```
-- file Pass3Xb.Mesa
-- last modified by Satterthwaite, August 3, 1978  9:53 AM

DIRECTORY
  ComData: FROM "comdata"
    USING [
      ownSymbols, typeBOOLEAN, typeCHARACTER, typeINTEGER, typeSTRING, xref],
  ErrorDefs: FROM "errordefs" USING [error, errorn, errortree],
  LitDefs: FROM "litdefs" USING [StringLiteralValue],
  P3Defs: FROM "p3defs"
    USING [
      Addr, Apply, Assignment, BumpCount, Bundling, CanonicalType, Case,
      DescOp, Discrimination, Dot, Fork, Id, IdentifiedType, Join,
      MakeLongType, New, OrderedType, PopCtx, PushCtx, Range, RecordReference,
      Signal, Start, TargetType, TypeExp, TypeForTree, Unbundle, UpArrow],
  Pass3: FROM "pass3" USING [implicitRecord, implicitTree, implicitType],
  SymDefs: FROM "symdefs"
    USING [ctxtype, setype,
      SEIndex, ISEIndex, CSEIndex, recordCSEIndex, CSENull,
      recordCSENull, codeCHARACTER, codeINTEGER, typeANY],
  SymTabDefs: FROM "symtabdefs"
    USING [NormalType, TypeForm, UnderType, XferMode],
  TableDefs: FROM "tabledefs" USING [TableBase, TableNotifier],
  TreeDefs: FROM "treedefs"
    USING [treetype,
      TreeIndex, TreeLink, TreeMap, empty,
      GetNode, listlength, mlpop, mlpush, pushtree, setinfo, testtree,
      updatelist],
  TypePackDefs: FROM "typepackdefs"
    USING [SymbolTableBase, AssignableTypes, EquivalentTypes];

Pass3Xb: PROGRAM
    IMPORTS
        ErrorDefs, LitDefs, 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;
  typeANY: SymDefs.CSEIndex = SymDefs.typeANY;


  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)

  own: TypePackDefs.SymbolTableBase;

  ExpBNotify: PUBLIC TableDefs.TableNotifier =
    BEGIN  -- called by allocator whenever table area is repacked
    seb ← base[SymDefs.setype];  ctxb ← base[SymDefs.ctxtype];
    tb ← base[treetype];  RETURN
    END;


 -- intermediate result bookkeeping

  OperandDescriptor: TYPE = RECORD[
    type: CSEIndex,             -- type of operand
    const: BOOLEAN];            -- true iff a manifest constant

  RStackLimit: INTEGER = 32;
  rStack: ARRAY [0 .. RStackLimit] OF OperandDescriptor;
  rI: INTEGER;                  -- index into rStack

  OperandStackOverflow: SIGNAL = CODE;

  RPush: PUBLIC PROCEDURE [type: CSEIndex, const: BOOLEAN] =
    BEGIN
    IF rI >= RStackLimit THEN ERROR OperandStackOverflow;
```

```
    rI ← rI + 1;
    rStack[rI] ← OperandDescriptor[type:type, const:const];
    RETURN
    END;

  RPop: PUBLIC PROCEDURE =
    BEGIN
    IF rI < 0 THEN ERROR;
    rI ← rI-1;  RETURN
    END;

  RType: PUBLIC PROCEDURE RETURNS [CSEIndex] =
    BEGIN
    RETURN [rStack[rI].type]
    END;

  RConst: PUBLIC PROCEDURE RETURNS [BOOLEAN] =
    BEGIN
    RETURN [rStack[rI].const]
    END;

  ExpInit: PUBLIC PROCEDURE =
    BEGIN
    passPtr.implicitType ← typeANY;  passPtr.implicitTree ← empty;
    passPtr.implicitRecord ← SymDefs.recordCSENull;
    own ← dataPtr.ownSymbols;    -- make a parameter?
    rI ← -1;  RETURN
    END;


-- tree manipulation utilities

  OperandType: PUBLIC PROCEDURE [t: TreeLink] RETURNS [CSEIndex] =
    BEGIN
    RETURN [WITH e:t SELECT FROM
      symbol => UnderType[(seb+e.index).idtype],
      literal =>
        WITH e.info SELECT FROM
          string => dataPtr.typeSTRING,
          ENDCASE => dataPtr.typeINTEGER,
      subtree => (tb+e.index).info,
      ENDCASE => SymDefs.CSENull]
    END;


-- type manipulation

  UnresolvedTypes: SIGNAL RETURNS [CSEIndex] = CODE;

  BalanceTypes: PROCEDURE [type1, type2: CSEIndex] RETURNS [type: CSEIndex] =
    BEGIN
    n1, n2: CARDINAL;
    SELECT TRUE FROM
      (type1 = type2), (type2 = typeANY) =>  type ← type1;
      (type1 = typeANY) =>  type ← type2;
      ENDCASE =>
        BEGIN
        n1 ← Bundling[type1];
        n2 ← Bundling[type2];
        WHILE n1 > n2
          DO  type1 ← Unbundle[LOOPHOLE[type1]];  n1 ← n1-1  ENDLOOP;
        WHILE n2 > n1
          DO  type2 ← Unbundle[LOOPHOLE[type2]];  n2 ← n2-1  ENDLOOP;
        -- check bundling
          DO
          type1 ← TargetType[type1];
          type2 ← TargetType[type2];
          SELECT TRUE FROM
            TypePackDefs.AssignableTypes[[own, type1], [own, type2]] =>
              BEGIN  type ← type1;  EXIT  END;
            TypePackDefs.AssignableTypes[[own, type2], [own, type1]] =>
              BEGIN  type ← type2;  EXIT  END;
            ENDCASE;
          IF n1 = 0 THEN GO TO Fail;
          n1 ← n1-1;
          type1 ← Unbundle[LOOPHOLE[type1]];
```

```
        type2 ← Unbundle[LOOPHOLE[type2]];
        REPEAT
          Fail => type ← SIGNAL UnresolvedTypes;
        ENDLOOP;
      END;
  RETURN
  END;

ForceType: PUBLIC PROCEDURE [t: TreeLink, type: CSEIndex] RETURNS [TreeLink] =
  BEGIN
  mlpush[t];
  WITH t SELECT FROM
    subtree =>
      SELECT (tb+index).name FROM
        constructx, vconstructx, unionx, rowconsx => pushtree[cast, 1];
        ENDCASE;
    ENDCASE => pushtree[cast, 1];
  setinfo[type]; RETURN [mlpop[]]
  END;


-- expressions

Exp: PUBLIC PROCEDURE [exp: TreeLink, target: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  sei: ISEIndex;
  type: CSEIndex;
  node: TreeIndex;
  const: BOOLEAN;
  IF exp = empty
    THEN BEGIN RPush[passPtr.implicitType, FALSE]; RETURN [empty] END;
  WITH e:exp SELECT FROM
    symbol =>
      BEGIN
      sei ← e.index; BumpCount[sei];
      IF dataPtr.xref THEN RecordReference[sei, mention];
      type ← UnderType[(seb+sei).idtype];
      const ← (seb+sei).constant AND XferMode[type] # procedure;
      RPush[type, const]; val ← exp;
      END;
    hash =>
      WITH (seb+target) SELECT FROM
        enumerated =>
          BEGIN PushCtx[valuectx];
          val ← Id[e.index]; PopCtx[]
          END;
        ENDCASE => val ← Id[e.index];
    literal =>
      BEGIN
      WITH e.info SELECT FROM
        string => RPush[dataPtr.typeSTRING, FALSE];
        ENDCASE => RPush[dataPtr.typeINTEGER, TRUE];
      val ← exp;
      END;
    subtree =>
      BEGIN val ← exp; node ← e.index;
      SELECT (tb+node).name FROM
        dot => Dot[node];
        uparrow => UpArrow[node];
        apply =>
          BEGIN Apply[node, target, FALSE]; CheckNonVoid[] END;
        uminus, abs => UnaryOp[node];
        plus => Plus[node];
        minus => Minus[node];
        times, div, mod => ArithOp[node];
        relE, relN => RelOp[node, FALSE];
        relL, relGE, relG, relLE => RelOp[node, TRUE];
        in, notin =>
          BEGIN OPEN (tb+node);
          son1 ← GenericRhs[son1, typeANY];
          son2 ← Range[son2, rStack[rI].type];
          SetAttributes[node]; RPop[]; RPush[dataPtr.typeBOOLEAN, FALSE];
          END;
        not => Negation[node];
        or, and => BoolOp[node];
        ifexp => IfExp[node, target];
```

```
caseexp => SelectExp[node, target, Case];
bindexp => SelectExp[node, target, Discrimination];
assignx => Assignment[node];
min, max => MinMax[node, target];
addr => Addr[node, target];
base, length, arraydesc => DescOp[node, target];
mwconst =>
  RPush[MakeLongType[dataPtr.typeINTEGER, target], TRUE];
clit => RPush[dataPtr.typeCHARACTER, TRUE];
llit => RPush[dataPtr.typeSTRING, FALSE];
signal, error, start, join =>
  BEGIN
  val ← SELECT (tb+node).name FROM
    start => Start[node],
    join => Join[node],
    ENDCASE => Signal[node];
  node ← GetNode[val];
  CheckNonVoid[];
  END;
new, fork =>
  BEGIN
  val ← (IF (tb+node).name = fork
          THEN Fork
          ELSE New)[node, target];
  node ← GetNode[val];
  END;
lengthen =>
  BEGIN OPEN (tb+node);
  type: CSEIndex;
  son1 ← GenericRhs[son1, target]; type ← rStack[rI].type;
  IF type = dataPtr.typeINTEGER
   OR (seb+type).typetag = pointer
   OR (seb+type).typetag = arraydesc
    THEN rStack[rI].type ← MakeLongType[type, target]
    ELSE
      BEGIN
      ErrorDefs.errortree[typeClash, son1];
      rStack[rI].type ← typeANY;
      END;
  END;
loophole =>
  BEGIN OPEN (tb+node);
  son1 ← Exp[son1, typeANY];
  IF son2 = empty
    THEN
      BEGIN
      IF target = typeANY
        THEN ErrorDefs.errortree[noTarget, [subtree[node]]];
      rStack[rI].type ← target;
      END
    ELSE
      BEGIN son2 ← TypeExp[son2];
      rStack[rI].type ← UnderType[TypeForTree[son2]];
      END;
  END;
register =>
  BEGIN OPEN (tb+node);
  son1 ← Rhs[son1, dataPtr.typeINTEGER]; attr1 ← FALSE;
  IF ~rStack[rI].const THEN ErrorDefs.errortree[nonConstant, son1];
  RPop[]; RPush[typeANY, FALSE];
  END;
memory =>
  BEGIN OPEN (tb+node);
  son1 ← Rhs[son1, dataPtr.typeINTEGER]; attr1 ← FALSE;
  RPop[]; RPush[typeANY, FALSE];
  END;
size =>
  BEGIN OPEN (tb+node);
  son1 ← TypeExp[son1];
  RPush[dataPtr.typeINTEGER, TRUE];
  END;
first, last => EndPoint[node];
item => (tb+node).son2 ← Exp[(tb+node).son2, target];
ENDCASE =>
  BEGIN ErrorDefs.error[unimplemented];
  RPush[typeANY, FALSE];
```

```
          END;
        (tb+node).info ← rStack[rI].type;
        END;
      ENDCASE;
    RETURN
    END;

    CheckNonVoid: PROCEDURE =
      BEGIN
      IF rStack[rI].type = SymDefs.CSENull
        THEN
          BEGIN  ErrorDefs.error[voidExpr];
          rStack[rI].type ← typeANY;
          END;
      RETURN
      END;

  VoidExp: PUBLIC PROCEDURE [exp: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    val ← Exp[exp, typeANY];  RPop[];  RETURN
    END;


-- literals

  TreeStringValue: PUBLIC PROCEDURE [t: TreeLink] RETURNS [STRING] =
    BEGIN
    WITH e:t SELECT FROM
      literal =>
        WITH e.info SELECT FROM
          string => RETURN [LitDefs.StringLiteralValue[index]];
          ENDCASE;
      ENDCASE;
    ERROR
    END;


-- arithmetic expression manipulation

  EvalNumeric: PROCEDURE [t: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    val ← GenericRhs[t, dataPtr.typeINTEGER];
    SELECT NormalType[rStack[rI].type] FROM
      dataPtr.typeINTEGER =>  NULL;
      typeANY => rStack[rI].type ← dataPtr.typeINTEGER;
      ENDCASE => ErrorDefs.errortree[typeClash, val];
    RETURN
    END;

  ArithOp: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    son1 ← EvalNumeric[son1];  son2 ← EvalNumeric[son2];
    BalanceAttributes[node];
    rStack[rI-1].const ← rStack[rI-1].const AND rStack[rI].const AND ~attr1;
    RPop[];  RETURN
    END;


  ArithType: PROCEDURE [type: CSEIndex] RETURNS [CSEIndex] =
    BEGIN
    type ← NormalType[type];
    RETURN [WITH (seb+type) SELECT FROM
      relative => NormalType[UnderType[offsetType]],
      ENDCASE => type]
    END;

  Plus: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    type: CSEIndex;
    lr: BOOLEAN;
    son1 ← GenericRhs[son1, typeANY];
    type ← ArithType[rStack[rI].type];
    IF (seb+type).typetag = pointer OR type = dataPtr.typeCHARACTER
      THEN  BEGIN lr ← TRUE; son2 ← EvalNumeric[son2] END
      ELSE
        BEGIN
```

```
      SELECT type FROM
        dataPtr.typeINTEGER, typeANY => NULL;
        ENDCASE => ErrorDefs.errortree[typeClash, son1];
      son2 ← GenericRhs[son2, typeANY];
      lr ← FALSE;  type ← ArithType[rStack[rI].type];
      IF (seb+type).typetag = pointer OR type = dataPtr.typeCHARACTER
        THEN  NULL
        ELSE
          BEGIN
          SELECT type FROM
            dataPtr.typeINTEGER => NULL;
            typeANY => rStack[rI].type ← dataPtr.typeINTEGER;
            ENDCASE => ErrorDefs.errortree[typeClash, son2];
          END;
      END;
  BalanceAttributes[node];
  rStack[rI-1].const ← rStack[rI-1].const AND rStack[rI].const AND ~attr1;
  IF ~lr THEN rStack[rI-1].type ← rStack[rI].type;
  RPop[];  RETURN
  END;

Minus: PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  type, subType: CSEIndex;
  lr: BOOLEAN;
  son1 ← GenericRhs[son1, typeANY];
  type ← NormalType[rStack[rI].type]; subType ← ArithType[type];  lr ← TRUE;
  IF (seb+subType).typetag = pointer OR subType = dataPtr.typeCHARACTER
    THEN
      BEGIN
      son2 ← GenericRhs[son2, typeANY];
      subType ← NormalType[rStack[rI].type];
      SELECT TRUE FROM
        subType = typeANY =>  NULL;
        TypePackDefs.EquivalentTypes[[own, type], [own, subType]] =>
          lr ← FALSE;
        subType = dataPtr.typeINTEGER =>  NULL;
        ENDCASE => ErrorDefs.errortree[typeClash, son2];
      END
    ELSE
      BEGIN
      SELECT type FROM
        dataPtr.typeINTEGER, typeANY =>  NULL;
        ENDCASE => ErrorDefs.errortree[typeClash, son1];
      son2 ← EvalNumeric[son2];
      END;
  BalanceAttributes[node];
  rStack[rI-1].const ← rStack[rI-1].const AND rStack[rI].const AND ~attr1;
  IF ~lr
    THEN rStack[rI-1].type ← IF attr1
              THEN MakeLongType[dataPtr.typeINTEGER, rStack[rI].type]
              ELSE dataPtr.typeINTEGER;
  RPop[];  RETURN
  END;

UnaryOp: PROCEDURE [node: TreeIndex] =
  BEGIN  OPEN (tb+node);
  l: CARDINAL = listlength[son1];
  IF l = 1
    THEN
      BEGIN
      son1 ← EvalNumeric[son1];  SetAttributes[node];
      IF attr1 THEN rStack[rI].const ← FALSE;
      END
    ELSE
      BEGIN
      IF l > 1
        THEN ErrorDefs.errorn[listLong, l-1]
        ELSE ErrorDefs.errorn[listShort, l+1];
      son1 ← updatelist[son1, VoidExp];
      RPush[typeANY, FALSE];
      END;
  RETURN
  END;

RelOp: PROCEDURE [node: TreeIndex, ordered: BOOLEAN] =
```

```
    BEGIN  OPEN (tb+node);
    type: CSEIndex;
    const: BOOLEAN;
    son1 ← GenericRhs[son1, typeANY];
    type ← NormalType[rStack[rI].type];
    son2 ← GenericRhs[son2, type];
    type ← BalanceTypes[type, NormalType[rStack[rI].type]
      !UnresolvedTypes =>
        BEGIN  ErrorDefs.errortree[typeClash, son2];  RESUME [typeANY]  END];
    IF (ordered AND ~OrderedType[type]) OR
       (~ordered AND ~IdentifiedType[type])
      THEN ErrorDefs.errortree[relationType, [subtree[node]]];
    IF son1 # empty THEN BalanceAttributes[node] ELSE SetAttributes[node];
    const ← SELECT (seb+type).typetag FROM
      basic, enumerated => rStack[rI-1].const AND rStack[rI].const,
      ENDCASE => FALSE;
    RPop[];  RPop[];  RPush[dataPtr.typeBOOLEAN, const];
    RETURN
    END;

  Negation: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    const: BOOLEAN;
    son1 ← Rhs[son1, dataPtr.typeBOOLEAN];
    const ← rStack[rI].const;
    RPop[];  RPush[dataPtr.typeBOOLEAN, const];
    RETURN
    END;

  BoolOp: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    const: BOOLEAN;
    son1 ← Rhs[son1, dataPtr.typeBOOLEAN];
    son2 ← Rhs[son2, dataPtr.typeBOOLEAN];
    const ← rStack[rI-1].const AND rStack[rI].const;
    RPop[];  RPop[];  RPush[dataPtr.typeBOOLEAN, const];
    RETURN
    END;


  BalancedTarget: PROCEDURE [target, type: CSEIndex] RETURNS [CSEIndex] =
    BEGIN
    RETURN [IF target = typeANY
        OR (~TypePackDefs.EquivalentTypes[[own, type], [own, target]]
            AND NormalType[type] = target)
      THEN TargetType[type]
      ELSE target]
    END;

  ResolveTypes: PROCEDURE [type1, type2, target: CSEIndex, t: TreeLink]
      RETURNS [type: CSEIndex] =
    BEGIN
    failed: BOOLEAN;
    IF target = typeANY
      THEN  failed ← TRUE
      ELSE
        BEGIN
        ENABLE UnresolvedTypes =>  BEGIN failed ← TRUE; RESUME [typeANY] END;
        failed ← FALSE;
        type1 ← BalanceTypes[target, type1];
        type2 ← BalanceTypes[target, type2];
        type ← BalanceTypes[type1, type2];
        END;
    IF failed
      THEN BEGIN ErrorDefs.errortree[typeClash, t]; type ← typeANY END;
    RETURN
    END;

  IfExp: PROCEDURE [node: TreeIndex, target: CSEIndex] =
    BEGIN  OPEN (tb+node);
    type: CSEIndex;
    const: BOOLEAN;
    son1 ← Rhs[son1, dataPtr.typeBOOLEAN];
    const ← rStack[rI].const;  RPop[];
    son2 ← BalancedRhs[son2, target];  const ← const AND rStack[rI].const;
    type ← rStack[rI].type;  RPop[];
```

```
    target ← BalancedTarget[target, type];
    son3 ← BalancedRhs[son3, target];  const ← const AND rStack[rI].const;
    type ← BalanceTypes[type, rStack[rI].type
        !UnresolvedTypes =>
          RESUME [ResolveTypes[type, rStack[rI].type, target, son3]]];
    RPop[];  RPush[type, const];  RETURN
    END;

  SelectExp: PROCEDURE [node: TreeIndex, target: CSEIndex, driver: PROCEDURE [TreeIndex, TreeMap]] =
    BEGIN
    type: CSEIndex;
    started: BOOLEAN;

    Selection: TreeMap =
      BEGIN
      subType: CSEIndex;
      v ← BalancedRhs[t, target];
      subType ← BalanceTypes[type, rStack[rI].type
        !UnresolvedTypes =>
          RESUME [ResolveTypes[type, rStack[rI].type, target, v]]];
      IF subType # typeANY THEN type ← subType;
      IF ~started THEN target ← BalancedTarget[target, type];
      RPop[];  started ← TRUE;  RETURN
      END;

    type ← typeANY;  started ← FALSE;
    driver[node, Selection];
    RPush[type, FALSE];   RETURN
    END;


  MinMax: PROCEDURE [node: TreeIndex, target: CSEIndex] =
    BEGIN  OPEN (tb+node);
    const, started: BOOLEAN;
    type: CSEIndex;

    SubMinMax: TreeMap =
      BEGIN
      subType: CSEIndex;
      v ← BalancedRhs[t, target];
      const ← const AND rStack[rI].const;
      subType ← CanonicalType[rStack[rI].type];
      subType ← BalanceTypes[subType, type
                !UnresolvedTypes =>
                  RESUME[ResolveTypes[subType, type, target, v]]];
      IF type # subType AND subType # typeANY
        THEN
          BEGIN
          IF ~OrderedType[subType]
            THEN ErrorDefs.errortree[relationType, [subtree[node]]];
          type ← subType;
          IF ~started THEN target ← BalancedTarget[target, type];
          END;
      RPop[];  started ← TRUE;  RETURN
      END;

    const ← TRUE;  started ← FALSE;  type ← typeANY;
    son1 ← updatelist[son1, SubMinMax];
    SELECT (seb+type).typetag FROM
      real =>  attr1 ← attr2 ← TRUE;
      long =>  BEGIN  attr1 ← TRUE;  attr2 ← FALSE  END;
      ENDCASE =>  attr1 ← attr2 ← FALSE;
    RPush[type, const AND ~attr1];  RETURN
    END;


  EndPoint: PROCEDURE [node: TreeIndex] =
    BEGIN  OPEN (tb+node);
    type: CSEIndex;
    son1 ← TypeExp[son1];
    type ← UnderType[TypeForTree[son1]];
      BEGIN
      WITH (seb+type) SELECT FROM
        basic =>
```

```
        SELECT code FROM
          SymDefs.codeINTEGER, SymDefs.codeCHARACTER => NULL;
          ENDCASE => GO TO fail;
      enumerated => NULL;
      relative =>
        IF TypeForm[offsetType] # subrange THEN GO TO fail;
      subrange => NULL;
      long =>
        IF UnderType[rangetype] # dataPtr.typeINTEGER THEN GO TO fail;
      ENDCASE => GO TO fail;
    EXITS
      fail => ErrorDefs.errortree[typeClash, son1];
    END;
  RPush[type, TRUE]; RETURN
  END;

Rhs: PUBLIC PROCEDURE [exp: TreeLink, lhsType: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  rhsType: CSEIndex;
  val ← Exp[exp, lhsType];
  rhsType ← rStack[rI].type;
  SELECT TRUE FROM
    (lhsType = rhsType), (lhsType = typeANY) => NULL;
    (rhsType = typeANY) => rStack[rI].type ← lhsType;
    ENDCASE =>
      BEGIN -- immediate matching is inconclusive
      UNTIL TypePackDefs.AssignableTypes[[own, lhsType], [own, rhsType]]
        DO
        WITH (seb+rhsType) SELECT FROM
          subrange => rhsType ← UnderType[rangetype];
          record =>
            BEGIN
            IF Bundling[rhsType] = 0 THEN GO TO nomatch;
            rhsType ← Unbundle[LOOPHOLE[rhsType, RecordSEIndex]];
            val ← ForceType[val, rhsType];
            END;
          ENDCASE =>
            BEGIN
            SELECT (seb+lhsType).typetag FROM
              long =>
                BEGIN
                IF ~TypePackDefs.EquivalentTypes[
                    [own, NormalType[lhsType]], [own, rhsType]]
                  THEN GO TO nomatch;
                val ← Lengthen[val, lhsType];
                END;
              real =>
                BEGIN
                IF NormalType[rhsType] # dataPtr.typeINTEGER
                  THEN GO TO nomatch;
                val ← Float[val, rhsType, lhsType];
                END;
              ENDCASE => GO TO nomatch;
            rhsType ← lhsType;
            END
        REPEAT
          nomatch =>
            BEGIN     -- no coercion is possible
            ErrorDefs.errortree[typeClash,
              IF exp = empty THEN passPtr.implicitTree ELSE val];
            rhsType ← lhsType;
            END;
        ENDLOOP;
      rStack[rI].type ← rhsType;
      END;
  RETURN
  END;

GenericRhs: PROCEDURE [exp: TreeLink, target: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  type: CSEIndex;
  val ← Exp[exp, target]; type ← rStack[rI].type;
  -- put value in canonical form
    DO
```

```
    WITH (seb+type) SELECT FROM
      subrange => type ← UnderType[rangetype];
      record =>
        BEGIN
        IF Bundling[type] = 0 THEN EXIT;
        type ← Unbundle[LOOPHOLE[type, RecordSEIndex]];
        val ← ForceType[val, type];
        END;
      ENDCASE => EXIT;
    rStack[rI].type ← type;
    ENDLOOP;
  RETURN
  END;

BalancedRhs: PROCEDURE [exp: TreeLink, target: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  type: CSEIndex;
  val ← Exp[exp, target];
  SELECT (seb+target).typetag FROM
    long, real =>
      BEGIN
      type ← CanonicalType[rStack[rI].type];
      IF type # typeANY AND (seb+target).typetag # (seb+type).typetag
       AND TypePackDefs.EquivalentTypes[
          [own, NormalType[target]], [own, type]]
        THEN
          BEGIN
          SELECT (seb+target).typetag FROM
            long => val ← Lengthen[val, target];
            real => val ← Float[val, type, target];
            ENDCASE;
          rStack[rI].type ← target; rStack[rI].const ← FALSE;
          END;
      END;
    ENDCASE;
  RETURN
  END;


SetAttributes: PROCEDURE [node: TreeIndex] =
  BEGIN
  SELECT (seb+rStack[rI].type).typetag FROM
    long => BEGIN (tb+node).attr1 ← TRUE; (tb+node).attr2 ← FALSE END;
    real => (tb+node).attr1 ← (tb+node).attr2 ← TRUE;
    ENDCASE => (tb+node).attr1 ← (tb+node).attr2 ← FALSE;
  RETURN
  END;


BalanceAttributes: PROCEDURE [node: TreeIndex] =
  BEGIN
  lType, rType: CSEIndex;
  lType ← rStack[rI-1].type; rType ← rStack[rI].type;
  SELECT (seb+lType).typetag FROM
    long =>
      BEGIN
      (tb+node).attr1 ← TRUE;
      SELECT (seb+rType).typetag FROM
        long => (tb+node).attr2 ← FALSE;
        real =>
          BEGIN
          rStack[rI-1].type ← rType;
          (tb+node).son1 ← Float[(tb+node).son1, lType, rType];
          rStack[rI-1].const ← FALSE; (tb+node).attr2 ← TRUE;
          END;
        ENDCASE =>
          BEGIN
          rStack[rI].type ← rType ← MakeLongType[rType, lType];
          (tb+node).son2 ← Lengthen[(tb+node).son2, rType];
          rStack[rI].const ← FALSE; (tb+node).attr2 ← FALSE;
          END;
      END;
    real =>
      BEGIN
      (tb+node).attr1 ← (tb+node).attr2 ← TRUE;
      SELECT (seb+rType).typetag FROM
```

```
        real => NULL;
        ENDCASE =>
          BEGIN
          rStack[rI].type ← lType;
          (tb+node).son2 ← Float[(tb+node).son2, rType, lType];
          rStack[rI].const ← FALSE;
          END;
      END;
    ENDCASE =>
      SELECT (seb+rType).typetag FROM
        long =>
          BEGIN
          rStack[rI-1].type ← lType ← MakeLongType[lType, rType];
          (tb+node).son1 ← Lengthen[(tb+node).son1, lType];
          rStack[rI-1].const ← FALSE;
          (tb+node).attr1 ← TRUE;  (tb+node).attr2 ← FALSE;
          END;
        real =>
          BEGIN
          rStack[rI-1].type ← rType;
          (tb+node).son1 ← Float[(tb+node).son1, lType, rType];
          rStack[rI-1].const ← FALSE;
          (tb+node).attr1 ← (tb+node).attr2 ← TRUE;
          END;
        ENDCASE => (tb+node).attr1 ← (tb+node).attr2 ← FALSE;
  RETURN
  END;

Lengthen: PROCEDURE [t: TreeLink, target: CSEIndex] RETURNS [v: TreeLink] =
  BEGIN
  IF testtree[t, arraydesc]
    THEN v ← LengthenDesc[t, target]
    ELSE
      BEGIN
      mlpush[t]; pushtree[lengthen, 1]; setinfo[target]; v ← mlpop[];
      END;
  RETURN
  END;

LengthenDesc: PROCEDURE [t: TreeLink, target: CSEIndex] RETURNS [TreeLink] =
  BEGIN
  node: TreeIndex = GetNode[t];
  subNode: TreeIndex = GetNode[(tb+node).son1];
  (tb+subNode).son1 ← Lengthen[(tb+subNode).son1,
      MakeLongType[OperandType[(tb+subNode).son1], typeANY]];
  (tb+node).info ← MakeLongType[(tb+node).info, target];
  (tb+node).attr1 ← TRUE;
  RETURN [t]
  END;

Float: PROCEDURE [t: TreeLink, type, target: CSEIndex] RETURNS [TreeLink] =
  BEGIN
  IF NormalType[type] # dataPtr.typeINTEGER
    THEN ErrorDefs.errortree[typeClash, t];
  IF (seb+type).typetag = long
    THEN mlpush[t]
    ELSE mlpush[Lengthen[t, MakeLongType[type, typeANY]]];
  pushtree[float, 1]; setinfo[target];
  RETURN [mlpop[]]
  END;

END.
```