```
-- file Pass4XA.Mesa
-- last written by Satterthwaite, July 16, 1978  11:02 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  ErrorDefs: FROM "errordefs",
  InlineDefs: FROM "inlinedefs",
  LitDefs: FROM "litdefs",
  P4Defs: FROM "p4defs",
  SymDefs: FROM "symdefs",
  SymTabDefs: FROM "symtabdefs",
  SystemDefs: FROM "systemdefs",
  TableDefs: FROM "tabledefs",
  TreeDefs: FROM "treedefs";

Pass4Xa: PROGRAM
    IMPORTS
        ErrorDefs, LitDefs, P4Defs, SymTabDefs, SystemDefs, TreeDefs
    EXPORTS P4Defs =
  BEGIN
  OPEN SymTabDefs, TreeDefs;

  Repr: TYPE = P4Defs.Repr;

 -- pervasive definitions from SymDefs

  SEIndex: TYPE = SymDefs.SEIndex;
  ISEIndex: TYPE = SymDefs.ISEIndex;
  CSEIndex: TYPE = SymDefs.CSEIndex;
  RecordSEIndex: TYPE = SymDefs.recordCSEIndex;
  ArraySEIndex: TYPE = SymDefs.arrayCSEIndex;

  BitAddress: TYPE = SymDefs.BitAddress;


  tb: TableDefs.TableBase;       -- tree base address (local copy)
  ltb: TableDefs.TableBase;      -- literal base address (local copy)
  seb: TableDefs.TableBase;      -- se table base address (local copy)
  ctxb: TableDefs.TableBase;     -- context table base address (local copy)

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


 -- expression list manipulation

  MakeRecord: PROCEDURE [record: RecordSEIndex, expList: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    sei: ISEIndex;
    const: BOOLEAN;
    subNode: TreeIndex;

    EvaluateField: TreeMap =
      BEGIN
      type: CSEIndex;
      IF t = empty
        THEN  BEGIN  v ← empty;  const ← FALSE  END
        ELSE
          BEGIN  type ← UnderType[(seb+sei).idtype];
          v ← P4Defs.RValue[t, P4Defs.BiasForType[type],
                TargetRep[P4Defs.RepForType[type]]];
          IF ~P4Defs.AssignableRanges[type, P4Defs.OperandType[v]]
            THEN  v ← ResolveSizes[v, type];
          IF ~P4Defs.TreeLiteral[v]
            THEN
              WITH v SELECT FROM
                subtree =>
                  SELECT (tb+index).name FROM
                    mwconst =>  NULL;
                    unionx => IF ~(tb+index).attr1 THEN const ← FALSE;
                    ENDCASE =>  const ← FALSE;
                ENDCASE =>  const ← FALSE;
          P4Defs.VPop[];
```

```
        END;
      sei ← NextSe[sei];
      RETURN
      END;

    IF ~testtree[expList, list]
      THEN
        BEGIN
        IF expList = empty
          THEN  pushproperlist[0]
          ELSE  BEGIN  mlpush[expList];  pushproperlist[1]  END;
        expList ← mlpop[];
        END;
    sei ← firstvisiblese[(seb+record).fieldctx];  const ← TRUE;
    val ← updatelist[expList, EvaluateField];
    subNode ← GetNode[val];  (tb+subNode).attr1 ← const;
    RETURN
    END;

  MakeArgRecord: PUBLIC PROCEDURE [record: RecordSEIndex, expList: TreeLink] RETURNS [val: TreeLink] =
    BEGIN
    type: CSEIndex;
    (seb+record).lengthUsed ← TRUE;
    SELECT listlength[expList] FROM
      0 =>  val ← empty;
      1 =>
        BEGIN
        type ← UnderType[(seb+firstvisiblese[(seb+record).fieldctx]).idtype];
        val ← P4Defs.RValue[expList, P4Defs.BiasForType[type],
                TargetRep[P4Defs.RepForType[type]]];
        IF ~P4Defs.AssignableRanges[type, P4Defs.OperandType[val]]
          THEN val ← ResolveSizes[val, type];
        P4Defs.VPop[];
        END;
      ENDCASE =>  val ← MakeRecord[record, expList];
    RETURN
    END;


  -- construction of packed values (machine dependent)

  WordLength: CARDINAL = AltoDefs.wordlength;
  ByteLength: CARDINAL = AltoDefs.charlength;

  FillMultiWord: PROCEDURE [words: DESCRIPTOR FOR ARRAY OF WORD,
        origin: CARDINAL, t: TreeLink] RETURNS [newOrigin: CARDINAL] =
    BEGIN
    desc: LitDefs.LitDescriptor;
    i: CARDINAL;
    WITH s: t SELECT FROM
      literal =>
        WITH s.info SELECT FROM
          word =>
            BEGIN
            desc ← LitDefs.LitDescriptorValue[index];
            FOR i IN [0 .. desc.length)
              DO words[origin + i] ← (ltb+desc.offset)[i]  ENDLOOP;
            END;
          ENDCASE => ERROR;
      ENDCASE => ERROR;
    RETURN [origin + desc.length]
    END;

  Masks: ARRAY [0..WordLength] OF WORD =
     [0B, 1B, 3B, 7B, 17B, 37B, 77B, 177B, 377B, 777B,
      1777B, 3777B, 7777B, 17777B, 37777B, 77777B, 177777B];

  PackRecord: PROCEDURE [record: RecordSEIndex, expList: TreeLink] RETURNS [TreeLink] =
    BEGIN
    n: CARDINAL = P4Defs.WordsForType[record];
    root, type: RecordSEIndex;
    list: TreeLink;
    sei: ISEIndex;
    offset: CARDINAL;
    words: DESCRIPTOR FOR ARRAY OF WORD;
    i: CARDINAL;
```

```
    more: BOOLEAN;

    StoreBits: PROCEDURE [sei: ISEIndex, value: WORD] =
      BEGIN
      OPEN InlineDefs;
      address: BitAddress;
      size, w, shift: CARDINAL;
      IF (seb+root).argument
        THEN  [address, size] ← FnField[sei]
        ELSE  BEGIN address ← (seb+sei).idvalue; size ← (seb+sei).idinfo END;
      w ← address.wd;
      shift ← (WordLength-offset) - (address.bd+size);
      words[w] ← BITOR[words[w], BITSHIFT[BITAND[value, Masks[size]], shift]];
      RETURN
      END;

    PackField: TreeScan =
      BEGIN
      node: TreeIndex;
      address: BitAddress;
      typeId: ISEIndex;
      subType: CSEIndex;
      IF P4Defs.TreeLiteral[t]
        THEN  StoreBits[sei, P4Defs.TreeLiteralValue[t]]
        ELSE
          BEGIN  node ← GetNode[t];
          SELECT (tb+node).name FROM
            mwconst =>
              BEGIN
              address ← IF (seb+root).argument
                THEN FnField[sei].offset
                ELSE (seb+sei).idvalue;
              [] ← FillMultiWord[words, address.wd, (tb+node).son1];
              END;
            unionx =>
              BEGIN
              WITH (tb+node).son1 SELECT FROM
                symbol =>  typeId ← index;
                ENDCASE =>  ERROR;
              subType ← UnderType[(seb+sei).idtype];
              WITH (seb+subType) SELECT FROM
                union =>
                  IF controlled THEN StoreBits[tagsei, (seb+typeId).idvalue];
                ENDCASE =>  ERROR;
              type ← LOOPHOLE[UnderType[typeId], RecordSEIndex];
              list ← (tb+node).son2;  more ← TRUE;
              END;
            ENDCASE =>  ERROR;
          END;
      sei ← NextSe[sei];  RETURN
      END;

    words ← DESCRIPTOR[SystemDefs.AllocateHeapNode[n], n];
    FOR i IN [0 .. n) DO words[i] ← 0 ENDLOOP;
    root ← type ← RecordRoot[record];
    offset ← IF (seb+record).length < WordLength
        THEN WordLength - (seb+record).length
        ELSE 0;
    list ← expList;  more ← TRUE;
    WHILE more
      DO
      more ← FALSE;
      sei ← firstvisiblese[(seb+type).fieldctx];
      scanlist[list, PackField];
      ENDLOOP;
    pushlittree[LitDefs.FindLitDescriptor[words]];
    pushtree[IF n=1 THEN cast ELSE mwconst, 1];  setinfo[record];
    SystemDefs.FreeHeapNode[BASE[words]];
    RETURN [mlpop[]]
    END;


  LitFromTree: PROCEDURE [t: TreeLink] RETURNS [LitDefs.LitDescriptor] =
    BEGIN
    node: TreeIndex;
      DO
```

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

  ExtractValue: PROCEDURE [t: TreeLink, addr: BitAddress, size: CARDINAL, type: CSEIndex]
      RETURNS [val: TreeLink] =
    BEGIN
    words: DESCRIPTOR FOR ARRAY OF WORD;
    i: CARDINAL;
    desc: LitDefs.LitDescriptor = LitFromTree[t];
    n: CARDINAL = size/WordLength;
    IF n > 1
      THEN
        BEGIN
        IF addr.bd # 0 THEN ErrorDefs.error[unimplemented];
        words ← DESCRIPTOR[SystemDefs.AllocateHeapNode[n], n];
        FOR i IN [0 .. n) DO words[i] ← (ltb+desc.offset)[addr.wd+i] ENDLOOP;
        pushlittree[LitDefs.FindLitDescriptor[words]];
        pushtree[mwconst, 1];  setinfo[type];
        SystemDefs.FreeHeapNode[BASE[words]];
        val ← mlpop[];
        END
      ELSE
        val ← P4Defs.MakeStructuredLiteral[
                InlineDefs.BITSHIFT[
                  InlineDefs.BITSHIFT[(ltb+desc.offset)[addr.wd], addr.bd],
                  -(WordLength - size)],
                type];
    RETURN
    END;


  UnpackField: PUBLIC PROCEDURE [t: TreeLink, field: ISEIndex] RETURNS [val: TreeLink] =
    BEGIN
    rType: CSEIndex = P4Defs.OperandType[t];
    vType: CSEIndex = UnderType[(seb+field).idtype];
    addr: BitAddress;
    addr ← (seb+field).idvalue;
    WITH r: (seb+rType) SELECT FROM
      record =>
        IF r.length < WordLength
          THEN  addr.bd ← addr.bd + (WordLength - r.length);
      ENDCASE => ERROR;
    RETURN [ExtractValue[t, addr, (seb+field).idinfo, vType]]
    END;

  UnpackElement: PUBLIC PROCEDURE [t: TreeLink, i: CARDINAL] RETURNS [val: TreeLink] =
    BEGIN
    aType: CSEIndex = P4Defs.OperandType[t];
    cType: CSEIndex;
    addr: BitAddress;
    nB: CARDINAL;
    BytesPerWord: CARDINAL = WordLength/ByteLength;
    WITH a: (seb+aType) SELECT FROM
      array =>
        BEGIN
        cType ← UnderType[a.componenttype];  nB ← P4Defs.BitsForType[cType];
        IF nB > ByteLength OR ~a.packed
          THEN
            BEGIN
            addr ← [wd:i, bd:0];  nB ← MAX[nB, WordLength];
            END
          ELSE
            BEGIN
```

```
            addr ← [wd:i/BytesPerWord, bd:(i MOD BytesPerWord)*ByteLength];
            nB ← ByteLength;
            END;
        END;
      ENDCASE => ERROR;
    RETURN [ExtractValue[t, addr, nB, cType]]
    END;


-- operators

  Call: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [TreeLink] =
    BEGIN  OPEN (tb+node);
    type: CSEIndex;
    son1 ← P4Defs.Exp[son1, P4Defs.none];  P4Defs.VPop[];
    type ← P4Defs.OperandType[son1];
    WITH (seb+type) SELECT FROM
      transfer =>
        BEGIN
        son2 ← MakeArgRecord[inrecord, son2];
        P4Defs.VPush[P4Defs.BiasForType[outrecord], P4Defs.RepForType[outrecord]];
        END;
      ENDCASE =>  ERROR;
    IF nsons > 2 THEN  P4Defs.CatchNest[son3];
    RETURN [TreeLink[subtree[index: node]]]
    END;

  New: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [TreeLink] =
    BEGIN  OPEN (tb+node);
    son1 ← P4Defs.NeutralExp[son1];
    IF nsons > 2 THEN P4Defs.CatchNest[son3];
    P4Defs.VPush[0, P4Defs.unsigned];
    RETURN [TreeLink[subtree[index: node]]]
    END;

  Fork: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [TreeLink] =
    BEGIN  OPEN (tb+node);
    type: CSEIndex;
    son1 ← P4Defs.Exp[son1, P4Defs.none];  P4Defs.VPop[];
    type ← P4Defs.OperandType[son1];
    WITH (seb+type) SELECT FROM
      transfer =>
        BEGIN
        son2 ← MakeArgRecord[inrecord, son2];
        P4Defs.VPush[0, P4Defs.other];
        END;
      ENDCASE =>  ERROR;
    IF nsons > 2 THEN  P4Defs.CatchNest[son3];
    RETURN [TreeLink[subtree[index: node]]]
    END;



  Construct: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    type: RecordSEIndex = info;
    record: RecordSEIndex = RecordRoot[type];
    subType: CSEIndex;
    subNode: TreeIndex;
    IF listlength[son2] = 1 AND ~testtree[son2, list] AND ~(seb+record).variant
      THEN
        BEGIN
        subType ← UnderType[(seb+firstvisiblese[(seb+type).fieldctx]).idtype];
        son2 ← P4Defs.RValue[son2, P4Defs.BiasForType[subType],
                TargetRep[P4Defs.RepForType[subType]]];
        IF ~P4Defs.AssignableRanges[subType, P4Defs.OperandType[son2]]
          THEN ErrorDefs.errortree[sizeClash, son2];
        val ← P4Defs.ForceType[son2, type];
        son2 ← empty;  freenode[node];
        END
      ELSE
        BEGIN
        subNode ← GetNode[son2 ← MakeRecord[record, son2]];
        IF (tb+subNode).attr1   -- all fields constant
          THEN  BEGIN  val ← PackRecord[type, son2];  freenode[node]  END
```

```
        ELSE
          BEGIN  son1 ← freetree[son1];
          pushtree[temp, 0];  setinfo[type];  setattr[1, FALSE];
          (seb+record).lengthUsed ← TRUE;
          son1 ← mlpop[];  val ← TreeLink[subtree[index: node]];
          END;
        P4Defs.VPush[0, P4Defs.other];
        END;
    RETURN
    END;

  Union: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [TreeLink] =
    BEGIN  OPEN (tb+node);
    type: RecordSEIndex;
    tSei: CSEIndex = UnderType[info];
    WITH son1 SELECT FROM
      symbol =>  type ← LOOPHOLE[UnderType[index], RecordSEIndex];
      ENDCASE =>  ERROR;
    son2 ← MakeRecord[type, son2];  (seb+type).lengthUsed ← TRUE;
    attr1 ← WITH son2 SELECT FROM
              subtree => (tb+index).attr1,
              ENDCASE => FALSE;
    attr2 ← WITH (seb+tSei) SELECT FROM
              union => controlled,
              ENDCASE => FALSE;
    P4Defs.VPush[0, P4Defs.other];
    RETURN [TreeLink[subtree[index: node]]]
    END;


  RowConstruct: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [val: TreeLink] =
    BEGIN  OPEN (tb+node);
    aType: ArraySEIndex = info;
    cType: CSEIndex = UnderType[(seb+aType).componenttype];
    n: CARDINAL = Cardinality[(seb+aType).indextype];
    cBias: INTEGER = P4Defs.BiasForType[cType];
    cRep: Repr = TargetRep[P4Defs.RepForType[cType]];
    const, strings, lstrings: BOOLEAN;
    l: CARDINAL;

    EvalElement: TreeMap =
      BEGIN
      node: TreeIndex;
      IF t = empty
        THEN  BEGIN  v ← empty;  const ← strings ← lstrings ← FALSE  END
        ELSE
          BEGIN
          v ← P4Defs.RValue[t, cBias, cRep];
          IF ~P4Defs.AssignableRanges[cType, P4Defs.OperandType[v]]
            THEN  v ← ResolveSizes[v, cType];
          IF P4Defs.TreeLiteral[v]
            THEN  strings ← lstrings ← FALSE
            ELSE
              WITH v SELECT FROM
                subtree =>
                  BEGIN  node ← index;
                  SELECT (tb+node).name FROM
                    mwconst =>  strings ← lstrings ← FALSE;
--                  rowconsx =>
--                    BEGIN  const ← FALSE;
--                    IF ~(tb+node).attr1 THEN strings ← FALSE;
--                    END;
                  ENDCASE =>  const ← strings ← lstrings ← FALSE;
                  END;
                literal =>
                  WITH info SELECT FROM
                    string =>
                      BEGIN   const ← FALSE;
                      IF LitDefs.MasterString[index] = index
                        THEN lstrings ← FALSE
                        ELSE strings ← FALSE;
                      END;
                    ENDCASE;
                ENDCASE =>  const ← strings ← lstrings ← FALSE;
          P4Defs.VPop[];
          END;
```

```
    RETURN
    END;

  w, nW: CARDINAL;
  words: DESCRIPTOR FOR ARRAY OF WORD;
  bitsLeft: CARDINAL;
  bitCount: CARDINAL;

  PackElement: TreeScan =
    BEGIN
    node: TreeIndex;
    IF P4Defs.TreeLiteral[t]
      THEN
        BEGIN
        bitsLeft ← bitsLeft - bitCount;
        words[w] ← InlineDefs.BITOR[words[w],
                InlineDefs.BITSHIFT[P4Defs.TreeLiteralValue[t], bitsLeft]];
        IF bitsLeft < bitCount
          THEN  BEGIN  w ← w+1;  bitsLeft ← WordLength  END;
        END
      ELSE
        BEGIN  node ← GetNode[t];
        SELECT (tb+node).name FROM
          mwconst =>  w ← FillMultiWord[words, w, (tb+node).son1];
          ENDCASE =>  ERROR;
        END;
    RETURN
    END;

  SELECT (l ← listlength[son2]) FROM
    = n => NULL;
    > n => ErrorDefs.errorn[listLong, l-n];
    < n => ErrorDefs.errorn[listShort, n-l];
    ENDCASE;
  const ← strings ← lstrings ← TRUE;
  son2 ← updatelist[son2, EvalElement];
  IF const AND l = n
    THEN
      BEGIN
      nW ← P4Defs.WordsForType[aType];
      words ← DESCRIPTOR[SystemDefs.AllocateHeapNode[nW], nW];
      FOR w IN [0 .. nW) DO words[w] ← 0 ENDLOOP;
      bitCount ←
        IF (seb+aType).packed AND P4Defs.BitsForType[cType] <= ByteLength
          THEN ByteLength ELSE WordLength;
      w ← 0;  bitsLeft ← WordLength;
      scanlist[son2, PackElement];  freenode[node];
      pushlittree[LitDefs.FindLitDescriptor[words]];
      pushtree[IF nW = 1 THEN cast ELSE mwconst, 1];  setinfo[aType];
      SystemDefs.FreeHeapNode[BASE[words]];
      val ← mlpop[];
      END
    ELSE
      BEGIN  attr1 ← strings # lstrings;
      son1 ← freetree[son1];
      pushtree[temp, 0]; setinfo[aType]; setattr[1, FALSE];  son1 ← mlpop[];
      (seb+aType).lengthUsed ← TRUE;
      val ← [subtree[index: node]];
      END;
  P4Defs.VPush[0, P4Defs.other];  RETURN
  END;


Assignment: PUBLIC PROCEDURE [node: TreeIndex] RETURNS [TreeLink] =
  BEGIN  OPEN (tb+node);
  son1 ← P4Defs.Exp[son1, P4Defs.none];
  son2 ← P4Defs.RValue[son2, P4Defs.VBias[], TargetRep[P4Defs.VRep[]]];
  P4Defs.VPop[];
  RETURN [RewriteAssign[TreeLink[subtree[index: node]]]]
  END;

TargetRep: PUBLIC PROCEDURE [rep: Repr] RETURNS [Repr] =
  BEGIN
  RETURN [IF rep = P4Defs.both THEN P4Defs.unsigned ELSE rep]
  END;
```

```
PushAssignment: PUBLIC PROCEDURE [id, val: TreeLink, type: CSEIndex] =
  BEGIN
  node: TreeIndex;
  rewrite: BOOLEAN;
  rewrite ← TRUE;
  WITH val SELECT FROM
    subtree =>
      BEGIN  node ← index;
      SELECT (tb+node).name FROM
        body, signalinit =>  rewrite ← FALSE;
        align =>
          BEGIN  val ← (tb+node).son1;
          (tb+node).son1 ← empty;  freenode[node];
          END;
        ENDCASE => NULL;
      END;
    ENDCASE => NULL;
  scanlist[id, mlpush];  mlpush[val];
  THROUGH [1 .. listlength[id])
    DO
    pushtree[assignx, 2];  setinfo[type];
    IF rewrite
      THEN mlpush[RewriteAssign[mlpop[]]]
      ELSE setattr[1, FALSE];
    ENDLOOP;
  pushtree[assign, 2];
  IF rewrite
    THEN mlpush[RewriteAssign[mlpop[]]]
    ELSE setattr[1, FALSE];
  RETURN
  END;


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

ResolveSizes: PUBLIC PROCEDURE [t: TreeLink, type: CSEIndex] RETURNS [val: TreeLink] =
  BEGIN
  SELECT (seb+type).typetag FROM
    record =>
      BEGIN
      mlpush[t];  pushtree[align, 1];  setinfo[type];  val ← mlpop[];
      END;
    union =>  val ← t;
    ENDCASE =>  BEGIN  val ← t;  ErrorDefs.errortree[sizeClash, t]  END;
  RETURN
  END;


RewriteAssign: PROCEDURE [t: TreeLink] RETURNS [TreeLink] =
  BEGIN
  node, subNode: TreeIndex;
  lType, rType, type: CSEIndex;
  leftAlign, rightAlign: BOOLEAN;
  node ← GetNode[t];
  lType ← P4Defs.OperandType[(tb+node).son1];
  rType ← P4Defs.OperandType[(tb+node).son2];
  (tb+node).attr1 ← leftAlign ← rightAlign ← FALSE;
  IF ~P4Defs.AssignableRanges[lType, rType]
    THEN
```

```
      WITH l: (seb+lType) SELECT FROM
        record =>
          WITH r: (seb+rType) SELECT FROM
            record =>
              SELECT l.length FROM
                > r.length => leftAlign ← TRUE;
                < r.length => rightAlign ← TRUE;
                ENDCASE;
          ENDCASE =>  ErrorDefs.errortree[sizeClash, (tb+node).son2];
        union =>  NULL;
        ENDCASE =>  ErrorDefs.errortree[sizeClash, (tb+node).son2];
    IF (tb+node).son2 # empty
      THEN
        WITH (tb+node).son2 SELECT FROM
          subtree =>
            BEGIN  subNode ← index;
            SELECT (tb+subNode).name FROM
              constructx, rowconsx =>
                IF testtree[(tb+subNode).son1, temp]
                 AND ~LongPath[(tb+node).son1]
                  THEN
                    BEGIN
                    IF (tb+node).name = assign
                      THEN
                        (tb+subNode).name ←
                          IF (tb+subNode).name = constructx
                            THEN construct
                            ELSE rowcons;
                    [] ← freetree[(tb+subNode).son1];
                    (tb+subNode).son1 ← IF lType # SymDefs.typeANY
                      THEN (tb+node).son1
                      ELSE P4Defs.ForceType[(tb+node).son1, rType];
                    (tb+subNode).info ← (tb+node).info;
                    (tb+node).son1 ← (tb+node).son2 ← empty;
                    freenode[node];  node ← subNode;
                    leftAlign ← rightAlign ← FALSE;
                    END;
              unionx =>
                BEGIN
                subNode ← GetNode[(tb+node).son1];
                SELECT (tb+subNode).name FROM
                  dot =>
                    BEGIN  type ← P4Defs.OperandType[(tb+subNode).son1];
                    mlpush[(tb+subNode).son1];  pushtree[uparrow, 1];
                    setinfo[WITH (seb+type) SELECT FROM
                      pointer => UnderType[pointedtotype],
                      ENDCASE => SymDefs.typeANY];
                    (tb+subNode).son1 ← mlpop[];
                    (tb+subNode).name ← dollar;
                    END;
                  dollar =>  NULL;
                  ENDCASE =>  ERROR;
                (tb+node).name ← IF (tb+node).name = assignx
                  THEN vconstructx
                  ELSE vconstruct;
                leftAlign ← rightAlign ← FALSE;
                END;
              dot, dollar =>
                IF (seb+rType).typetag = union THEN (tb+node).attr1 ← TRUE;
              ENDCASE;
            END;
          ENDCASE => NULL;
    IF leftAlign
      THEN
        BEGIN
        mlpush[(tb+node).son1]; pushtree[align, 1]; setinfo[rType];
        (tb+node).son1 ← mlpop[];
        END;
    IF rightAlign
      THEN
        BEGIN
        mlpush[(tb+node).son2]; pushtree[align, 1]; setinfo[lType];
        (tb+node).son2 ← mlpop[];
        END;
    IF (tb+node).name = assignx
      THEN (tb+node).info ← IF rightAlign THEN lType ELSE rType;
```

```
    RETURN [TreeLink[subtree[index: node]]]
    END;

  END.
```