```
-- file Pass3I.Mesa
-- last modified by Satterthwaite, July 16, 1978  10:00 AM

DIRECTORY
  ComData: FROM "comdata"
    USING [importCtx, mainCtx, seAnon, textIndex, typeBOOLEAN, xref],
  CompilerDefs: FROM "compilerdefs" USING [AppendXrefWords],
  CopierDefs: FROM "copierdefs"
    USING [CompleteContext, Delink, SearchFileCtx],
  ErrorDefs: FROM "errordefs"
    USING [error, errorhti, errorsei, WarningSei, errortree],
  P3Defs: FROM "p3defs"
    USING [
      P3Mark,
      BumpCount, Exp, LongPath, OperandType, ResolveReference, Rhs,
      RPop, RPush, RType, TargetType, UnionField, VariantUnionType, VoidExp,
      CheckExprLoop, CheckTypeLoop, LogExprLoop],
  Pass3: FROM "pass3" USING [implicitTree, implicitType],
  SymDefs: FROM "symdefs"
    USING [setype, ctxtype, mdtype, bodytype,
      HTIndex, SEIndex, ISEIndex, CSEIndex, recordCSEIndex,
      CTXIndex, includedCTXIndex, BTIndex,
      HTNull, SENull, ISENull, recordCSENull, CTXNull, BTNull,
      1G, 1Z, typeANY],
  SymTabDefs: FROM "symtabdefs"
    USING [
      ConstantId, firstvisiblese, NextSe, NormalType, SearchContext,
      setselink, UnderType, visiblectxentries, XferMode],
  SystemDefs: FROM "systemdefs"
    USING [
      AllocateHeapNode, AllocateSegment, FreeHeapNode, FreeSegment,
      SegmentSize],
  TableDefs: FROM "tabledefs"
    USING [TableBase, TableNotifier],
  TreeDefs: FROM "treedefs"
    USING [treetype,
      TreeIndex, TreeLink, TreeMap, TreeScan,
      empty, nullid,
      CopyTree, freenode, freetree, GetNode, mlpop, mlpush, pushtree,
      scanlist, setattr, setinfo, setshared, shared, testtree,
      updatelist, UpdateTree],
  XrefJournalDefs: FROM "xrefjournaldefs"
    USING [RefData, ReferenceNumber, RefType];

Pass3I: PROGRAM
    IMPORTS
        CompilerDefs, CopierDefs, ErrorDefs, P3Defs, SymTabDefs, SystemDefs,
        TreeDefs,
        dataPtr: ComData, passPtr: Pass3
    EXPORTS P3Defs =
  BEGIN
  OPEN SymTabDefs, P3Defs, SymDefs, TreeDefs;

  UndeclaredIdentifier: PUBLIC SIGNAL [HTIndex] RETURNS [ISEIndex] = CODE;
  AmbiguousIdentifier: PUBLIC SIGNAL [ISEIndex] RETURNS [ISEIndex] = CODE;

 -- cross reference interface

  seqNumber: XrefJournalDefs.ReferenceNumber;
  RefType: TYPE = XrefJournalDefs.RefType;

  RecordReference: PUBLIC PROCEDURE [sei: ISEIndex, type: RefType] =
    BEGIN  OPEN XrefJournalDefs;
    ref: RefData ← [
        symbol: sei,
        whereReferenced: dataPtr.textIndex,
        refTag: type,
        sequenceNumber: (seqNumber←seqNumber+1)];
    CompilerDefs.AppendXrefWords[@ref, SIZE[RefData]];  RETURN
    END;


 -- tables defining the current symbol table

  tb: TableDefs.TableBase;              -- tree base
  seb: TableDefs.TableBase;             -- se table
```

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

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


-- identifier look-up

  Id: PUBLIC PROCEDURE [hti: HTIndex] RETURNS [val: TreeLink] =
    BEGIN
    sei: ISEIndex;
    type: CSEIndex;
    baseV: TreeLink;
    indirect, const: BOOLEAN;
    [sei, baseV, indirect] ← FindSe[hti
      !UndeclaredIdentifier =>
        BEGIN
        IF hti # HTNull THEN ErrorDefs.errorhti[unknownId, hti];
        RESUME [dataPtr.seAnon]
        END;
       AmbiguousIdentifier =>
        BEGIN
        ErrorDefs.errorhti[ambiguousId, hti];  RESUME[dataPtr.seAnon]
        END];
    IF sei # SENull
      THEN
        BEGIN
        IF (ctxb+(seb+sei).ctxnum).ctxType = included
          THEN  BEGIN  IF baseV = empty THEN  CheckUnbased[sei]  END
          ELSE
            IF ~(seb+sei).mark3
              THEN ResolveId[sei
                !LogExprLoop =>
                  ErrorDefs.errorsei[circularValue, sei]];
        BumpCount[sei];
        IF dataPtr.xref THEN RecordReference[sei, mention];
        type ← UnderType[(seb+sei).idtype];  const ← ConstantId[sei];
        RPush[type, const];
        val ← TreeLink[symbol[index: sei]];
        IF baseV # empty AND ~const
         AND (ctxb+(seb+sei).ctxnum).ctxType # imported
          THEN
            BEGIN  CountTreeIds[baseV];
            mlpush[baseV];  mlpush[val];
            IF indirect
              THEN
                BEGIN
                pushtree[dot, 2];
                setattr[1, (seb+OperandType[baseV]).typetag = long];
                END
              ELSE
                BEGIN
                pushtree[dollar, 2];  setattr[1, LongPath[baseV]];
                END;
            setinfo[RType[]];
            val ← mlpop[];
            END;
        END
      ELSE
        BEGIN
        CountTreeIds[baseV];  mlpush[baseV];
        type ← OperandType[baseV];
        IF indirect
          THEN
            BEGIN
            pushtree[uparrow, 1];  setattr[1, (seb+type).typetag = long];
            type ← NormalType[type];
            type ← WITH (seb+type) SELECT FROM
```

```
            pointer => UnderType[pointedtotype],
            ENDCASE => typeANY;
          setinfo[type];
          END;
      val ← mlpop[];  RPush[type, FALSE];
      END;
  RETURN
  END;


FieldId: PUBLIC PROCEDURE [hti: HTIndex, type: recordCSEIndex]
    RETURNS [n: CARDINAL, sei: ISEIndex] =
  BEGIN
  [n, sei] ← SearchRecord[hti, type];
  IF n # 0
    THEN
      BEGIN
      IF ~(seb+sei).mark3
        THEN ResolveId[sei
          !LogExprLoop => ErrorDefs.errorsei[circularValue, sei]];
      BumpCount[sei];
      IF dataPtr.xref THEN RecordReference[sei, mention];
      END;
  RETURN
  END;

DefinedId: PUBLIC PROCEDURE [hti: HTIndex, type: CSEIndex]
    RETURNS [found: BOOLEAN, sei: ISEIndex] =
  BEGIN
  WITH (seb+type) SELECT FROM
    definition =>
      BEGIN
      [found, sei] ← SearchCtxList[hti, defCtx];
      IF found
        THEN
          BEGIN
          IF ~(seb+sei).mark3
            THEN ResolveId[sei
              !LogExprLoop =>
                ErrorDefs.errorsei[circularValue, sei]];
          CheckUnbased[sei];  BumpCount[sei];
          IF dataPtr.xref THEN RecordReference[sei, mention];
          END;
      END;
    ENDCASE => BEGIN  found ← FALSE;  sei ← ISENull  END;
  RETURN
  END;


CompleteRecord: PUBLIC PROCEDURE [rSei: recordCSEIndex] =
  BEGIN
  ctx: CTXIndex = (seb+rSei).fieldctx;
  WITH (ctxb+ctx) SELECT FROM
    simple => NULL;
    included =>
      IF ctxlevel = 1Z
        THEN CopierDefs.CompleteContext[LOOPHOLE[ctx, includedCTXIndex], FALSE];
    ENDCASE;
  RETURN
  END;


-- keyed-list matching

ArrangeKeys: PUBLIC PROCEDURE
    [expList: TreeLink, ctx: CTXIndex, omittedKey: PROCEDURE [ISEIndex] RETURNS [TreeLink]]
    RETURNS [nFields: CARDINAL] =
  BEGIN
  Pair: TYPE = RECORD[
    key: ISEIndex,
    defined: BOOLEAN,
    attr: TreeLink];
  i: CARDINAL;
  aList: DESCRIPTOR FOR ARRAY OF Pair;
  sei: ISEIndex;
```

```
    KeyItem: TreeMap =
      BEGIN
      node: TreeIndex;
      hti: HTIndex;
      i: CARDINAL;
      WITH t SELECT FROM
        subtree =>
          BEGIN  node ← index;
          WITH (tb+node).son1 SELECT FROM
            hash =>
              BEGIN  hti ← index;
              FOR i IN [0 .. nFields)
                DO
                IF (seb+aList[i].key).htptr = hti
                  THEN
                    BEGIN
                    IF ~aList[i].defined
                      THEN
                        BEGIN  aList[i].attr ← (tb+node).son2;
                        (tb+node).son2 ← empty;
                        aList[i].defined ← TRUE;
                        END
                      ELSE
                        BEGIN  ErrorDefs.errorhti[duplicateKey, hti];
                        (tb+node).son2 ← VoidExp[(tb+node).son2];
                        END;
                    EXIT
                    END;
                REPEAT
                  FINISHED =>
                    BEGIN  ErrorDefs.errorhti[unknownKey, hti];
                    (tb+node).son2 ← VoidExp[(tb+node).son2];
                    END;
                ENDLOOP;
              freenode[node];
              END;
            ENDCASE =>  ERROR;
          END;
        ENDCASE =>  ERROR;
      RETURN [empty]
      END;

    nFields ← visiblectxentries[ctx];
    aList ← DESCRIPTOR[
        SystemDefs.AllocateHeapNode[nFields*SIZE[Pair]],
        nFields];
    i ← 0;
    FOR sei ← firstvisiblese[ctx], NextSe[sei] UNTIL sei = SENull
      DO
      aList[i] ← Pair[key:sei, defined:FALSE, attr:empty];  i ← i+1;
      ENDLOOP;
    expList ← freetree[updatelist[expList, KeyItem]];
    FOR i IN [0 .. nFields)
      DO
      mlpush[IF aList[i].defined
        THEN aList[i].attr
        ELSE omittedKey[aList[i].key]];
      ENDLOOP;
    SystemDefs.FreeHeapNode[BASE[aList]];
    RETURN
    END;


  -- auxiliary service routines

   ResolveId: PROCEDURE [sei: ISEIndex] =
     BEGIN
     declNode: TreeIndex;
     declNode ← (seb+sei).idvalue;
     IF (tb+declNode).mark # P3Mark
       THEN ResolveReference[sei]
       ELSE
         IF SIGNAL CheckExprLoop[declNode]
           THEN SIGNAL LogExprLoop[declNode];
     RETURN
     END;
```

```
CheckUnbased: PROCEDURE [sei: ISEIndex] =
  BEGIN
  IF ~ConstantId[sei]
    THEN
      WITH (ctxb+(seb+sei).ctxnum) SELECT FROM
        included =>
          IF (ctxb+(seb+sei).ctxnum).ctxType # imported
            THEN  ErrorDefs.errorsei[notImported, sei]
            ELSE  ErrorDefs.errorsei[missingBase, sei];
        ENDCASE;
  RETURN
  END;


CountTreeIds: PUBLIC TreeScan =
  BEGIN  -- traverses the tree, incrementing reference counts for ids

  CountIds: TreeMap =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        BumpCount[sei];
        IF dataPtr.xref THEN RecordReference[sei, implicit];
        END;
      subtree =>  [] ← UpdateTree[t, CountIds];
      ENDCASE =>  NULL;
    RETURN [t]
    END;

  [] ← CountIds[t];  RETURN
  END;


LambdaApply: PUBLIC PROCEDURE [t: TreeLink, formal, actual: ISEIndex] RETURNS [TreeLink] =
  BEGIN

  Substitute: TreeMap =
    BEGIN
    sei: ISEIndex;
    WITH t SELECT FROM
      symbol =>
        BEGIN  sei ← index;
        IF sei = formal THEN sei ← actual;
        BumpCount[sei];
        IF dataPtr.xref THEN RecordReference[sei, implicit];
        v ← [symbol[index: sei]];
        END;
      subtree =>
        IF shared[t]
          THEN  BEGIN  CountTreeIds[t];  v ← t  END
          ELSE  v ← CopyTree[[baseP:@tb, link:t], Substitute];
      ENDCASE =>  v ← t;
    RETURN
    END;

  RETURN [Substitute[t]];
  END;


-- context stack management

ContextEntry: TYPE = RECORD[
  base: TreeLink,              -- the basing expr (empty if none)
  indirect: BOOLEAN,  -- true iff basing expr is pointer
  info: SELECT tag: * FROM
    list => [ctx: CTXIndex],          -- a single context
    record => [rSei: recordCSEIndex], -- a group of contexts
    hash => [ctxHti: HTIndex],                -- a single identifier
    ENDCASE];

ContextStack: TYPE = DESCRIPTOR FOR ARRAY OF ContextEntry;
```

```
ctxStack: ContextStack;
ctxIndex: INTEGER;
ContextIncr: CARDINAL = 25;

MakeStack: PROCEDURE [size: CARDINAL] RETURNS [ContextStack] =
  BEGIN
  OPEN SystemDefs;
  base: POINTER = AllocateSegment[size*SIZE[ContextEntry]];
  RETURN [DESCRIPTOR[base, SegmentSize[base]/SIZE[ContextEntry]]]
  END;

FreeStack: PROCEDURE [s: ContextStack] =
  BEGIN
  OPEN SystemDefs;
  IF LENGTH [s] > 0 THEN FreeSegment[BASE[s]];
  RETURN
  END;

ExpandStack: PROCEDURE =
  BEGIN
  i: CARDINAL;
  oldstack: ContextStack ← ctxStack;
  ctxStack ← MakeStack[LENGTH[oldstack]+ContextIncr];
  FOR i IN [0 .. LENGTH[oldstack]) DO ctxStack[i] ← oldstack[i] ENDLOOP;
  FreeStack[oldstack];
  RETURN
  END;



PushCtx: PUBLIC PROCEDURE [ctx: CTXIndex] =
  BEGIN
  IF (ctxIndex ← ctxIndex+1) >= LENGTH[ctxStack] THEN  ExpandStack[];
  ctxStack[ctxIndex] ← ContextEntry[
      base: empty,
      indirect: FALSE,
      info: list[ctx: ctx]];
  RETURN
  END;

ReplaceCtx: PUBLIC PROCEDURE [old, new: CTXIndex] =
  BEGIN
  i: INTEGER;
  FOR i IN [0..ctxIndex] DO
    WITH ctxStack[i] SELECT FROM
      list => IF ctx = old THEN ctx ← new;
      ENDCASE;
    ENDLOOP;
  RETURN
  END;

SetCtxBase: PROCEDURE [base: TreeLink, indirect: BOOLEAN] =
  BEGIN
  IF ~testtree[base, openexp] THEN ERROR;
  ctxStack[ctxIndex].base ← base;  ctxStack[ctxIndex].indirect ← indirect;
  RETURN
  END;

PushRecordCtx: PROCEDURE [rSei: recordCSEIndex, base: TreeLink, indirect: BOOLEAN] =
  BEGIN
  IF (ctxIndex ← ctxIndex+1) >= LENGTH[ctxStack] THEN  ExpandStack[];
  ctxStack[ctxIndex] ← ContextEntry[
      base: base,
      indirect: indirect,
      info: record[rSei: rSei]];
  END;

UpdateRecordCtx: PROCEDURE [type: recordCSEIndex] =
  BEGIN
  WITH ctxStack[ctxIndex] SELECT FROM
    record => rSei ← type;
    ENDCASE => ERROR;
  RETURN
  END;

PushHtCtx: PROCEDURE [hti: HTIndex, base: TreeLink, indirect: BOOLEAN] =
```

```
    BEGIN
    IF (ctxIndex ← ctxIndex+1) >= LENGTH[ctxStack] THEN  ExpandStack[];
    ctxStack[ctxIndex] ← ContextEntry[
        base: base,
        indirect: indirect,
        info: hash[ctxHti: hti]];
    RETURN
    END;

  PopCtx: PUBLIC PROCEDURE =
    BEGIN  ctxIndex ← ctxIndex-1;  RETURN
    END;

  TopCtx: PUBLIC PROCEDURE RETURNS [CTXIndex] =
    BEGIN
    WITH ctxStack[ctxIndex] SELECT FROM
      list =>  RETURN [ctx];
      ENDCASE =>  ERROR;
    END;


-- primary lookup

  FindSe: PUBLIC PROCEDURE [hti: HTIndex] RETURNS [ISEIndex, TreeLink, BOOLEAN] =
    BEGIN
    i: INTEGER;
    found: BOOLEAN;
    nHits: CARDINAL;
    sei: ISEIndex;
    FOR i DECREASING IN [0 .. ctxIndex]
      DO
      WITH ctxStack[i] SELECT FROM
        list =>
          BEGIN
          [found, sei] ← SearchCtxList[hti, ctx];
          IF found THEN GO TO Found;
          END;
        record =>
          BEGIN
          [nHits, sei] ← SearchRecord[hti, rSei];
          IF nHits # 0
            THEN
              BEGIN
              IF nHits # 1 THEN sei ← SIGNAL AmbiguousIdentifier[sei];
              GO TO Found;
              END;
          END;
        hash =>
          IF hti = ctxHti  THEN  BEGIN  sei ← ISENull;  GO TO Found  END;
        ENDCASE;
      REPEAT
        Found =>  RETURN [sei, ctxStack[i].base, ctxStack[i].indirect];
        FINISHED =>
          BEGIN   sei ← SIGNAL UndeclaredIdentifier[hti];
          RETURN [sei, empty, FALSE]
          END;
      ENDLOOP;
    END;


  SearchCtxList: PUBLIC PROCEDURE [hti: HTIndex, ctx: CTXIndex]
      RETURNS [found: BOOLEAN, sei: ISEIndex] =
    BEGIN
    IF ctx = CTXNull THEN RETURN [FALSE, ISENull];
    WITH c: (ctxb+ctx) SELECT FROM
      included =>
        IF c.restricted
          THEN
            BEGIN
            sei ← SearchRestrictedCtx[hti, LOOPHOLE[ctx]];
            found ← (sei # SENull);
            IF found AND ~(seb+sei).public AND ~(mdb+c.ctxmodule).mdshared
             AND sei # dataPtr.seAnon
              THEN ErrorDefs.errorhti[noAccess, hti];
            END
          ELSE
```

```
          BEGIN
          sei ← SearchContext[hti, ctx];
          IF sei # SENull
            THEN  found ← (seb+sei).public OR (mdb+c.ctxmodule).mdshared
            ELSE IF ~c.ctxclosed AND ~c.ctxreset
              THEN
                [found, sei] ← CopierDefs.SearchFileCtx[hti, LOOPHOLE[ctx]]
              ELSE found ← FALSE;
          END;
      imported =>
        BEGIN
        sei ← SearchContext[hti, ctx];
        IF sei = SENull
         AND (mdb+(ctxb+c.includeLink).ctxmodule).mdExported
         AND (ctxb+c.includeLink).ctxlevel = 1G
          THEN
            BEGIN
            sei ← SearchContext[hti, dataPtr.mainCtx];
            IF sei # SENull AND (~CheckExport[sei] OR NamedImport[ctx])
              THEN sei ← ISENull;
            END;
        IF sei # SENull
          THEN  found ← TRUE
          ELSE
            BEGIN
            [found, sei] ← SearchCtxList[hti, c.includeLink];
            IF found AND ~(seb+sei).constant AND sei # dataPtr.seAnon
              THEN
                BEGIN  CopierDefs.Delink[sei];  (seb+sei).ctxnum ← ctx;
                setselink[sei, (ctxb+ctx).selist];  (ctxb+ctx).selist ← sei;
                END;
            END;
        END;
      ENDCASE =>
        BEGIN  sei ← SearchContext[hti, ctx];  found ← (sei # SENull)  END;
    RETURN
    END;


  CheckExport: PROCEDURE [sei: ISEIndex] RETURNS [BOOLEAN] =
    BEGIN
    declNode: TreeIndex;
    loop: BOOLEAN;
    IF ~(seb+sei).public THEN RETURN [FALSE];
    IF ~(seb+sei).mark3
      THEN
        BEGIN
        declNode ← (seb+sei).idvalue;
        IF (tb+declNode).mark = P3Mark THEN RETURN [FALSE];
        loop ← FALSE;
        ResolveId[sei
          !CheckTypeLoop, CheckExprLoop =>
            BEGIN  loop ← TRUE;  RESUME [FALSE]  END];
        IF loop THEN RETURN [FALSE];
        END;
    RETURN [SELECT XferMode[(seb+sei).idtype] FROM
        procedure, signal, error, program => (seb+sei).constant,
        ENDCASE => FALSE]
    END;

  NamedImport: PROCEDURE [ctx: CTXIndex] RETURNS [BOOLEAN] =
    BEGIN
    sei: ISEIndex;
    type: CSEIndex;
    IF dataPtr.importCtx # CTXNull THEN
      FOR sei ← (ctxb+dataPtr.importCtx).selist, NextSe[sei] UNTIL sei = SENull
        DO
        type ← UnderType[(seb+sei).idtype];
        WITH (seb+type) SELECT FROM
          definition =>
            IF defCtx = ctx
              THEN RETURN [(tb+LOOPHOLE[(seb+sei).idvalue, TreeIndex]).attr1];
          ENDCASE;
        ENDLOOP;
    RETURN [TRUE]
    END;
```

```
-- searching records

 SearchRecordSegment: PROCEDURE
      [hti: HTIndex, rSei: recordCSEIndex, suffixed: BOOLEAN]
    RETURNS [nHits: CARDINAL, sei: ISEIndex] =
  BEGIN
  tSei: CSEIndex;
  found: BOOLEAN;
  n: CARDINAL;
  match: ISEIndex;
  [found, sei] ← SearchCtxList[hti, (seb+rSei).fieldctx];
  nHits ← IF found THEN 1 ELSE 0;
  IF (seb+rSei).variant
    THEN
      BEGIN
      tSei ← UnderType[(seb+UnionField[rSei]).idtype];
      WITH (seb+tSei) SELECT FROM
        union =>
          BEGIN
          IF ~suffixed AND ~controlled AND overlayed
            THEN
              BEGIN
              [n, match] ← SearchOverlays[hti, casectx];
              IF ~found THEN sei ← match;
              nHits ← nHits + n;
              END;
          IF controlled AND (seb+tagsei).htptr = hti
            THEN  BEGIN  sei ← tagsei;  nHits ← nHits + 1  END;
          END;
        ENDCASE =>  NULL;
      END;
  RETURN
  END;

 SearchOverlays: PROCEDURE [hti: HTIndex, ctx: CTXIndex]
    RETURNS [nHits: CARDINAL, sei: ISEIndex] =
  BEGIN
  vSei: ISEIndex;
  rSei: SEIndex;
  n: CARDINAL;
  match: ISEIndex;
  WITH (ctxb+ctx) SELECT FROM
    included => CopierDefs.CompleteContext[LOOPHOLE[ctx], FALSE];
    ENDCASE;
  nHits ← 0;  sei ← ISENull;
  FOR vSei ← (ctxb+ctx).selist, NextSe[vSei] UNTIL vSei = SENull
    DO
    rSei ← (seb+vSei).idinfo;
    WITH r: (seb+rSei) SELECT FROM
      id => NULL;
      constructor =>
        WITH r SELECT FROM
          record =>
            BEGIN
            [n, match] ← SearchRecordSegment[hti, LOOPHOLE[rSei], FALSE];
            IF nHits = 0 THEN sei ← match;
            nHits ← nHits + n;
            END;
        ENDCASE => ERROR;
    ENDCASE;
    ENDLOOP;
  RETURN
  END;

 SearchRecord: PROCEDURE [hti: HTIndex, type: recordCSEIndex]
    RETURNS [nHits: CARDINAL, sei: ISEIndex] =
  BEGIN
  rSei: recordCSEIndex;
  suffixed: BOOLEAN;
  rSei ← type;  suffixed ← FALSE;
  UNTIL rSei = SENull
    DO
    [nHits, sei] ← SearchRecordSegment[hti, rSei, suffixed];
    IF nHits # 0 THEN RETURN;
```

```
    rSei ← WITH (seb+rSei) SELECT FROM
      linked => LOOPHOLE[UnderType[linktype]],
      ENDCASE => recordCSENull;
    suffixed ← TRUE;
    ENDLOOP;
  RETURN [0, ISENull]
  END;


-- management of restricted contexts

 CtxRestriction: TYPE = RECORD [ctx: includedCTXIndex, list: TreeLink];

 ctxIdTable: DESCRIPTOR FOR ARRAY OF CtxRestriction;
 ctxIdTableLimit: CARDINAL;


 GetDirectoryIds: PUBLIC TreeScan =
   BEGIN
   node: TreeIndex = GetNode[t];
   saveIndex: CARDINAL = dataPtr.textIndex;
   dataPtr.textIndex ← (tb+node).info;
   WITH (tb+node).son1 SELECT FROM
     symbol =>
       BEGIN
       sei: ISEIndex = index;
       type: CSEIndex = UnderType[(seb+sei).idtype];
       bti: BTIndex;
       WITH (seb+type) SELECT FROM
         definition => (tb+node).son3 ← IncludedIds[defCtx, (tb+node).son3];
         transfer =>
           IF (bti ← (seb+sei).idinfo) # BTNull
             THEN (tb+node).son3 ← IncludedIds[(bb+bti).localCtx, (tb+node).son3];
         ENDCASE => NULL;
       END;
     ENDCASE => ERROR;
   dataPtr.textIndex ← saveIndex;  RETURN
   END;


 IncludedIds: PROCEDURE [ctx: CTXIndex, list: TreeLink] RETURNS [val: TreeLink] =
   BEGIN
   iCtx: includedCTXIndex;

   IncludedId: TreeMap =
     BEGIN
     WITH t SELECT FROM
       hash =>
         BEGIN
         hti: HTIndex = index;
         sei: ISEIndex;
         found, duplicate: BOOLEAN;

         CheckDuplicate: TreeScan =
           BEGIN
           WITH t SELECT FROM
             symbol => IF index = sei THEN duplicate ← TRUE;
             ENDCASE;
           RETURN
           END;

         sei ← SearchContext[hti, ctx];
         IF sei = SENull
           THEN  [found, sei] ← CopierDefs.SearchFileCtx[hti, iCtx]
           ELSE
             BEGIN
             found ← TRUE; duplicate ← FALSE; scanlist[list, CheckDuplicate];
             IF duplicate THEN ErrorDefs.errorhti[duplicateId, hti]
             END;
         IF found
           THEN
             BEGIN (seb+sei).ctxnum ← CTXNull; v ← [symbol[index: sei]] END
           ELSE
             BEGIN ErrorDefs.errorhti[unknownId, hti]; v ← t END;
         END;
       ENDCASE => ERROR;
```

```
    RETURN
    END;

  SELECT (ctxb+ctx).ctxType FROM
    included =>
      BEGIN  iCtx ← LOOPHOLE[ctx];
      IF ((ctxb+iCtx).restricted ← list # empty)
        THEN
          BEGIN
          (mdb+(ctxb+iCtx).ctxmodule).mdshared ← TRUE;
          val ← updatelist[list, IncludedId];
          (mdb+(ctxb+iCtx).ctxmodule).mdshared ← FALSE;
          ctxIdTable[ctxIdTableLimit] ← CtxRestriction[iCtx, val];
          ctxIdTableLimit ← ctxIdTableLimit + 1;
          END
        ELSE  val ← empty;
      END;
    ENDCASE;
  RETURN
  END;


CheckDirectoryIds: PUBLIC TreeScan =
  BEGIN

  CheckId: TreeScan =
    BEGIN
    WITH t SELECT FROM
      symbol =>
        IF (seb+index).ctxnum = CTXNull
          THEN ErrorDefs.WarningSei[unusedId, index];
      ENDCASE;
    RETURN
    END;

  node: TreeIndex = GetNode[t];
  saveIndex: CARDINAL = dataPtr.textIndex;
  dataPtr.textIndex ← (tb+node).info;
  scanlist[(tb+node).son3, CheckId];
  dataPtr.textIndex ← saveIndex;  RETURN
  END;


SearchRestrictedCtx: PROCEDURE [hti: HTIndex, ctx: includedCTXIndex]
    RETURNS [sei: ISEIndex] =
  BEGIN

  TestId: TreeScan =
    BEGIN
    WITH t SELECT FROM
      hash =>  IF index = hti THEN  sei ← dataPtr.seAnon;
      symbol =>
        IF (seb+index).htptr = hti
          THEN
            BEGIN
            sei ← index;
            SELECT (seb+sei).ctxnum FROM
              CTXNull =>  (seb+sei).ctxnum ← ctx;
              ctx =>  NULL;
              ENDCASE =>  [ , sei] ← CopierDefs.SearchFileCtx[hti, ctx];
            END;
      ENDCASE;
    RETURN
    END;

  i: CARDINAL;
  FOR i IN [0 .. ctxIdTableLimit)
    DO
    IF ctxIdTable[i].ctx = ctx THEN EXIT;
    REPEAT
      FINISHED => ERROR;
    ENDLOOP;
  sei ← ISENull;  scanlist[ctxIdTable[i].list, TestId];  RETURN
  END;
```

```
CheckDisjoint: PUBLIC PROCEDURE [ctx1, ctx2: CTXIndex] =
  BEGIN
  sei: ISEIndex;
  hti: HTIndex;
  saveIndex: CARDINAL = dataPtr.textIndex;
  IF ctx1 # CTXNull AND ctx2 # CTXNull
    THEN
      FOR sei ← (ctxb+ctx2).selist, NextSe[sei] UNTIL sei = SENull
        DO
        hti ← (seb+sei).htptr;
        IF hti # HTNull AND SearchContext[hti, ctx1] # SENull
          THEN
            BEGIN
            IF ~(seb+sei).mark3
              THEN dataPtr.textIndex ←
                    (tb+LOOPHOLE[(seb+sei).idvalue, TreeIndex]).info;
            ErrorDefs.errorhti[duplicateId, hti];
            END;
      ENDLOOP;
  dataPtr.textIndex ← saveIndex;  RETURN
  END;


-- basing management

BaseTree: PROCEDURE [t: TreeLink, type: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  mlpush[t];  pushtree[openexp, 1];  setinfo[type];  setattr[1, FALSE];
  val ← mlpop[];  setshared[val, TRUE];  RETURN
  END;

OpenBase: PUBLIC PROCEDURE [t: TreeLink, hti: HTIndex] RETURNS [v: TreeLink] =
  BEGIN
  type, vType: CSEIndex;

  OpenRecord: PROCEDURE [indirect: BOOLEAN] =
    BEGIN
    WITH (seb+type) SELECT FROM
      record =>
        BEGIN
        v ← BaseTree[v, vType];
        IF hti # HTNull
          THEN PushHtCtx[hti, v, indirect]
          ELSE PushRecordCtx[LOOPHOLE[type, recordCSEIndex], v, indirect];
        END;
      ENDCASE => IF type # typeANY THEN ErrorDefs.errortree[typeClash, v];
    RETURN
    END;

  v ← Exp[t, typeANY];
  vType ← RType[];  type ← NormalType[vType];  RPop[];
  WITH (seb+type) SELECT FROM
    definition =>
      BEGIN
      IF hti # HTNull THEN ErrorDefs.errorhti[openId, hti];
      PushCtx[defCtx];
      END;
    pointer =>
      BEGIN
      dereferenced ← TRUE;  type ← UnderType[pointedtotype];
      OpenRecord[TRUE];
      END;
    ENDCASE =>  OpenRecord[FALSE];
  RETURN
  END;


CloseBase: PUBLIC PROCEDURE [t: TreeLink, hti: HTIndex] =
  BEGIN
  type: CSEIndex;

  CloseRecord: PROCEDURE =
    BEGIN
    WITH (seb+type) SELECT FROM
      record =>  PopCtx[];
```

```
      ENDCASE;
    RETURN
    END;

  type ← NormalType[OperandType[t]];
  WITH (seb+type) SELECT FROM
    definition => BEGIN IF hti # HTNull THEN NULL; PopCtx[] END;
    pointer => BEGIN type ← UnderType[pointedtotype]; CloseRecord[] END;
    ENDCASE => CloseRecord[];
  RETURN
  END;

-- binding of variant records

Discrimination: PUBLIC PROCEDURE [node: TreeIndex, selection: TreeMap] =
  BEGIN OPEN (tb+node);
  idNode: TreeIndex;
  type, subType, uType, tagType: CSEIndex;
  vCtx: CTXIndex;
  base, baseId: TreeLink;
  saveType: CSEIndex = passPtr.implicitType;
  saveTree: TreeLink = passPtr.implicitTree;

  BindError: PROCEDURE =
    BEGIN
    IF son2 # empty THEN son2 ← VoidExp[son2];
    vCtx ← CTXNull; tagType ← typeANY;
    RETURN
    END;

  BindItem: TreeScan =
    BEGIN
    subNode: TreeIndex;
    vType: CSEIndex;
    saveIndex: CARDINAL = dataPtr.textIndex;
    WITH t SELECT FROM
      subtree =>
        BEGIN subNode ← index;
        dataPtr.textIndex ← (tb+subNode).info;
        [(tb+subNode).son1, vType] ← BindTest[(tb+subNode).son1, vCtx];
        SetBaseType[base, vType];
        IF baseId = nullid AND (seb+vType).typetag = record
          THEN UpdateRecordCtx[LOOPHOLE[vType, recordCSEIndex]];
        (tb+subNode).son2 ← selection[(tb+subNode).son2];
        IF baseId = nullid AND (seb+type).typetag = record
          THEN UpdateRecordCtx[LOOPHOLE[type, recordCSEIndex]];
        (tb+subNode).attr1 ← TRUE;
        END;
      ENDCASE => ERROR;
    dataPtr.textIndex ← saveIndex; RETURN
    END;

  WITH son1 SELECT FROM
    subtree => idNode ← index;
    ENDCASE => ERROR;
  (tb+idNode).son2 ← Exp[(tb+idNode).son2, typeANY];
  subType ← RType[]; RPop[]; type ← NormalType[subType];
  WITH (seb+type) SELECT FROM
    pointer =>
      BEGIN
      dereferenced ← TRUE; type ← UnderType[pointedtotype];
      mlpush[(tb+idNode).son2]; pushtree[uparrow, 1];
      setinfo[type]; setattr[1, (seb+subType).typetag = long];
      base ← mlpop[];
      END;
    ENDCASE => base ← (tb+idNode).son2;
  baseId ← (tb+idNode).son1;
  WITH (seb+type) SELECT FROM
    record =>
      BEGIN
      (tb+idNode).son2 ← base ← BaseTree[base, type];
      IF baseId = nullid
        THEN PushRecordCtx[LOOPHOLE[type, recordCSEIndex], base, FALSE]
        ELSE
          WITH (tb+idNode).son1 SELECT FROM
```

```
            hash => PushHtCtx[index, base, FALSE];
            ENDCASE => ERROR;
      IF variant
        THEN
          BEGIN uType ← VariantUnionType[type];
          WITH (seb+uType) SELECT FROM
            union =>
              BEGIN
              vCtx ← casectx;
              tagType ← UnderType[(seb+tagsei).idtype];
              IF son2 = empty
                THEN
                  BEGIN
                  IF ~controlled THEN ErrorDefs.error[missingBinding];
                  CountTreeIds[base]; mlpush[base];
                  mlpush[TreeLink[symbol[index: tagsei]]];
                  pushtree[dollar, 2];
                  setinfo[tagType]; setattr[1, LongPath[base]];
                  son2 ← mlpop[];
                  END
                ELSE
                  BEGIN
                  IF controlled
                    THEN ErrorDefs.errortree[spuriousBinding, son2];
                  son2 ← Rhs[son2, TargetType[tagType]];
                  RPop[];
                  END;
              END;
            ENDCASE =>
              BEGIN ErrorDefs.error[noAccess]; BindError[] END;
          END
        ELSE
          BEGIN ErrorDefs.errortree[noVariants, (tb+idNode).son2];
          BindError[];
          END;
      END;
    ENDCASE =>
      BEGIN ErrorDefs.errortree[noVariants, (tb+idNode).son2];
      BindError[];
      END;
  passPtr.implicitType ← tagType; passPtr.implicitTree ← son2;
  scanlist[son3, BindItem];
  SetBaseType[base, type]; son4 ← selection[son4];
  WITH (seb+type) SELECT FROM
    record => PopCtx[];
    ENDCASE;
  passPtr.implicitType ← saveType; passPtr.implicitTree ← saveTree;
  RETURN
  END;

  SetBaseType: PROCEDURE [base: TreeLink, type: CSEIndex] =
    BEGIN
    IF base # empty
      THEN
        WITH base SELECT FROM
          subtree => (tb+index).info ← type;
          ENDCASE => NULL;
    RETURN
    END;


BindTest: PROCEDURE [t: TreeLink, vCtx: CTXIndex] RETURNS [val: TreeLink, vType: CSEIndex] =
  BEGIN
  mixed: BOOLEAN;

  TestItem: TreeMap =
    BEGIN
    subNode: TreeIndex;
    iType: ISEIndex;
    uType: CSEIndex;
    found: BOOLEAN;
    WITH t SELECT FROM
      subtree =>
        BEGIN subNode ← index;
        SELECT (tb+subNode).name FROM
          relE =>
```

```
            WITH (tb+subNode).son2 SELECT FROM
              hash =>
                BEGIN
                [found, iType] ← SearchCtxList[index, vCtx];
                IF found
                  THEN
                    BEGIN  uType ← UnderType[iType];
                    (tb+subNode).son2 ← TreeLink[symbol[index: iType]];
                    SELECT vType FROM
                      uType => NULL;
                      typeANY => vType ← uType;
                      ENDCASE => mixed ← TRUE;
                    END
                  ELSE
                    IF vCtx # CTXNull
                      THEN ErrorDefs.errorhti[unknownVariant, index];
                v ← t;
                END;
              ENDCASE =>
                BEGIN
                v ← Rhs[t, dataPtr.typeBOOLEAN];  RPop[];
                ErrorDefs.errortree[nonVariantLabel, t];
                END;
          ENDCASE =>
            BEGIN
            v ← Rhs[t, dataPtr.typeBOOLEAN];  RPop[];
            ErrorDefs.errortree[nonVariantLabel, t];
            END;
        END;
      ENDCASE =>  ERROR;
    RETURN
    END;

  vType ← typeANY;  mixed ← FALSE;
  val ← updatelist[t, TestItem];
  IF mixed THEN vType ← typeANY;
  RETURN
  END;


-- initialization/finalization

 IdInit: PUBLIC PROCEDURE [nIdLists: CARDINAL] =
   BEGIN
   ctxStack ← MakeStack[2*ContextIncr];  ctxIndex ← -1;
   seqNumber ← 0;
   ctxIdTable ← DESCRIPTOR[
       SystemDefs.AllocateHeapNode[nIdLists*SIZE[CtxRestriction]],
       nIdLists];
   ctxIdTableLimit ← 0;
   RETURN
   END;

 IdFinish: PUBLIC PROCEDURE =
   BEGIN
   SystemDefs.FreeHeapNode[BASE[ctxIdTable]];
   FreeStack[ctxStack];  RETURN
   END;

 END.
```