```
-- file Pass4Xb.Mesa
-- last written by Satterthwaite, August 2, 1978  10:33 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  ComData: FROM "comdata",
  ErrorDefs: FROM "errordefs",
  InlineDefs: FROM "inlinedefs",
  LitDefs: FROM "litdefs",
  P4Defs: FROM "p4defs",
  Pass4: FROM "pass4",
  SymDefs: FROM "symdefs",
  SymSegDefs: FROM "symsegdefs",
  SymTabDefs: FROM "symtabdefs",
  TableDefs: FROM "tabledefs",
  TreeDefs: FROM "treedefs";

Pass4Xb: PROGRAM
    IMPORTS
        ErrorDefs, LitDefs, P4Defs, SymSegDefs, SymTabDefs, TreeDefs,
        dataPtr: ComData, passPtr: Pass4
    EXPORTS P4Defs =
  BEGIN
  OPEN SymTabDefs, TreeDefs;

 -- pervasive definitions from SymDefs

  ISEIndex: TYPE = SymDefs.ISEIndex;
  CSEIndex: TYPE = SymDefs.CSEIndex;


  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)

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


 -- respresentations (temporary)

  Repr: TYPE = P4Defs.Repr;
    none: Repr = P4Defs.none;
    signed: Repr = P4Defs.signed;
    unsigned: Repr = P4Defs.unsigned;
    both: Repr = P4Defs.both;
    other: Repr = P4Defs.other;

  MixedRepresentation: PUBLIC SIGNAL = CODE;

  CommonRep: PROCEDURE [Repr, Repr] RETURNS [Repr] =
    LOOPHOLE[InlineDefs.BITAND];

  Compare: PUBLIC PROCEDURE [l, r: WORD, rep: Repr] RETURNS [INTEGER] =
    BEGIN
    RETURN [IF l = r
      THEN 0
      ELSE
        IF (IF CommonRep[rep, unsigned] # none
                THEN l > r
                ELSE LOOPHOLE[l, INTEGER] > LOOPHOLE[r, INTEGER])
          THEN 1
          ELSE -1]
    END;


 -- intermediate result bookkeeping

  ValueDescriptor: TYPE = RECORD[
    bias: INTEGER,              -- bias in representation (scalars only)
    rep: Repr]; -- signed/unsigned (scalars only)
```

```
VStackLimit: INTEGER = 32;
vStack: ARRAY [0 .. VStackLimit] OF ValueDescriptor;
vI: INTEGER;          -- index into vStack

VStackOverflow: SIGNAL = CODE;

VPush: PUBLIC PROCEDURE [bias: INTEGER, rep: Repr] =
  BEGIN
  IF (vI ← vI+1) > VStackLimit THEN ERROR VStackOverflow;
  vStack[vI] ← ValueDescriptor[bias:bias, rep:rep];
  RETURN
  END;

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

VBias: PUBLIC PROCEDURE RETURNS [INTEGER] =
  BEGIN
  RETURN [vStack[vI].bias]
  END;

VRep: PUBLIC PROCEDURE RETURNS [Repr] =
  BEGIN
  RETURN [vStack[vI].rep]
  END;



Pass4XInit: PUBLIC PROCEDURE =
  BEGIN
  vI ← -1;  RETURN
  END;


OperandType: PUBLIC PROCEDURE [t: TreeLink] RETURNS [CSEIndex] =
  BEGIN
  RETURN [WITH t SELECT FROM
    symbol => UnderType[(seb+index).idtype],
    literal =>
      IF info.litTag = string THEN dataPtr.typeSTRING ELSE dataPtr.typeINTEGER,
    subtree =>
      IF t = empty THEN passPtr.implicitType ELSE (tb+index).info,
    ENDCASE => SymDefs.typeANY]
  END;

ForceType: PUBLIC PROCEDURE [t: TreeLink, type: CSEIndex] RETURNS [TreeLink] =
  BEGIN
  mlpush[t];
  IF ~testtree[t, mwconst] THEN pushtree[cast, 1];
  setinfo[type];  RETURN [mlpop[]]
  END;


-- literals

TreeLiteral: PUBLIC PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
  BEGIN
  node: TreeIndex;
    DO
    WITH t SELECT FROM
      literal => RETURN[info.litTag = word];
      subtree =>
        BEGIN  node ← index;
        SELECT (tb+node).name FROM
          cast =>  t ← (tb+node).son1;
          ENDCASE => RETURN [FALSE];
        END;
      ENDCASE => RETURN[FALSE]
    ENDLOOP
  END;

TreeLiteralValue: PUBLIC PROCEDURE [t: TreeLink] RETURNS [WORD] =
```

```
    BEGIN
    node: TreeIndex;
      DO
      WITH e:t SELECT FROM
        literal =>
          WITH e.info SELECT FROM
            word => RETURN [LitDefs.LiteralValue[index]];
            ENDCASE => EXIT;
        subtree =>
          BEGIN  node ← e.index;
          SELECT (tb+node).name FROM
            cast =>  t ← (tb+node).son1;
            ENDCASE => EXIT;
          END;
        ENDCASE => EXIT
      ENDLOOP;
    ERROR;
    END;

  MakeTreeLiteral: PUBLIC PROCEDURE [val: WORD] RETURNS [TreeLink] =
    BEGIN
    RETURN [[[literal[info: [word[index: LitDefs.FindLiteral[val]]]]]]]
    END;


  StructuredLiteral: PROCEDURE [t: TreeLink] RETURNS [BOOLEAN] =
    BEGIN
    node: TreeIndex;
      DO
      WITH t SELECT FROM
        literal => RETURN[info.litTag = word];
        subtree =>
          BEGIN  node ← index;
          SELECT (tb+node).name FROM
            mwconst => RETURN [TRUE];
            cast =>  t ← (tb+node).son1;
            ENDCASE => RETURN [FALSE];
          END;
        ENDCASE => RETURN[FALSE]
      ENDLOOP
    END;

  MakeStructuredLiteral: PUBLIC PROCEDURE [val: WORD, type: CSEIndex] RETURNS [t: TreeLink] =
    BEGIN
    t ← MakeTreeLiteral[val];
    SELECT (seb+type).typetag FROM
      basic, enumerated, subrange, mode => NULL;
      ENDCASE => t ← ForceType[t, type];
    RETURN
    END;


  LiteralRep: PROCEDURE [val: WORD, rep: Repr] RETURNS [Repr] =
    BEGIN
    RETURN [IF rep = other OR rep = none
      THEN rep
      ELSE IF val > 77777B
        THEN IF rep = both THEN unsigned ELSE rep
        ELSE both]
    END;


-- operators

  Exp: PUBLIC PROCEDURE [exp: TreeLink, target: Repr] RETURNS [val: TreeLink] =
    BEGIN
    sei: ISEIndex;
    type: CSEIndex;
    rep: Repr;
    node: TreeIndex;
    WITH expr: exp SELECT FROM

      symbol =>
        BEGIN  sei ← expr.index;
        IF ~(seb+sei).mark4
          THEN  P4Defs.DeclItem[TreeLink[subtree[index: (seb+sei).idvalue]]];
```

```
    type ← UnderType[(seb+sei).idtype];  rep ← P4Defs.RepForType[type];
    IF ~(seb+sei).constant
      THEN val ← expr
      ELSE
        SELECT XferMode[type] FROM
          procedure, signal, error =>
            val ← IF ConstantId[sei] AND ~(seb+sei).extended
              THEN MakeStructuredLiteral[(seb+sei).idvalue, type]
              ELSE expr;
          ENDCASE =>
            IF (seb+sei).extended
              THEN
                BEGIN
                val ← IdentityMap[SymSegDefs.FindExtension[sei]];
                WITH val SELECT FROM
                  subtree => (tb+index).info ←UnderType[(seb+sei).idtype];
                  ENDCASE;
                END
              ELSE
                BEGIN  rep ← LiteralRep[(seb+sei).idvalue, rep];
                val ← MakeStructuredLiteral[(seb+sei).idvalue, type];
                END;
    VPush[P4Defs.BiasForType[type], rep];
    END;

  literal =>
    BEGIN
    WITH expr.info SELECT FROM
      word =>  rep ← LiteralRep[LitDefs.LiteralValue[index], unsigned];
      string =>
        BEGIN LitDefs.StringLiteralReference[index]; rep ← unsigned END;
      ENDCASE => rep ← none;
    VPush[0, rep];  val ← expr;
    END;

  subtree =>
    IF expr = empty
      THEN
        BEGIN  val ← empty;
        VPush[passPtr.implicitBias, passPtr.implicitRep];
        END
      ELSE
        BEGIN  node ← expr.index;
        SELECT (tb+node).name FROM

          dot =>
            BEGIN  OPEN (tb+node);
            son1 ← NeutralExp[son1];  son2 ← Exp[son2, none];  val ← expr;
            END;

          dollar =>  val ← Dollar[node];

          cdot =>
            BEGIN
            val ← Exp[(tb+node).son2, none];
            (tb+node).son2 ← empty;  freenode[node];
            END;

          uparrow =>
            BEGIN  OPEN (tb+node);
            son1 ← NeutralExp[son1];
            VPush[P4Defs.BiasForType[info], P4Defs.RepForType[info]];
            val ← expr;
            END;

          call, portcall, signal, error, start, join =>
            val ← P4Defs.Call[node];
          index, dindex =>  val ← Index[node];

          seqindex =>
            BEGIN  OPEN (tb+node);
            son1 ← NeutralExp[son1];  son2 ← NeutralExp[son2];
            VPush[0, both];  val ← expr;
            END;

          reloc =>
```

```
    BEGIN  OPEN (tb+node);
    son1 ← NeutralExp[son1];  son2 ← NeutralExp[son2];
    VPush[0, unsigned];  val ← expr;
    END;

  constructx =>  val ← P4Defs.Construct[node];
  unionx =>  val ← P4Defs.Union[node];
  rowconsx =>  val ← P4Defs.RowConstruct[node];

  uminus =>
    BEGIN  OPEN (tb+node);
    IF ~TreeLiteral[son1 ← Exp[son1, signed]]
      THEN val ← expr
      ELSE
        BEGIN
        val ← MakeTreeLiteral[-TreeLiteralValue[son1]];
        freenode[node];
        END;
    vStack[vI].bias ← -vStack[vI].bias;
    SELECT vStack[vI].rep FROM
      unsigned, both =>  vStack[vI].rep ← signed;
      none =>
        BEGIN
        ErrorDefs.WarningTree[mixedRepresentation, val];
        vStack[vI].rep ← signed;
        END;
      ENDCASE =>  NULL;
    IF ~attr1 THEN  attr2 ← FALSE;
    END;

  abs =>
    BEGIN  OPEN (tb+node);
    son1 ← RValue[son1, 0, signed];
    SELECT vStack[vI].rep FROM
      unsigned, both =>
        BEGIN
        ErrorDefs.WarningTree[unsignedCompare, expr];
        val ← son1;  son1 ← empty;  freenode[node];
        END;
      none =>
        BEGIN
        val ← expr;
        ErrorDefs.errortree[mixedRepresentation, val];
        vStack[vI].rep ← both;
        END;
      ENDCASE =>
        BEGIN
        IF ~attr1 THEN
          BEGIN attr2 ← FALSE; vStack[vI].rep ← both  END;
        IF ~TreeLiteral[son1]
          THEN  val ← expr
          ELSE
            BEGIN
            val ← MakeTreeLiteral[
                  ABS[LOOPHOLE[TreeLiteralValue[son1], INTEGER]]];
            freenode[node];
            END;
        END;
    END;

  plus, minus =>  val ← AddOp[node, target];
  times =>  val ← Mult[node, target];
  div, mod =>  val ← DivMod[node, target];
  relE, relN, relL, relGE, relG, relLE =>  val ← RelOp[node];
  in, notin =>  val ← In[node];

  not =>
    BEGIN  OPEN (tb+node);
    IF ~TreeLiteral[son1 ← Exp[son1, none]]
      THEN  val ← expr
      ELSE
        BEGIN
        val ← IF son1 # passPtr.tFALSE
          THEN passPtr.tFALSE
          ELSE passPtr.tTRUE;
        freenode[node];
```

```
      END;
  END;

or, and => val ← BoolOp[node];
ifexp => val ← IfExp[node, target];
caseexp => val ← CaseExp[node, target];
bindexp => val ← BindExp[node, target];
assignx => val ← P4Defs.Assignment[node];
min, max => val ← MinMax[node, target];
addr => val ← Addr[node];

base =>
  BEGIN OPEN (tb+node);
  son1 ← Exp[son1, none]; VPop[];
  VPush[0, unsigned]; val ← expr;
  END;

length =>
  BEGIN OPEN (tb+node);
  type: CSEIndex;
  son1 ← Exp[son1, none];
  type ← OperandType[son1]; VPop[];
  WITH (seb+type) SELECT FROM
    array =>
      BEGIN
      val ← MakeTreeLiteral[Cardinality[indextype]];
      freenode[node];
      END;
    ENDCASE => val ← expr;
  VPush[0, both];
  END;

arraydesc =>
  BEGIN OPEN (tb+node);
  subNode: TreeIndex = GetNode[son1];
  (tb+subNode).son1 ← NeutralExp[(tb+subNode).son1];
  (tb+subNode).son2 ← NeutralExp[(tb+subNode).son2];
  IF (tb+subNode).son3 # empty
    THEN P4Defs.TypeExp[typeExp:(tb+subNode).son3, body:FALSE];
  VPush[0, other]; val ← expr;
  END;

mwconst =>
  BEGIN
  val ← expr; VPush[0, other];
  END;

clit =>
  BEGIN
  val ← (tb+node).son1; freenode[node]; VPush[0, both];
  END;

llit =>
  BEGIN
  IF (bb+dataPtr.bodyIndex).level > SymDefs.lG THEN
    WITH e: (tb+node).son1 SELECT FROM
      literal =>
        WITH e.info SELECT FROM
          string =>
            index ← LitDefs.FindLocalStringLiteral[index];
          ENDCASE;
      ENDCASE;
  val ← Exp[(tb+node).son1, none];
  (tb+node).son1 ← empty; freenode[node];
  END;

new => val ← P4Defs.New[node];
fork => val ← P4Defs.Fork[node];

memory, register =>
  BEGIN OPEN (tb+node);
  son1 ← NeutralExp[son1]; VPush[0, both+other]; val ← expr;
  END;

lengthen => val ← Lengthen[node];
```

```
            float =>
              BEGIN  OPEN (tb+node);
              son1 ← NeutralExp[son1];  VPush[0, other];  val ← expr;
              END;

            loophole =>  val ← Loophole[node];

            cast =>
              BEGIN  OPEN (tb+node);
              rep: Repr = P4Defs.RepForType[info];
              son1 ← Exp[son1, rep];
              vStack[vI].rep ← rep;
              val ← expr;
              END;

            openexp =>
              BEGIN  OPEN (tb+node);
              IF attr1
                THEN val ← son1
                ELSE
                  BEGIN  son1 ← NeutralExp[son1];
                  IF TreeDefs.shared[son1]
                    THEN  son1 ← ForceType[son1, OperandType[son1]];
                  setshared[son1, TRUE];  attr1 ← TRUE;
                  val ← expr;
                  END;
              VPush[0, other];
              END;

            size =>
              BEGIN  OPEN (tb+node);
              P4Defs.TypeExp[typeExp:son1, body:FALSE];
              val ← MakeTreeLiteral[P4Defs.WordsForType[
                      UnderType[P4Defs.TypeForTree[son1]]]];
              freenode[node];  VPush[0, both];
              END;

            first, last =>  val ← EndPoint[node];

            ENDCASE =>
              BEGIN  ErrorDefs.error[unimplemented];
              VPush[0, none];  val ← expr;
              END;

          END;

    ENDCASE =>  ERROR;
  RETURN
  END;

NeutralExp: PUBLIC PROCEDURE [exp: TreeLink] RETURNS [val: TreeLink] =
  BEGIN
  val ← RValue[exp, 0, none];  VPop[];  RETURN
  END;


Dollar: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN  OPEN (tb+node);
  sei: ISEIndex;
  son1 ← Exp[son1, none];  VPop[];  son2 ← Exp[son2, none];
  IF ~StructuredLiteral[son1]
    THEN val ← TreeLink[subtree[index: node]]
    ELSE
      WITH son2 SELECT FROM
        symbol =>
          BEGIN  sei ← index;
          val ← P4Defs.UnpackField[son1, sei];
          freenode[node];
          END;
        ENDCASE => ERROR;
  RETURN
  END;


Index: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN  OPEN (tb+node);
```

```
    aType, iType, cType: CSEIndex;
    k: CARDINAL;
    son1 ← Exp[son1, none];  VPop[];
    aType ← OperandType[son1];
      DO
      WITH (seb+aType) SELECT FROM
        array =>
          BEGIN
          iType ← UnderType[indextype]; cType ← UnderType[componenttype]; EXIT
          END;
        arraydesc =>  aType ← UnderType[describedType];
        long =>  aType ← UnderType[rangetype];
        ENDCASE =>  ERROR;
      ENDLOOP;
    IF (k ← P4Defs.WordsForType[cType]) # 1
      THEN
        BEGIN  mlpush[son2];  mlpush[MakeTreeLiteral[k]];
        pushtree[times, 2];  setinfo[dataPtr.typeINTEGER];  setattr[1, FALSE];
        son2 ← mlpop[];
        END;
    son2 ← RValue[son2, k*P4Defs.BiasForType[iType],
              P4Defs.TargetRep[P4Defs.RepForType[iType]]];
    VPop[];  VPush[P4Defs.BiasForType[cType], P4Defs.RepForType[cType]];
    IF ~(TreeLiteral[son2] AND StructuredLiteral[son1])
      THEN  val ← TreeLink[subtree[index: node]]
      ELSE
        BEGIN
        val ← P4Defs.UnpackElement[son1, TreeLiteralValue[son2]];
        freenode[node];
        END;
    RETURN
    END;


  AddOp: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    op: NodeName = (tb+node).name;
    type: CSEIndex = (tb+node).info;
    bias, shift: INTEGER;
    rep: Repr;
    son1 ← Exp[son1, target];  son2 ← Exp[son2, target];
    val ← TreeLink[subtree[index: node]];
    bias ← vStack[vI-1].bias;  shift ← vStack[vI].bias;
    rep ← CommonRep[vStack[vI-1].rep, vStack[vI].rep];
    IF rep = none THEN rep ← target;
    IF ~attr1 THEN  attr2 ← (IF rep=both THEN target ELSE rep) = unsigned;
    SELECT TRUE FROM
      TreeLiteral[son2] =>
        BEGIN  val ← son1;
        shift ← shift + TreeLiteralValue[son2];
        son1 ← empty;  freenode[node];
        END;
      (op = plus AND TreeLiteral[son1]) =>
        BEGIN  val ← son2;
        shift ← shift + TreeLiteralValue[son1];
        son2 ← empty;  freenode[node];
        END;
      ENDCASE;
    bias ← bias + (IF op = plus THEN shift ELSE -shift);
    VPop[];  VPop[];
    IF ~TreeLiteral[val]
      THEN
        BEGIN
        SELECT rep FROM
          both =>  rep ← IF target = none THEN signed ELSE target;
          none =>
            BEGIN
            ErrorDefs.WarningTree[mixedRepresentation, val];  rep ← both;
            END;
          ENDCASE =>  NULL;
        END;
    VPush[bias, rep];
    IF type # dataPtr.typeINTEGER AND OperandType[val] # type
      THEN val ← ForceType[val, type];
    RETURN
```

```
    END;

  Mult: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    const1, const2: BOOLEAN;
    rep: Repr;
    v, v1, v2: WORD;
    bias: INTEGER;
    t: TreeLink;
    const1 ← TreeLiteral[son1 ← Exp[son1, target]];
    const2 ← TreeLiteral[son2 ← Exp[son2, target]];
    val ← TreeLink[subtree[index: node]];
    IF const1 OR ~const2
      THEN  son1 ← AdjustBias[son1, -vStack[vI-1].bias];
    IF ~const1 OR const2
      THEN  son2 ← AdjustBias[son2, -vStack[vI].bias];
    rep ← CommonRep[vStack[vI-1].rep, vStack[vI].rep];
    IF const1 THEN v1 ← TreeLiteralValue[son1];
    IF const2 THEN v2 ← TreeLiteralValue[son2];
    IF const1 AND const2
      THEN
        BEGIN  bias ← 0;  v ← v1*v2;
        val ← MakeTreeLiteral[v];  freenode[node];
        rep ← LiteralRep[v, rep];
        END
      ELSE
        BEGIN
        SELECT rep FROM
          both =>  rep ← IF target = none THEN signed ELSE target;
          none =>  IF target # none THEN  rep ← target;
          ENDCASE =>  NULL;
        IF ~attr1 THEN  attr2 ← rep = unsigned;
        bias ← SELECT TRUE FROM
          const1 => v1*vStack[vI].bias,
          const2 => vStack[vI-1].bias*v2,
          ENDCASE => 0;
        IF const1 -- AND ~const2
          THEN  BEGIN  t ← son2;  son2 ← son1;  son1 ← t  END;
        IF const1 OR const2
          THEN
            SELECT (IF const1 THEN v1 ELSE v2) FROM
              0 =>
                BEGIN
                val ← son2; son2 ← empty; freenode[node]; rep ← both;
                END;
              1 =>
                BEGIN
                val ← son1;  son1 ← empty;  freenode[node];
                rep ← vStack[IF const1 THEN vI ELSE vI-1].rep;
                END;
              -1 =>
                BEGIN  mlpush[son1];  son1 ← empty;  freenode[node];
                pushtree[uminus, 1];  setinfo[dataPtr.typeINTEGER];
                setattr[1, FALSE];  setattr[2, FALSE];
                val ← mlpop[];
                END;
            ENDCASE;
        IF rep = none
          THEN
            BEGIN
            ErrorDefs.WarningTree[mixedRepresentation, val];  rep ← both;
            END;
        END;
    VPop[];  VPop[];  VPush[bias, rep];
    RETURN
    END;

  DivMod: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    v, v1, v2: WORD;
    rep: Repr;
    son1 ← RValue[son1, 0, target];  son2 ← RValue[son2, 0, target];
    val ← TreeLink[subtree[index: node]];
    rep ← CommonRep[vStack[vI-1].rep, vStack[vI].rep];
    IF rep = none
      THEN
```

```
      IF target # none
        THEN  rep ← target
        ELSE
          BEGIN
          ErrorDefs.errortree[mixedRepresentation, val];  rep ← both;
          END;
    IF ~TreeLiteral[son1] OR ~TreeLiteral[son2]
      THEN
        BEGIN
        IF ~attr1 THEN  attr2 ← CommonRep[rep, unsigned] # none;
        IF name = div AND TreeLiteral[son2]
          THEN
            SELECT TreeLiteralValue[son2] FROM
              = 1 =>
                BEGIN  val ← son1;  son1 ← empty;  freenode[node];  END;
              >=2 =>  IF rep = unsigned THEN rep ← both;
              ENDCASE;
        END
      ELSE
        BEGIN
        v1 ← TreeLiteralValue[son1];  v2 ← TreeLiteralValue[son2];
        IF CommonRep[rep, unsigned] = none
          THEN
            SELECT name FROM
              div =>  v ← LOOPHOLE[v1, INTEGER] / LOOPHOLE[v2, INTEGER];
              mod =>  v ← LOOPHOLE[v1, INTEGER] MOD LOOPHOLE[v2, INTEGER];
              ENDCASE =>  ERROR
          ELSE
            SELECT name FROM
              div =>  v ← v1 / v2;
              mod =>  v ← v1 MOD v2;
              ENDCASE =>  ERROR;
        val ← MakeTreeLiteral[v];  freenode[node];
        rep ← LiteralRep[v, rep];
        END;
    VPop[];  VPop[];  VPush[0, rep];  RETURN
    END;


  RelOp: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    rep, rep1, rep2: Repr;
    d1, d2: INTEGER;
    lit1, lit2: BOOLEAN;
    v1, v2: WORD;
    v: INTEGER;
    uc, b: BOOLEAN;
    ZeroWarning: ARRAY NodeName [relE..relLE] OF [0..2] = [0, 0, 2, 2, 1, 1];
    son1 ← Exp[son1, none];  rep1 ← vStack[vI].rep;  d1 ← vStack[vI].bias;
    son2 ← Exp[son2, none];  rep2 ← vStack[vI].rep;  d2 ← vStack[vI].bias;
    val ← TreeLink[subtree[index: node]];
    IF ~P4Defs.ComparableRanges[OperandType[son1], OperandType[son2]]
      THEN ErrorDefs.errortree[sizeClash, son2];
    SELECT name FROM
      relE, relN => uc ← FALSE;
      ENDCASE =>
        BEGIN
        IF rep1 = unsigned OR rep2 = unsigned
          THEN
            BEGIN
            son1 ← AdjustBias[son1, -d1];
            son2 ← AdjustBias[son2, -d2];
            d1 ← d2 ← 0;
            END;
        uc ← CommonRep[CommonRep[rep1,rep2], unsigned] # none;
        END;
    IF d1 # d2
      THEN
        IF (~uc AND TreeLiteral[son2]) OR (uc AND d2 > d1)
          THEN  son2 ← AdjustBias[son2, d1-d2]
          ELSE  son1 ← AdjustBias[son1, d2-d1];
    rep ← CommonRep[rep1, rep2];
    IF rep = none
      THEN
        SELECT name FROM
          relE, relN =>  ErrorDefs.WarningTree[mixedRepresentation, val];
```

```
        ENDCASE => ErrorDefs.errortree[mixedRepresentation, val];
    IF (lit1 ← TreeLiteral[son1]) THEN v1 ← TreeLiteralValue[son1];
    IF (lit2 ← TreeLiteral[son2]) THEN v2 ← TreeLiteralValue[son2];
    IF CommonRep[rep, unsigned] # none
      THEN
        BEGIN
        SELECT ZeroWarning[name] FROM
          1 => IF lit1 AND v1 = 0 THEN GO TO warn;
          2 => IF lit2 AND v2 = 0 THEN GO TO warn;
          ENDCASE;
        EXITS
          warn => ErrorDefs.WarningTree[unsignedCompare, val];
        END;
    IF ~lit1 OR ~lit2
      THEN BEGIN IF ~attr1 THEN attr2 ← CommonRep[rep, signed] = none END
      ELSE
        BEGIN
        v ← Compare[v1, v2, rep];
        SELECT name FROM
          relE => b ← v = 0;
          relN => b ← v # 0;
          relL => b ← v < 0;
          relGE => b ← v >= 0;
          relG => b ← v > 0;
          relLE => b ← v <= 0;
          ENDCASE => ERROR;
        val ← IF b THEN passPtr.tTRUE ELSE passPtr.tFALSE;
        freenode[node];
        END;
    VPop[]; VPop[]; VPush[0, both];
    RETURN
    END;

  In: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN OPEN (tb+node);
    bias: INTEGER;
    rep: Repr;
    void: BOOLEAN;
    son1 ← Exp[son1, none];
    bias ← vStack[vI].bias; rep ← vStack[vI].rep; VPop[];
--  IF rep = unsigned THEN
      BEGIN son1 ← AdjustBias[son1, -bias]; bias ← 0 END;
    void ← FALSE; val ← TreeLink[subtree[index: node]];
    son2 ← Range[son2, bias, rep, none
     !MixedRepresentation =>
        BEGIN ErrorDefs.errortree[mixedRepresentation, val]; RESUME END;
      EmptyInterval => BEGIN void ← TRUE; RESUME END];
    IF void AND son1 # empty
      THEN BEGIN freenode[node]; val ← passPtr.tFALSE END;
    VPop[]; VPush[0, both]; RETURN
    END;


  Range: PUBLIC PROCEDURE [t: TreeLink, bias: INTEGER, rep, target: Repr]
      RETURNS [val: TreeLink] =
    BEGIN
    node: TreeIndex;
    lBound: INTEGER;
    val ← t;
      DO
      WITH val SELECT FROM
        symbol =>
          BEGIN lBound ← P4Defs.BiasForType[UnderType[index]];
          THROUGH [1..2]
            DO
            mlpush[MakeTreeLiteral[ABS[lBound]]];
            IF lBound < 0 THEN pushtree[uminus, 1];
            ENDLOOP;
          mlpush[MakeTreeLiteral[Cardinality[index] - 1]];
          pushtree[plus, 2]; setinfo[dataPtr.typeINTEGER];
          val ← maketree[intCC, 2];
          END;
        subtree =>
          BEGIN node ← index;
          SELECT (tb+node).name FROM
            subrangeTC, cdot =>
```

```
            BEGIN  val ← (tb+node).son2;
            (tb+node).son2 ← empty;  freenode[node];
            END;
          IN [intOO .. intCC] =>
            BEGIN
            IF Interval[node, bias, target].const
              THEN [] ← ConstantInterval[node];
            rep ← CommonRep[rep, vStack[vI].rep];
            IF rep = none THEN SIGNAL MixedRepresentation;
            IF ~(tb+node).attr1
              THEN  (tb+node).attr2 ← CommonRep[rep, signed] = none;
            EXIT
            END;
          ENDCASE => ERROR;
        END;
      ENDCASE => ERROR;
    ENDLOOP;
  RETURN
  END;

Interval: PUBLIC PROCEDURE [node: TreeIndex, bias: INTEGER, target: Repr]
    RETURNS [const: BOOLEAN] =
  BEGIN  OPEN (tb+node);
  rep: Repr;
  son1 ← RValue[son1, bias, target];
  son2 ← RValue[son2, bias, target];
  rep ← CommonRep[vStack[vI-1].rep, vStack[vI].rep];
  VPop[];  VPop[];  VPush[bias, rep];
  const ← TreeLiteral[son1] AND TreeLiteral[son2];  RETURN
  END;

EmptyInterval: PUBLIC SIGNAL = CODE;

ConstantInterval: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [origin, range: INTEGER] =
  BEGIN  OPEN (tb+node);
  uBound: INTEGER;
  rep: Repr;
  rep ← vStack[vI].rep;
  origin ← TreeLiteralValue[son1];  uBound ← TreeLiteralValue[son2];
  SELECT name FROM
    intOO, intOC =>
      BEGIN
      IF Compare[origin, uBound, rep] >= 0 THEN SIGNAL EmptyInterval;
      origin ← origin + 1;
      son1 ← freetree[son1];
      name ← IF name = intOO THEN intCO ELSE intCC;
      son1 ← MakeTreeLiteral[origin];
      END;
    ENDCASE;
  SELECT name FROM
    intCC =>  IF Compare[origin, uBound, rep] > 0 THEN SIGNAL EmptyInterval;
    intCO =>
      BEGIN
      IF Compare[origin, uBound, rep] >= 0 THEN SIGNAL EmptyInterval;
      uBound ← uBound - 1;
      son2 ← freetree[son2];
      name ← intCC;  son2 ← MakeTreeLiteral[uBound];
      END;
    ENDCASE => ERROR;
  range ← uBound - origin;  RETURN
  END;


BoolOp: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN  OPEN (tb+node);
  b: TreeLink = IF name = and THEN passPtr.tTRUE ELSE passPtr.tFALSE;
  son1 ← Exp[son1, none];  son2 ← Exp[son2, none];
  IF TreeLiteral[son1]
    THEN
      BEGIN
      IF son1 = b
        THEN  BEGIN  val ← son2;  son2 ← empty  END
        ELSE
          val ← IF b = passPtr.tTRUE THEN passPtr.tFALSE ELSE passPtr.tTRUE;
      freenode[node];
      END
```

```
    ELSE
      IF ~TreeLiteral[son2] OR son2 # b
        THEN  val ← TreeLink[subtree[index: node]]
        ELSE  BEGIN  val ← son1;  son1 ← empty;  freenode[node]  END;
  VPop[];  VPop[];  VPush[0, both];
  RETURN
  END;


IfExp: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [val: TreeLink] =
  BEGIN  OPEN (tb+node);
  select: TreeLink;
  rep: Repr;
  nw: CARDINAL = P4Defs.WordsForType[info];
  bias: INTEGER = P4Defs.BiasForType[info];
  son1 ← RValue[son1, 0, none];  VPop[];
  IF TreeLiteral[son1]
    THEN
      BEGIN
      IF son1 # passPtr.tFALSE
        THEN  BEGIN  select ← son2;  son2 ← empty  END
        ELSE  BEGIN  select ← son3;  son3 ← empty  END;
      freenode[node];
      val ← Exp[select, target];
      END
    ELSE
      BEGIN
      son2 ← RValue[son2, bias, target];  rep ← vStack[vI].rep;  VPop[];
      IF P4Defs.WordsForType[OperandType[son2]] # nw
        THEN  ErrorDefs.errortree[sizeClash, son2];
      son3 ← RValue[son3, bias, target];
      IF P4Defs.WordsForType[OperandType[son3]] # nw
        THEN  ErrorDefs.errortree[sizeClash, son3];
      rep ← CommonRep[vStack[vI].rep, rep];
      vStack[vI].rep ← IF rep = none THEN target ELSE rep;
      val ← TreeLink[subtree[index: node]];
      END;
  RETURN
  END;

CaseExp: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [val: TreeLink] =
  BEGIN
  type: CSEIndex = (tb+node).info;
  bias: INTEGER = P4Defs.BiasForType[type];
  rep: Repr;
  nw: CARDINAL = P4Defs.WordsForType[type];

  Selection: TreeMap =
    BEGIN
    v ← RValue[t, bias, target];
    rep ← CommonRep[rep, vStack[vI].rep];  VPop[];
    IF P4Defs.WordsForType[OperandType[v]] # nw
      THEN  ErrorDefs.errortree[sizeClash, v];
    RETURN
    END;

  rep ← both+other;
  val ← P4Defs.CaseDriver[node, Selection];
  VPush[bias, IF rep = none THEN target ELSE rep];
  RETURN
  END;

BindExp: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [TreeLink] =
  BEGIN

  BoundExp: TreeMap =
    BEGIN
    RETURN [CaseExp[GetNode[t], target]]
    END;

  RETURN [P4Defs.Binding[node, caseexp, BoundExp]]
  END;


MinMax: PROCEDURE [node: TreeIndex, target: Repr] RETURNS [val: TreeLink] =
  BEGIN  OPEN (tb+node);
```

```
    started, const, zeroTest: BOOLEAN;
    rep: Repr;
    m: WORD;

    Item: TreeMap =
      BEGIN
      n: WORD;
      v ← RValue[t, 0, target];
      IF ~TreeLiteral[v]
        THEN const ← FALSE
        ELSE
          BEGIN  n ← TreeLiteralValue[v];  IF n = 0 THEN zeroTest ← TRUE  END;
      rep ← CommonRep[rep, vStack[vI].rep];
      IF const
        THEN
          IF ~started
            THEN  BEGIN  started ← TRUE;  m ← n  END
            ELSE
              SELECT name FROM
                min =>  m ← IF Compare[m, n, rep] <= 0 THEN m ELSE n;
                max =>  m ← IF Compare[m, n, rep] >= 0 THEN m ELSE n;
                ENDCASE;
      VPop[];  RETURN
      END;

    IF listlength[son1] = 1
      THEN  BEGIN val ← Exp[son1, target]; son1 ← empty; freenode[node] END
      ELSE
        BEGIN
        started ← zeroTest ← FALSE;  const ← TRUE;  rep ← both+other;
        son1 ← updatelist[son1, Item];  val ← TreeLink[subtree[index: node]];
        IF zeroTest AND CommonRep[rep, unsigned] # none
          THEN  ErrorDefs.WarningTree[unsignedCompare, val];
        IF const
          THEN
            BEGIN  val ← MakeTreeLiteral[m];  freenode[node];
            rep ← LiteralRep[m, rep];
            END
          ELSE
            BEGIN
            SELECT rep FROM
              both =>  rep ← IF target = none THEN signed ELSE target;
              none =>
                IF target # none
                  THEN  rep ← target
                  ELSE
                    BEGIN
                    ErrorDefs.errortree[mixedRepresentation, val]; rep ← both;
                    END;
              ENDCASE => NULL;
            IF ~attr1 THEN  attr2 ← rep = unsigned;
            END;
        VPush[0, rep];
        END;
    RETURN
    END;


  Addr: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    subNode: TreeIndex;
    t: TreeLink;
    type: CSEIndex;
    son1 ← Exp[son1, none];  VPop[];
    t ← son1;
      DO
      WITH t SELECT FROM
        symbol =>
          BEGIN
          IF (ctxb+(seb+index).ctxnum).ctxlevel = SymDefs.1Z
            AND LOOPHOLE[(seb+index).idvalue, SymDefs.BitAddress].bd # 0
                THEN GO TO fail;
          GO TO pass;
          END;
        subtree =>
          BEGIN  subNode ← index;
```

```
        SELECT (tb+subNode).name FROM
          dot, dollar => t ← (tb+subNode).son2;
          index, dindex =>
            BEGIN type ← NormalType[OperandType[(tb+subNode).son1]];
              DO
              WITH (seb+type) SELECT FROM
                array => IF packed THEN GO TO fail ELSE GO TO pass;
                arraydesc => type ← UnderType[describedType];
                ENDCASE => ERROR;
              ENDLOOP;
            END;
          seqindex => GO TO fail;
          uparrow, reloc, memory => GO TO pass;
          cast => t ← (tb+subNode).son1;
          ENDCASE => ERROR;
        END;
      ENDCASE => ERROR;
    REPEAT
      pass => NULL;
      fail => ErrorDefs.errortree[nonAddressable, son1];
    ENDLOOP;
  val ← TreeLink[subtree[index: node]];
  IF testtree[son1, dot]
    THEN
      BEGIN subNode ← GetNode[son1];
      IF TreeLiteral[(tb+subNode).son1]
        THEN
          WITH (tb+subNode).son2 SELECT FROM
            symbol =>
              BEGIN
              val ← MakeStructuredLiteral[
                     TreeLiteralValue[(tb+subNode).son1] +
                      LOOPHOLE[(seb+index).idvalue, SymDefs.BitAddress].wd,
                    info];
              freenode[node];
              END;
            ENDCASE => ERROR;
      END;
  VPush[0, unsigned+other]; RETURN
  END;


Lengthen: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN OPEN (tb+node);
  v: ARRAY [0..2) OF WORD;
  son1 ← RValue[son1, 0, unsigned];
  attr1 ← SELECT TypeForm[OperandType[son1]] FROM
      pointer, arraydesc => TRUE,
      ENDCASE => FALSE;
  IF vStack[vI].rep = none
    THEN ErrorDefs.errortree[mixedRepresentation, son1];
  attr2 ← CommonRep[vStack[vI].rep, unsigned] # none; VPop[];
  IF ~TreeLiteral[son1]
    THEN val ← [subtree[index: node]]
    ELSE
      BEGIN
      v[0] ← TreeLiteralValue[son1];
      v[1] ← IF attr2 OR CommonRep[v[0], 1B5] = 0 THEN 0 ELSE 177777B;
      pushlittree[LitDefs.FindLitDescriptor[DESCRIPTOR[v]]];
      pushtree[mwconst, 1]; setinfo[info];
      val ← mlpop[]; freenode[node];
      END;
  VPush[0, other];
  END;

Loophole: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN OPEN (tb+node);
  type: CSEIndex = info;
  rep: Repr = P4Defs.RepForType[type];
  val ← Exp[son1, rep];
  IF son2 # empty
    THEN P4Defs.TypeExp[typeExp:son2, body:FALSE];
  IF P4Defs.WordsForType[OperandType[val]] # P4Defs.WordsForType[type]
    THEN ErrorDefs.errortree[sizeClash, son1];
  val ← IF TreeLiteral[val]
          THEN MakeStructuredLiteral[TreeLiteralValue[val], type]
```

```
        ELSE ForceType[val, type];
  son1 ← empty;  freenode[node];
  vStack[vI].rep ← rep;  RETURN
  END;


EndPoint: PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
  BEGIN  OPEN (tb+node);
  type: CSEIndex;
  first: BOOLEAN = (name=first);
  MaxInteger: INTEGER = AltoDefs.maxinteger;
  MaxWord: INTEGER = AltoDefs.maxword;
  v: WORD;
  vv: ARRAY [0..2) OF WORD;
  P4Defs.TypeExp[typeExp:son1, body:FALSE];
  type ← UnderType[P4Defs.TypeForTree[son1]];
    DO
    WITH (seb+type) SELECT FROM
      basic =>
        BEGIN
        v ← SELECT code FROM
          SymDefs.codeINTEGER => IF first THEN -MaxInteger-1 ELSE MaxInteger,
          SymDefs.codeCHARACTER => IF first THEN 0 ELSE AltoDefs.maxcharcode,
          ENDCASE => IF first THEN 0 ELSE MaxWord;
        GO TO short
        END;
      enumerated =>
        BEGIN
        v ← IF first THEN 0 ELSE Cardinality[type]-1;  GO TO short
        END;
      relative =>  type ← UnderType[offsetType];
      subrange =>
        BEGIN
        v ← IF first THEN origin ELSE origin+range;  GO TO short
        END;
      long =>
        BEGIN
        vv ← IF first THEN [0, -MaxInteger-1] ELSE [MaxWord, MaxInteger];
        GO TO long
        END;
      ENDCASE =>  ERROR;
    REPEAT
      short =>  val ← MakeTreeLiteral[v];
      long =>
        BEGIN
        pushlittree[LitDefs.FindLitDescriptor[DESCRIPTOR[vv]]];
        pushtree[mwconst, 1];  setinfo[type];  val ← mlpop[];
        END;
    ENDLOOP;
  freenode[node];  VPush[0, P4Defs.RepForType[type]];  RETURN
  END;


AdjustBias: PUBLIC PROCEDURE [t: TreeLink, delta: INTEGER] RETURNS [TreeLink] =
  BEGIN
  op: NodeName;
  type: CSEIndex;
  IF t = empty THEN passPtr.implicitBias ← passPtr.implicitBias + delta;
  IF delta = 0 THEN RETURN [t];
  type ← OperandType[t];
  IF TreeLiteral[t]
    THEN RETURN [MakeStructuredLiteral[TreeLiteralValue[t]-delta, type]];
  IF delta > 0
    THEN  op ← minus
    ELSE  BEGIN  op ← plus;  delta ← -delta  END;
  mlpush[t];  mlpush[MakeTreeLiteral[delta]];
  pushtree[op, 2];  setinfo[type];  setattr[1, FALSE];
  RETURN [mlpop[]]
  END;

RValue: PUBLIC PROCEDURE [exp: TreeLink, bias: INTEGER, target: Repr] RETURNS [val: TreeLink] =
  BEGIN
  d: INTEGER;
  val ← Exp[exp, target];  d ← bias - vStack[vI].bias;
  IF d # 0
    THEN  BEGIN  vStack[vI].bias ← bias;  val ← AdjustBias[val, d]  END;
```

```
    RETURN
    END;

  END.
```