```
-- file SymbolCopier.Mesa
-- last modified by Satterthwaite, July 16, 1978  10:15 AM

DIRECTORY
  CopierDefs: FROM "copierdefs",
  InlineDefs: FROM "inlinedefs",
  LitDefs: FROM "litdefs",
  StringDefs: FROM "stringdefs",
  SymbolTableDefs: FROM "symboltabledefs",
  SymDefs: FROM "symdefs",
  SymSegDefs: FROM "symsegdefs",
  SymTabDefs: FROM "symtabdefs",
  SystemDefs: FROM "systemdefs",
  TableDefs: FROM "tabledefs",
  TreeDefs: FROM "treedefs";

SymbolCopier: PROGRAM
    IMPORTS
        CopierDefs, LitDefs, SymbolTableDefs, SymSegDefs, SymTabDefs,
        SystemDefs, TableDefs, TreeDefs
    EXPORTS CopierDefs SHARES CopierDefs =
  BEGIN
  OPEN SymTabDefs, SymDefs;

 -- tables defining the current symbol table

  seb: TableDefs.TableBase;              -- se table
  ctxb: TableDefs.TableBase;             -- context table
  mdb: TableDefs.TableBase;              -- module directory base
  bb: TableDefs.TableBase;               -- body table

  CopierNotify: TableDefs.TableNotifier =
    BEGIN  -- called whenever the main symbol table is repacked
    seb ← base[setype];
    ctxb ← base[ctxtype];  mdb ← base[mdtype];
    bb ← base[bodytype];
    RETURN
    END;


 -- table bases for the current include module

  iBase: SymbolTableDefs.SymbolTableBase;

  iHt: DESCRIPTOR FOR ARRAY --HTIndex-- OF HTRecord;
  iSeb: TableDefs.TableBase;
  iCtxb: TableDefs.TableBase;


  SearchMemo: TYPE = RECORD[
    hti: HTIndex,
    ctx: CTXIndex];

  NullMemo: SearchMemo = SearchMemo[hti:HTNull, ctx:CTXNull];

  MemoCacheSize: CARDINAL = 777B;
  memoCache: DESCRIPTOR FOR ARRAY OF SearchMemo;


  CopierInit: PUBLIC PROCEDURE =
    BEGIN  OPEN SystemDefs;
    i: CARDINAL;
    TableDefs.AddNotify[CopierNotify];
    memoCache ← DESCRIPTOR [AllocateSegment[MemoCacheSize*SIZE[SearchMemo]], MemoCacheSize];
    FOR i IN [0 .. MemoCacheSize) DO memoCache[i] ← NullMemo ENDLOOP;
    SymbolTableDefs.SetSymbolCacheSize[100];
    RETURN
    END;

  CopierReset: PUBLIC PROCEDURE =
    BEGIN
    SymbolTableDefs.SetSymbolCacheSize[0];
    SystemDefs.FreeSegment[BASE[memoCache]];
    ResetIncludeContexts[];  TableDefs.DropNotify[CopierNotify];
    RETURN
    END;
```

```
  SubString: TYPE = StringDefs.SubString;
  SubStringDescriptor: TYPE = StringDefs.SubStringDescriptor;


  SearchFileCtx: PUBLIC PROCEDURE [hti: HTIndex, ctx: includedCTXIndex] RETURNS [found: BOOLEAN, sei: I
**SEIndex] =
    BEGIN
    desc: SubStringDescriptor;
    s: SubString = @desc;
    hash: [0..MemoCacheSize);
    iHti: HTIndex;
    isei: ISEIndex;
    mdi: MDIndex = (ctxb+ctx).ctxmodule;
    ignoreprivate: BOOLEAN = (mdb+mdi).mdshared;
    SubStringForHash[s, hti];
    hash ← InlineDefs.LongDivMod[
                InlineDefs.LongMult[LOOPHOLE[hti], LOOPHOLE[ctx]],
                MemoCacheSize].remainder;
    IF memoCache[hash].hti = hti AND memoCache[hash].ctx = ctx
      THEN RETURN [FALSE, ISENull];
    IF OpenIncludedTable[mdi]
      THEN
        BEGIN
        iHti ← iBase.FindString[s];
        IF iHti # HTNull
          AND
         (iHt[iHti].anyPublic
           OR (ignoreprivate AND iHt[iHti].anyInternal))
          THEN
            BEGIN
            isei ← iBase.SearchContext[iHti, (ctxb+ctx).ctxmap];
            found ← isei # SENull AND ((iSeb+isei).public OR ignoreprivate);
            IF found THEN sei ← CopyCtxSe[isei, hti, ctx, mdi];
            END
          ELSE found ← FALSE;
        CloseIncludedTable[];
        END
      ELSE
        BEGIN  found ← FALSE;  sei ← ISENull
        END;
    IF ~found THEN  memoCache[hash] ← SearchMemo[hti:hti, ctx:ctx];
    RETURN
    END;


  CompleteContext: PUBLIC PROCEDURE [ctx: includedCTXIndex, ignoreprivate: BOOLEAN] =
    BEGIN
    IF ~(ctxb+ctx).ctxreset AND OpenIncludedTable[(ctxb+ctx).ctxmodule]
      THEN
        BEGIN
        FillContext[ctx, ignoreprivate];  CloseIncludedTable[];
        END;
    RETURN
    END;


  CopyUnion: PUBLIC PROCEDURE [ctx: CTXIndex] =
    BEGIN
    isei, iroot: ISEIndex;
    WITH (ctxb+ctx) SELECT FROM
      included =>
        IF ~ctxreset AND OpenIncludedTable[ctxmodule]
          THEN
            BEGIN
            isei ← iroot ← (iCtxb+ctxmap).selist;
              DO
              IF isei = SENull THEN EXIT;
              IF iBase.TypeForm[(iSeb+isei).idtype] = union
                THEN
                  BEGIN
                  IF (iSeb+isei).htptr # HTNull
                    THEN [] ← CopyIncludedSymbol[isei, ctxmodule]
                    ELSE FillContext[LOOPHOLE[ctx], TRUE];
```

```
                EXIT
                END;
            IF (isei ← iBase.NextSe[isei]) = iroot THEN EXIT;
            ENDLOOP;
          CloseIncludedTable[];
          END;
    ENDCASE;
  RETURN
  END;


FillContext: PROCEDURE [ctx: includedCTXIndex, ignoreprivate: BOOLEAN] =
  BEGIN
  sei, isei, psei: ISEIndex;
  complete: BOOLEAN;
  mdi: MDIndex = (ctxb+ctx).ctxmodule;
  hti: HTIndex;
  ignoreprivate ← ignoreprivate OR (mdb+mdi).mdshared;
  complete ← TRUE;  psei ← ISENull;
  FOR isei ← iBase.FirstCtxSe[(ctxb+ctx).ctxmap], iBase.NextSe[isei] UNTIL isei = SENull
    DO
    IF ~((iSeb+isei).public OR ignoreprivate)
      THEN  complete ← FALSE
      ELSE
        BEGIN  hti ← MapHti[(iSeb+isei).htptr];
        sei ← SearchContext[hti, ctx];
        IF sei = SENull THEN sei ← CopyCtxSe[isei, hti, ctx, mdi];
        IF psei # SENull AND NextSe[psei] # sei
          THEN
            BEGIN  Delink[sei];
            setselink[sei, NextSe[psei]];
            setselink[psei, sei];
            END;
        (ctxb+ctx).selist ← psei ← sei;
        END;
    ENDLOOP;
  resetctx[ctx];  (ctxb+ctx).ctxcomplete ← complete;
  RETURN
  END;

Delink: PUBLIC PROCEDURE [sei: ISEIndex] =
  BEGIN
  prev, next: ISEIndex;
  ctx: CTXIndex = (seb+sei).ctxnum;    -- assumed not reset
  prev ← (ctxb+ctx).selist;
    DO
    next ← NextSe[prev];
    SELECT next FROM
      sei => EXIT;
      (ctxb+ctx).selist, ISENull => ERROR;
      ENDCASE => prev ← next;
    ENDLOOP;
  IF NextSe[sei] = sei
    THEN (ctxb+ctx).selist ← ISENull
    ELSE
      BEGIN
      IF sei = (ctxb+ctx).selist THEN (ctxb+ctx).selist ← prev;
      setselink[prev, NextSe[sei]];
      END;
  setselink[sei, ISENull];  RETURN
  END;


MapHti: PROCEDURE [iHti: HTIndex] RETURNS [hti: HTIndex] =
  BEGIN
  desc: SubStringDescriptor;
  s: SubString = @desc;
  IF iHti = HTNull
    THEN  hti ← HTNull
    ELSE
      BEGIN
      iBase.SubStringForHash[s, iHti];
      hti ← EnterString[s ! TableRelocated => s.base ← iBase.ssb];
      END;
  RETURN
  END;
```

```
  MissingHti: ERROR = CODE;

  InverseMapHti: PROCEDURE [hti: HTIndex] RETURNS [iHti: HTIndex] =
    BEGIN
    -- N.B.  assumes that the included table has been selected
    desc: SubStringDescriptor;
    s: SubString = @desc;
    IF hti = HTNull
      THEN  iHti ← HTNull
      ELSE
        BEGIN
        SubStringForHash[s, hti];
        iHti ← iBase.FindString[s];
        IF iHti = HTNull THEN ERROR MissingHti;
        END;
    RETURN
    END;


  FindIncludedCtx: PUBLIC PROCEDURE [mdi: MDIndex, ictx: CTXIndex] RETURNS [includedCTXIndex] =
    BEGIN
    ctx, last: includedCTXIndex;
    target: CTXIndex;
    mdroot: MDIndex;
    desc: SubStringDescriptor;
    s: SubString = @desc;
    WITH (iCtxb+ictx) SELECT FROM
      included =>
        BEGIN
        iBase.SubStringForHash[s, (iBase.mdb+ctxmodule).mdhti];
        mdroot ← CopierDefs.FindMdEntry[s, (iBase.mdb+ctxmodule).mdStamp
            !TableRelocated => s.base ← iBase.ssb];
        target ← ctxmap;
        END;
      ENDCASE =>
        BEGIN  mdroot ← mdi;  target ← ictx;
        END;
    last ← includedCTXNull;
    FOR ctx ← (mdb+mdroot).mdctx, (ctxb+ctx).ctxchain UNTIL ctx = CTXNull
      DO
      IF (ctxb+ctx).ctxmap = target THEN RETURN [ctx];
      last ← ctx;
      ENDLOOP;
    ctx ← TableDefs.Allocate[ctxtype, SIZE[included CTXRecord]];
    (ctxb+ctx)↑ ← CTXRecord[
        sn: snNil,
        selist: ISENull,
        ctxlevel: (iCtxb+ictx).ctxlevel,
        extension: included[
          ctxchain:  includedCTXNull,
          ctxmodule: mdroot,
          ctxmap:  target,
          restricted: FALSE,
          ctxcomplete: FALSE,
          ctxclosed: FALSE,
          ctxreset: FALSE]];
    IF last = CTXNull
      THEN (mdb+mdroot).mdctx ← ctx
      ELSE (ctxb+last).ctxchain ← ctx;
    RETURN [ctx]
    END;


  UnknownModule: PUBLIC SIGNAL [HTIndex] = CODE;

  FillModule: PUBLIC PROCEDURE [sei: ISEIndex, file: HTIndex] =
    BEGIN
    mdi: MDIndex = CopierDefs.HtiToMdi[file];
    iHti: HTIndex;
    isei: ISEIndex;
    IF mdi = MDNull OR ~OpenIncludedTable[mdi]
      THEN DummyCtxSe[sei]
      ELSE
        BEGIN
          BEGIN
```

```
        iHti ← InverseMapHti[(seb+sei).htptr !MissingHti => GO TO failed];
        isei ← iBase.SearchContext[iHti, iBase.stHandle.directoryCtx];
        IF isei = SENull OR ~(iSeb+isei).public THEN GO TO failed;
        CopyCtxSeInfo[sei, isei, mdi];  (seb+sei).public ← FALSE;
        EXITS
          failed =>
            BEGIN  SIGNAL UnknownModule[(seb+sei).htptr]; DummyCtxSe[sei];
            END;
        END;
      CloseIncludedTable[];
      END;
  RETURN
  END;

DummyCtxSe: PROCEDURE [sei: ISEIndex] =
  BEGIN OPEN (seb+sei);
  idtype ← typeANY;  idinfo ← idvalue ← 0;
  extended ← public ← linkSpace ← FALSE;
  mark3 ← mark4 ← writeonce ← constant ← TRUE;
  RETURN
  END;


CopyIncludedSymbol: PUBLIC PROCEDURE [isei: SEIndex, mdi: MDIndex] RETURNS [sei: SEIndex] =
  BEGIN
  ctx: includedCTXIndex;
  hti, iHti: HTIndex;
  imdi: MDIndex;
  IF isei = SENull THEN RETURN [SENull];
  WITH (iSeb+isei) SELECT FROM
    id =>
      BEGIN
      IF ctxnum = FIRST[CTXIndex]+SIZE[simple CTXRecord] THEN RETURN [isei];
      ctx ← FindIncludedCtx[mdi, ctxnum];
      hti ← MapHti[htptr];
      sei ← SearchContext[hti, ctx];
      IF sei = SENull
        THEN
          BEGIN  imdi ← (ctxb+ctx).ctxmodule;
          IF imdi = mdi OR ~(mdb+imdi).mdshared
            THEN  sei ← CopyCtxSe[LOOPHOLE[isei, ISEIndex], hti, ctx, mdi]
            ELSE
              BEGIN
              CloseIncludedTable[];
              IF OpenIncludedTable[imdi]
                THEN
                  BEGIN  iHti ← InverseMapHti[hti];
                  isei ← iBase.SearchContext[iHti, (ctxb+ctx).ctxmap];
                  END
                ELSE [] ← OpenIncludedTable[imdi←mdi];
              sei ← CopyCtxSe[LOOPHOLE[isei, ISEIndex], hti, ctx, imdi];
              CloseIncludedTable[];
              [] ← OpenIncludedTable[mdi];
              END;
          END;
      END;
    constructor =>  sei ← CopyNonCtxSe[LOOPHOLE[isei, CSEIndex], mdi];
    ENDCASE;
  RETURN
  END;


CopyCtxSe: PROCEDURE [isei: ISEIndex, hti: HTIndex, ctx: CTXIndex, mdi: MDIndex] RETURNS [sei: ISEInd
**ex] =
  BEGIN
  sei ← makectxse[hti, ctx];  CopyCtxSeInfo[sei, isei, mdi];  RETURN
  END;

CopyCtxSeInfo: PROCEDURE [sei, isei: ISEIndex, mdi: MDIndex] =
  BEGIN
  OPEN id: (seb+sei);
  IF (iSeb+isei).ctxnum = CTXNull THEN id.ctxnum ← CTXNull;
  id.extended ← (iSeb+isei).extended;
  id.public ← (iSeb+isei).public;
  id.writeonce ← (iSeb+isei).writeonce;
  id.constant ← (iSeb+isei).constant;
```

```
    id.linkSpace ← (iSeb+isei).linkSpace;
    id.idtype ← CopyIncludedSymbol[(iSeb+isei).idtype, mdi];
    IF (iSeb+isei).idtype = typeTYPE
      THEN id.idinfo ← CopyIncludedSymbol[(iSeb+isei).idinfo, mdi]
      ELSE IF (iSeb+isei).constant AND
         (SELECT iBase.XferMode[(iSeb+isei).idtype] FROM
          procedure, program => TRUE,
          ENDCASE => FALSE)
        THEN id.idinfo ← CopyIncludedBody[(iSeb+isei).idinfo, sei, mdi]
        ELSE id.idinfo ← (iSeb+isei).idinfo;
    id.idvalue ← (iSeb+isei).idvalue;
    id.mark3 ← id.mark4 ← TRUE;
    IF id.linkSpace THEN BEGIN  id.writeonce ← TRUE;  id.idinfo ← 0  END;
    IF id.extended
      THEN SymSegDefs.EnterExtension[sei, CopyExtension[isei]];
    RETURN
    END;


  CopyExtension: PROCEDURE [iSei: ISEIndex] RETURNS [TreeDefs.TreeLink] =
    BEGIN
    OPEN TreeDefs;

    InputTree: TreeMap =
      BEGIN
      WITH link: t SELECT FROM
        literal =>  v ← InputLiteral[link];
        subtree =>  v ← CopyTree[[baseP:@iBase.tb, link:link], InputTree];
        ENDCASE => ERROR;       -- for now
      RETURN
      END;

    InputLiteral: PROCEDURE [t: literal TreeLink] RETURNS [TreeLink] =
      BEGIN
      WITH t.info SELECT FROM
        word =>  index ← LitDefs.CopyLiteral[[baseP:@iBase.ltb, index:index]];
        ENDCASE => ERROR;
      RETURN [t]
      END;

    exti: SymSegDefs.ExtIndex;
    FOR exti ← FIRST[SymSegDefs.ExtIndex], exti + SIZE[SymSegDefs.ExtRecord]
      DO
      IF (iBase.extb+exti).sei = iSei THEN EXIT;
      ENDLOOP;
    RETURN [InputTree[(iBase.extb+exti).tree]]
    END;


  CopyIncludedBody: PROCEDURE [iBti: CBTIndex, sei: ISEIndex, mdi: MDIndex] RETURNS [bti: CBTIndex] =
    BEGIN
    iCtx: CTXIndex;
    IF iBti = BTNull
      THEN  bti ← CBTNull
      ELSE
        BEGIN  OPEN TableDefs;
        iCtx ← (iBase.bb+iBti).localCtx;
        WITH body: (iBase.bb+iBti) SELECT FROM
          Outer =>
            BEGIN
            bti ← Allocate[bodytype, SIZE[Outer Callable BodyRecord]];
            (bb+LOOPHOLE[bti, OCBTIndex])↑ ← body;
            END;
          Inner =>
            BEGIN
            bti ← Allocate[bodytype, SIZE[Inner Callable BodyRecord]];
            (bb+LOOPHOLE[bti, ICBTIndex])↑ ← body;
            END;
          ENDCASE;
        (bb+bti).link ← [parent, BTNull];  (bb+bti).firstSon ← BTNull;
        (bb+bti).id ← sei;  (bb+bti).ioType ← (seb+sei).idtype;
        (bb+bti).localCtx ← IF (iBase.bb+iBti).level = 1G
                        THEN FindIncludedCtx[mdi, iCtx]
                        ELSE CTXNull;
        END;
    RETURN
```

```
    END;


  CopyNonCtxSe: PROCEDURE [isei: CSEIndex, mdi: MDIndex] RETURNS [sei: CSEIndex] =
    BEGIN
    tsei1, tsei2, tsei3: SEIndex;
    rsei1, rsei2: recordCSEIndex;
    tag: ISEIndex;
    tctx: CTXIndex;
    IF isei = SENull THEN RETURN [CSENull];
    WITH (iSeb+isei) SELECT FROM
      mode => RETURN [typeTYPE];
      basic => RETURN [isei];
      ENDCASE;
    WITH  itype: (iSeb+isei) SELECT FROM
      enumerated =>
        BEGIN
        sei ← makenonctxse[SIZE[enumerated constructor SERecord]];
        tctx ← CopyIncludedValues[itype.valuectx, mdi, sei];
        (seb+sei).typeinfo ← enumerated[
            ordered: itype.ordered,
            valuectx: tctx,
            nvalues: itype.nvalues];
        END;
      record =>
        BEGIN
        tctx ← FindIncludedCtx[mdi, itype.fieldctx];
        WITH itype SELECT FROM
          notlinked =>
            BEGIN
            sei ← makenonctxse[SIZE[notlinked record constructor SERecord]];
            (seb+sei).typeinfo ← record[
                machineDep: itype.machineDep,
                unifield: itype.unifield,
                argument: itype.argument,
                defaultFields: itype.defaultFields,
                fieldctx: tctx,
                length: itype.length,
                comparable: itype.comparable,
                privateFields: itype.privateFields,
                lengthUsed: FALSE,
                monitored: itype.monitored,
                variant: itype.variant,
                linkpart: notlinked[]];
            END;
          linked =>
            BEGIN
            sei ← makenonctxse[SIZE[linked record constructor SERecord]];
            tsei1 ← CopyIncludedSymbol[linktype, mdi];
            (seb+sei).typeinfo ← record[
                machineDep: itype.machineDep,
                unifield: itype.unifield,
                argument: itype.argument,
                defaultFields: itype.defaultFields,
                fieldctx: tctx,
                length: itype.length,
                comparable: itype.comparable,
                privateFields: itype.privateFields,
                lengthUsed: FALSE,
                monitored: itype.monitored,
                variant: itype.variant,
                linkpart: linked[linktype: tsei1]];
            END;
          ENDCASE;
        END;
      pointer =>
        BEGIN
        sei ← makenonctxse[SIZE[pointer constructor SERecord]];
        tsei1 ← CopyIncludedSymbol[itype.pointedtotype, mdi];
        (seb+sei).typeinfo ← pointer[
            pointedtotype: tsei1,
            readonly: itype.readonly,
            ordered: itype.ordered,
            basing: itype.basing,
            dereferenced: FALSE];
        END;
```

```
    array =>
      BEGIN
      sei ← makenonctxse[SIZE[array constructor SERecord]];
      tsei1 ← CopyIncludedSymbol[itype.indextype, mdi];
      tsei2 ← CopyIncludedSymbol[itype.componenttype, mdi];
      (seb+sei).typeinfo ← array[
          packed: itype.packed,
          indextype: tsei1,
          componenttype: tsei2,
          comparable: itype.comparable,
          lengthUsed: FALSE];
      END;
    arraydesc =>
      BEGIN
      sei ← makenonctxse[SIZE[arraydesc constructor SERecord]];
      tsei1 ← CopyIncludedSymbol[itype.describedType, mdi];
      (seb+sei).typeinfo ← arraydesc[describedType: tsei1];
      END;
    transfer =>
      BEGIN
      sei ← makenonctxse[SIZE[transfer constructor SERecord]];
      rsei1 ← CopyArgRecord[itype.inrecord, mdi];
      rsei2 ← CopyArgRecord[itype.outrecord, mdi];
      (seb+sei).typeinfo ← transfer[
          mode: itype.mode,
          inrecord: rsei1,
          outrecord: rsei2];
      END;
    definition =>
      BEGIN
      sei ← makenonctxse[SIZE[definition constructor SERecord]];
      tctx ← FindIncludedCtx[mdi, itype.defCtx];
      (seb+sei).typeinfo ← definition[
              nGfi: itype.nGfi,
              defCtx: tctx];
      END;
    union =>
      BEGIN
      sei ← makenonctxse[SIZE[union constructor SERecord]];
      tctx ← FindIncludedCtx[mdi, itype.casectx];
      tag ← CopyCtxSe[itype.tagsei, MapHti[(iSeb+itype.tagsei).htptr], CTXNull, mdi];
      (seb+sei).typeinfo ← union[
          casectx: tctx,
          overlayed: itype.overlayed,
          controlled: itype.controlled,
          tagsei: tag,
          equalLengths: itype.equalLengths];
      END;
    relative =>
      BEGIN
      sei ← makenonctxse[SIZE[relative constructor SERecord]];
      tsei1 ← CopyIncludedSymbol[itype.baseType, mdi];
      tsei2 ← CopyIncludedSymbol[itype.offsetType, mdi];
      tsei3 ← IF itype.resultType = itype.offsetType
                THEN tsei2
                ELSE CopyIncludedSymbol[itype.resultType, mdi];
      (seb+sei).typeinfo ← relative[
              baseType: tsei1,
              offsetType: tsei2,
              resultType: tsei3];
      END;
    subrange =>
      BEGIN
      sei ← makenonctxse[SIZE[subrange constructor SERecord]];
      tsei1 ← CopyIncludedSymbol[itype.rangetype, mdi];
      (seb+sei).typeinfo ← subrange[
          filled: itype.filled,
          empty: itype.empty,
          flexible: itype.flexible,
          rangetype: tsei1,
          origin: itype.origin,
          range: itype.range];
      END;
    long =>
      BEGIN
      sei ← makenonctxse[SIZE[long constructor SERecord]];
```

```
        tsei1 ← CopyIncludedSymbol[itype.rangetype, mdi];
        (seb+sei).typeinfo ← long[rangetype: tsei1];
        END;
      real =>
        BEGIN
        sei ← makenonctxse[SIZE[real constructor SERecord]];
        tsei1 ← CopyIncludedSymbol[itype.rangetype, mdi];
        (seb+sei).typeinfo ← real[rangetype: tsei1];
        END;
      ENDCASE => ERROR;
    (seb+sei).mark3 ← (seb+sei).mark4 ← TRUE;  RETURN
    END;


  CopyArgRecord: PROCEDURE [irsei: recordCSEIndex, mdi: MDIndex] RETURNS [rsei: recordCSEIndex] =
    BEGIN
    ctx, ictx: CTXIndex;
    sei, isei, seChain: ISEIndex;
    IF irsei = SENull
      THEN rsei ← recordCSENull
      ELSE
        BEGIN
        rsei ← LOOPHOLE[makenonctxse[SIZE[notlinked record constructor SERecord]]];
         ictx ← (iSeb+irsei).fieldctx;
        ctx ← makenewctx[(iCtxb+ictx).ctxlevel];
        seChain ← makeSEChain[ctx, iBase.CtxEntries[ictx], FALSE];
        (ctxb+ctx).selist ← seChain;
        FOR isei ← (iCtxb+ictx).selist, iBase.NextSe[isei] UNTIL isei = SENull
          DO
          sei ← seChain;  seChain ← NextSe[seChain];
          (seb+sei).htptr ← MapHti[(iSeb+isei).htptr];
          CopyCtxSeInfo[sei, isei, mdi];
          ENDLOOP;
        (seb+rsei)↑ ← SERecord[
                mark3: TRUE,
                mark4: TRUE,
                sebody: constructor[
                  record[
                    machineDep: FALSE,
                    unifield: (iSeb+irsei).unifield,
                    argument: TRUE,
                    defaultFields: (iSeb+irsei).defaultFields,
                    fieldctx: ctx,
                    length: (iSeb+irsei).length,
                    comparable: (iSeb+irsei).comparable,
                    privateFields: (iSeb+irsei).privateFields,
                    lengthUsed: FALSE,
                    monitored: FALSE,
                    variant: FALSE,
                    linkpart: notlinked[]]]];
        END;
    RETURN
    END;


  CopyIncludedValues: PROCEDURE [ictx: CTXIndex, mdi: MDIndex, type: SEIndex] RETURNS [ctx: includedCTX
**Index] =
    BEGIN
    isei, sei, seChain: ISEIndex;
    ctx ← FindIncludedCtx[mdi, ictx];
    isei ← (iCtxb+ictx).selist;
    IF isei # SENull AND (iSeb+(iSeb+isei).idtype).setag # id
      THEN
        BEGIN
        seChain ← makeSEChain[ctx, iBase.CtxEntries[ictx], FALSE];
        (ctxb+ctx).selist ← seChain;
        (ctxb+ctx).ctxclosed ← (ctxb+ctx).ctxreset ← TRUE;
        UNTIL isei = SENull
          DO
          sei ← seChain;  seChain ← NextSe[seChain];
          (seb+sei).htptr ← MapHti[(iSeb+isei).htptr];
          (seb+sei).extended ← (seb+sei).linkSpace ← FALSE;
          (seb+sei).writeonce ← (seb+sei).constant ← TRUE;
          (seb+sei).public ← (iSeb+isei).public;
          (seb+sei).idtype ← type;  (seb+sei).idinfo ← 0;
          (seb+sei).idvalue ← (iSeb+isei).idvalue;
```

```
        (seb+sei).mark3 ← (seb+sei).mark4 ← TRUE;
        isei ← iBase.NextSe[isei];
        ENDLOOP;
      (ctxb+ctx).ctxcomplete ← TRUE;
      END;
  RETURN
  END;


-- included module accounting


  resetctx: PROCEDURE [ctx: includedCTXIndex] =
    BEGIN
    IF ~(ctxb+ctx).ctxreset
      THEN
        BEGIN  resetctxlist[ctx];
        (ctxb+ctx).ctxclosed ← (ctxb+ctx).ctxreset ← TRUE;
        END;
    RETURN
    END;

  ResetIncludeContexts: PUBLIC PROCEDURE =
    BEGIN
    mdi: MDIndex;
    limit: MDIndex = LOOPHOLE[TableDefs.TableBounds[mdtype].size];
    ctx: includedCTXIndex;
    FOR mdi ← FIRST[MDIndex], mdi + SIZE[MDRecord] UNTIL mdi = limit
      DO
      FOR ctx ← (mdb+mdi).mdctx, (ctxb+ctx).ctxchain UNTIL ctx = CTXNull
        DO
        resetctx[ctx];
        ENDLOOP;
      ENDLOOP;
    RETURN
    END;


  TableRelocated: PUBLIC SIGNAL = CODE;

  OpenIncludedTable: PUBLIC PROCEDURE [mdi: MDIndex] RETURNS [success: BOOLEAN] =
    BEGIN
    base: SymbolTableDefs.SymbolTableBase = CopierDefs.GetSymbolTable[mdi];
    IF success ← (base # NIL)
      THEN
        BEGIN
        iBase ← base;
        iBase.notifier ← SetIncludedBases;
        SetIncludedBases[iBase];
        END;
    RETURN
    END;

  SetIncludedBases: PROCEDURE [base: SymbolTableDefs.SymbolTableBase] =
    BEGIN
    IF base # iBase THEN ERROR;
    iHt ← base.ht;  iSeb ← base.seb;  iCtxb ← base.ctxb;
    SIGNAL TableRelocated;
    RETURN
    END;

  CloseIncludedTable: PUBLIC PROCEDURE =
    BEGIN
    iBase.notifier ← iBase.NullNotifier;
    CopierDefs.FreeSymbolTable[iBase];
    RETURN
    END;

  END...
```