.errortree[typeClash, son1];
          ENDCASE;
        son2 ← MatchFields[inrecord, son2, FALSE];
        RPush[outrecord, FALSE];
        name ← SELECT mode FROM
          procedure => call,
          port => portcall,
          process => join,
          signal => signal,
          error => error,
          program => start,
          ENDCASE => apply;
        EXIT
        END;
      array =>
        BEGIN
        IF UniOperand[]
          THEN
            BEGIN
            IF KeyedList[son2] THEN ErrorDefs.error[keys];
            son2 ← Rhs[son2, TargetType[UnderType[indextype]]];
            END;
        RPop[];  RPush[UnderType[componenttype], FALSE];
        IF mustXfer
          THEN
            BEGIN
```

```
        opType ← RType[]; RPop[];
        mlpush[son1]; mlpush[son2];
        pushtree[IF desc THEN dindex ELSE index, 2];
        setinfo[opType]; setattr[1, long];
        son1 ← mlpop[]; son2 ← empty;
        IF nsons > 2 THEN ErrorDefs.error[misplacedCatch];
        mustXfer ← FALSE; -- to avoid looping
        END
      ELSE
        BEGIN
        name ← IF desc THEN dindex ELSE index; attr1 ← long;
        EXIT
        END;
    END;
  arraydesc =>
    BEGIN
    long ← (seb+opType).typetag = long;
    opType ← UnderType[describedType]; const ← FALSE; desc ← TRUE;
    END;
  pointer =>
    SELECT TRUE FROM
      basing =>
        BEGIN
        IF UniOperand[]
          THEN
            BEGIN
            son2 ← Rhs[son2, typeANY];
            subType ← CanonicalType[RType[]];
            RPop[];
            WITH (seb+subType) SELECT FROM
              relative =>
                BEGIN
                IF ~TypePackDefs.AssignableTypes[
                          [own, UnderType[baseType]],
                          [own, opType]]
                  THEN ErrorDefs.errortree[typeClash, son1];
                type ← UnderType[resultType];
                END;
              ENDCASE =>
                BEGIN type ← typeANY;
                IF subType # typeANY
                  THEN ErrorDefs.errortree[typeClash, son2];
                END;
            subType ← NormalType[type];
            attr1 ← (seb+opType).typetag = long
                      OR (seb+type).typetag = long;
            attr2 ← (seb+subType).typetag = arraydesc;
            WITH (seb+subType) SELECT FROM
              pointer =>
                BEGIN
                dereferenced ← TRUE; type ← UnderType[pointedtotype];
                END;
              arraydesc => type ← UnderType[describedType];
              ENDCASE;
            RPush[type, FALSE]; name ← reloc;
            END;
        EXIT
        END;
      nType = dataPtr.typeSTRING =>
        BEGIN
        IF UniOperand[]
          THEN
            BEGIN dereferenced ← TRUE;
            son2 ← Rhs[son2, dataPtr.typeINTEGER];
            RPop[]; RPush[dataPtr.typeCHARACTER, FALSE];
            name ← seqindex; attr1 ← (seb+opType).typetag = long;
            END;
        EXIT
        END;
      ENDCASE =>
        BEGIN
        const ← FALSE;
        dereferenced ← TRUE; subType ← UnderType[pointedtotype];
        WITH (seb+subType) SELECT FROM
          record =>
            IF (ctxb+fieldctx).ctxlevel = SymDefs.lG
```

```
                THEN
                  BEGIN  opType ← XferForFrame[fieldctx];
                  son1 ← ForceType[son1, opType];
                  END
                ELSE GO TO deRef;
            ENDCASE => GO TO deRef;
          EXITS
            deRef =>
              BEGIN
              IF (nDerefs ← nDerefs+1) > 255 THEN GO TO fail;
              long ← (seb+opType).typetag = long;
              son1 ← Dereference[son1, subType, long];
              opType ← subType;
              END;
          END;
      record =>
        BEGIN
        IF nType = dataPtr.typeCONDITION
          THEN
            BEGIN
            IF son2 # empty
              THEN ErrorDefs.errorn[listLong, listlength[son2]];
            RPush[SymDefs.CSENull, FALSE];
            name ← wait;
            EXIT
            END;
        IF Bundling[opType] = 0 THEN GO TO fail;
        opType ← Unbundle[LOOPHOLE[opType, RecordSEIndex]];
        son1 ← ForceType[son1, opType];
        END;
      ENDCASE =>  GO TO fail;
    REPEAT
      fail =>  ApplyError[opType#typeANY OR nDerefs#0];
    ENDLOOP;
  IF nsons > 2 THEN
    BEGIN
    SELECT name FROM
      call, portcall, signal, error, start, fork, join, wait, apply =>
        NULL;
      ENDCASE =>  ErrorDefs.error[misplacedCatch];
    [] ← CatchPhrase[son3];
    END;
  RETURN
  END;

Construct: PROCEDURE [node: TreeIndex, type: RecordSEIndex] =
  BEGIN  OPEN (tb+node);
  cType: CSEIndex ← type;
  t: TreeLink;
  son2 ← MatchFields[type, son2, TRUE];
  WITH (seb+type) SELECT FROM
    linked =>
      BEGIN
      name ← unionx;
      cType ← VariantUnionType[linktype];
      END;
    ENDCASE =>
      BEGIN
      name ← constructx;
      IF variant AND (t←listtail[son2]) # empty
        THEN  cType ← DiscriminatedType[type, t];
      END;
  info ← cType;  RPush[cType, FALSE];
  RETURN
  END;

RowCons: PROCEDURE [node: TreeIndex, type: CSEIndex] =
  BEGIN  OPEN (tb+node);
  cType: CSEIndex = TargetType[WITH (seb+type) SELECT FROM
      array => UnderType[componenttype],
      ENDCASE => typeANY];

  MapValue: TreeMap =
    BEGIN
    IF t # empty
      THEN  BEGIN  v ← Rhs[t, cType];  RPop[]  END
```

```
      ELSE  v ← empty;
    RETURN
    END;

  IF KeyedList[son2] THEN ErrorDefs.error[keys];
  son2 ← updatelist[son2, MapValue];
  name ← rowconsx;  info ← type;
  RPush[type, FALSE];
  END;


Assignment: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  lhsType, rhsType: CSEIndex;
  son1 ← Exp[son1, typeANY ! CheckExprLoop => RESUME [FALSE]];
  IF ~OperandLhs[son1] THEN ErrorDefs.errortree[nonLHS, son1];
  lhsType ← RType[];  RPop[];
  son2 ← Rhs[son2, TargetType[lhsType]];
  IF (seb+lhsType).typetag = union
    THEN
      IF ~TypePackDefs.AssignableTypes[
       [own, DiscriminatedType[typeANY, son1]],
       [own, DiscriminatedType[typeANY, son2]]]
        THEN ErrorDefs.errortree[typeClash, son2];
  rhsType ← RType[];  RPop[];
  RPush[rhsType, FALSE];  RETURN
  END;


Extract: PUBLIC PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  type: CSEIndex;
  ctx: CTXIndex;
  sei: ISEIndex;
  nL, nR: CARDINAL;
  saveRecord: RecordSEIndex = passPtr.implicitRecord;

  FillNull: PROCEDURE [ISEIndex] RETURNS [TreeLink] =
    BEGIN
    RETURN [empty]
    END;

  PushItem: TreeMap =
    BEGIN
    mlpush[t];  RETURN [empty];
    END;

  AssignItem: TreeMap =
    BEGIN
    saveType: CSEIndex = passPtr.implicitType;
    IF t = empty
      THEN  v ← empty
      ELSE
        BEGIN
        passPtr.implicitType ← IF sei = SENull
              THEN typeANY
              ELSE UnderType[(seb+sei).idtype];
        mlpush[t];  mlpush[empty];  v ← maketree[assign, 2];
        Assignment[GetNode[v]];  RPop[];
        END;
    IF sei # SENull THEN sei ← NextSe[sei];
    passPtr.implicitType ← saveType;  RETURN
    END;

  son2 ← Exp[son2, typeANY];  type ← RType[];  RPop[];
  IF type = SENull
    THEN
      BEGIN  ErrorDefs.errortree[typeClash, son2];
      type ← typeANY;  nR ← 0;  sei ← SymDefs.ISENull;
      END
    ELSE
      BEGIN
      type ← TypeRoot[type];
      WITH (seb+type) SELECT FROM
        record =>
          BEGIN
```

```
          CompleteRecord[LOOPHOLE[type, RecordSEIndex]];
          IF ContextComplete[fieldctx]
            THEN
              BEGIN
              passPtr.implicitRecord ← LOOPHOLE[type, RecordSEIndex];
              ctx ← fieldctx;  sei ← firstvisiblese[ctx];
              nR ← visiblectxentries[ctx];
              END
            ELSE
              BEGIN  ErrorDefs.error[noAccess];
              type ← typeANY;  nR ← 0;  sei ← SymDefs.ISENull;
              END;
          END;
        ENDCASE =>
          BEGIN
          IF type # typeANY THEN ErrorDefs.errortree[typeClash, son2];
          type ← typeANY;  nR ← 0;  sei ← SymDefs.ISENull;
          END;
      END;
  IF KeyedList[son1] AND type # typeANY
    THEN  nL ← ArrangeKeys[son1, ctx, FillNull]
    ELSE
      BEGIN
      nL ← listlength[son1];
      son1 ← freetree[updatelist[son1, PushItem]];
      IF nL > nR AND type # typeANY
        THEN  ErrorDefs.errorn[listLong, nL-nR];
      THROUGH (nL .. nR] DO  mlpush[empty]  ENDLOOP;
      nL ← MAX[nL, nR];
      END;
  pushproperlist[nR];  setinfo[type];
  son1 ← updatelist[mlpop[], AssignItem];
  passPtr.implicitRecord ← saveRecord;  RETURN
  END;


Addr: PUBLIC PROCEDURE [node: TreeIndex, target: CSEIndex] =
  BEGIN  OPEN (tb+node);
  type: CSEIndex;
  son1 ← Exp[son1, typeANY ! CheckExprLoop => RESUME [FALSE]];
  IF ~OperandLhs[son1] THEN ErrorDefs.errortree[nonAddressable, son1];
  type ← MakePointerType[RType[], NormalType[target]];
  IF (attr1 ← LongPath[son1])
    THEN  type ← MakeLongType[type, target];
  RPop[];  RPush[type, FALSE];  RETURN
  END;


DescOp: PUBLIC PROCEDURE [node: TreeIndex, target: CSEIndex] =
  BEGIN
  SELECT (tb+node).name FROM
    base => Base[node, target];
    length => Length[node];
    arraydesc => Desc[node, target];
    ENDCASE => ERROR;
  RETURN
  END;


StripRelative: PROCEDURE [rType: CSEIndex] RETURNS [type: CSEIndex, bType: SEIndex] =
  BEGIN
  WITH (seb+rType) SELECT FROM
    relative =>  BEGIN type ← UnderType[offsetType]; bType ← baseType END;
    ENDCASE =>   BEGIN type ← rType; bType ← SENull END;
  RETURN
  END;

MakeRelativeType: PROCEDURE [type: CSEIndex, bType: SEIndex, hint: CSEIndex]
    RETURNS [CSEIndex] =
  BEGIN
  rType, tType: CSEIndex;
  WITH (seb+hint) SELECT FROM
    relative =>
      IF offsetType = type AND UnderType[baseType] = UnderType[bType]
        THEN  RETURN [hint];
```

```
    ENDCASE;
  tType ← IF TypeForm[bType] = long OR TypeForm[type] = long
            THEN  MakeLongType[NormalType[type], type]
            ELSE  type;
  rType ← makenonctxse[SIZE[relative constructor SymDefs.SERecord]];
  (seb+rType).typeinfo ← relative[
            baseType: bType,
            offsetType: type,
            resultType: tType];
  (seb+rType).mark3 ← (seb+rType).mark4 ← TRUE;
  RETURN [rType]
  END;


Base: PROCEDURE [node: TreeIndex, target: CSEIndex] =
  BEGIN  OPEN (tb+node);
  type, aType, nType, subTarget: CSEIndex;
  bType: SEIndex;
  long: BOOLEAN;
  IF listlength[son1] = 1
    THEN
      BEGIN
      son1 ← Exp[son1, typeANY];
      [aType, bType] ← StripRelative[CanonicalType[RType[]]];
      RPop[];
      nType ← NormalType[aType];  [subTarget, ] ← StripRelative[target];
      WITH (seb+nType) SELECT FROM
        array =>
          BEGIN  name ← addr;
          IF ~OperandLhs[son1]
            THEN ErrorDefs.errortree[nonAddressable, son1];
          long ← LongPath[son1];
          END;
        arraydesc =>
          BEGIN
          long ← (seb+aType).typetag = long;
          nType ← UnderType[describedType];
          END;
        ENDCASE =>
          IF nType # typeANY THEN ErrorDefs.errortree[typeClash, son1];
      END
    ELSE
      BEGIN
      CheckLength[son1, 1];  son1 ← updatelist[son1, VoidExp];
      long ← FALSE;
      END;
  type ← MakePointerType[nType, NormalType[subTarget]];
  IF (attr1 ← long) THEN type ← MakeLongType[type, subTarget];
  IF bType # SENull THEN type ← MakeRelativeType[type, bType, target];
  RPush[type, FALSE];  RETURN
  END;

Length: PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  type: CSEIndex;
  const: BOOLEAN;
  IF listlength[son1] = 1
    THEN
      BEGIN
      son1 ← Exp[son1, typeANY];
      type ← RType[];  RPop[];
      type ← IF (seb+type).mark3
             THEN NormalType[StripRelative[CanonicalType[type]].type]
             ELSE typeANY;
      WITH (seb+type) SELECT FROM
        array =>  const ← TRUE;
        arraydesc =>  const ← FALSE;
        ENDCASE =>
          BEGIN  const ← TRUE;
          IF type # typeANY THEN ErrorDefs.errortree[typeClash, son1];
          END;
      END
    ELSE
      BEGIN  const ← TRUE;
      CheckLength[son1, 1];  son1 ← updatelist[son1, VoidExp];
      END;
```

```
    RPush[dataPtr.typeINTEGER, const];  RETURN
    END;

  Desc: PROCEDURE [node: TreeIndex, target: CSEIndex] =
    BEGIN  OPEN (tb+node);
    type, subType: CSEIndex;
    aType, bType, cType: SEIndex;
    fixed, long: BOOLEAN;
    subNode: TreeIndex;
    subTarget: CSEIndex = StripRelative[target].type;
    nTarget: CSEIndex = NormalType[subTarget];
    aType ← bType ← SENull;
    SELECT listlength[son1] FROM
      1 =>
        BEGIN
        son1 ← Exp[son1, typeANY
                ! CheckExprLoop => RESUME [FALSE]];
        IF ~OperandLhs[son1] THEN ErrorDefs.errortree[nonAddressable, son1];
        long ← LongPath[son1];
        subType ← CanonicalType[RType[]];  RPop[];
        IF (seb+subType).typetag = array
          THEN  BEGIN  aType ← OperandType[son1];  fixed ← TRUE  END
          ELSE
            BEGIN  fixed ← FALSE;
            IF subType # typeANY THEN ErrorDefs.errortree[typeClash, son1];
            END;
        mlpush[son1];
        pushtree[addr, 1];
        setinfo[MakePointerType[subType, typeANY]];  setattr[1, long];
        mlpush[IdentityMap[son1]];
        pushtree[length, 1];  setinfo[dataPtr.typeINTEGER];
        mlpush[empty];
        son1 ← makelist[3];
        END;
      3 =>
        BEGIN  subNode ← GetNode[son1];
        (tb+subNode).son1 ← Exp[(tb+subNode).son1, typeANY];
        [subType,bType] ← StripRelative[CanonicalType[RType[]]];
        RPop[];
        SELECT (seb+NormalType[subType]).typetag FROM
          basic, pointer => NULL;
          ENDCASE => ErrorDefs.errortree[typeClash, (tb+subNode).son1];
        long ← (seb+subType).typetag = long;
        (tb+subNode).son2 ← Rhs[(tb+subNode).son2, dataPtr.typeINTEGER];
        RPop[];
        IF (fixed ← (tb+subNode).son3 # empty)
          THEN
            BEGIN
            (tb+subNode).son3 ← TypeExp[(tb+subNode).son3];
            cType ← TypeForTree[(tb+subNode).son3];
            END;
        END;
      ENDCASE;
    IF aType = SENull
      THEN
        BEGIN
        WITH (seb+nTarget) SELECT FROM
          arraydesc =>
            BEGIN  subType ← UnderType[describedType];
            WITH t: (seb+subType) SELECT FROM
              array =>
                IF ~fixed
                 OR TypePackDefs.EquivalentTypes[
                        [own, UnderType[t.componenttype]],
                        [own, UnderType[cType]]]
                  THEN  BEGIN  aType ← describedType;  GO TO old  END;
            ENDCASE;
            END;
          ENDCASE;
        GO TO new;
        EXITS
          old => NULL;
          new =>
            BEGIN
            subType ← makenonctxse[SIZE[array constructor SymDefs.SERecord]];
            (seb+subType).typeinfo ← array[
```

```
            packed: FALSE,
            lengthUsed: FALSE,
            comparable: FALSE,
            indextype: dataPtr.idCARDINAL,
            componenttype: IF fixed THEN cType ELSE typeANY];
          (seb+subType).mark3 ← (seb+subType).mark4 ← TRUE;
          aType ← subType;
          END;
      END;
  -- make type description
    BEGIN
    WITH t: (seb+nTarget) SELECT FROM
      arraydesc =>
        IF TypePackDefs.EquivalentTypes[
            [own, UnderType[t.describedType]],
            [own, UnderType[aType]]]
          THEN GO TO old;
      ENDCASE =>
        IF ~fixed AND target = typeANY
          THEN ErrorDefs.errortree[noTarget, [subtree[node]]];
    GO TO new;
    EXITS
      old =>  type ← nTarget;
      new =>
        BEGIN
        type ← makenonctxse[SIZE[arraydesc constructor SymDefs.SERecord]];
        (seb+type).typeinfo ← arraydesc[describedType: aType];
        (seb+type).mark3 ← (seb+type).mark4 ← TRUE;
        END;
    END;
  IF (attr1 ← long) THEN type ← MakeLongType[type, subTarget];
  IF bType # SENull THEN type ← MakeRelativeType[type, bType, target];
  RPush[type, FALSE];  RETURN
  END;

END.
```