```
-- file Pass4L.Mesa
-- last modified by Satterthwaite, July 17, 1978  4:22 PM

DIRECTORY
  AltoDefs: FROM "altodefs",
  ComData: FROM "comdata",
  CompilerDefs: FROM "compilerdefs",
  ControlDefs: FROM "controldefs",
  ErrorDefs: FROM "errordefs",
  P4Defs: FROM "p4defs",
  SymDefs: FROM "symdefs",
  SymTabDefs: FROM "symtabdefs",
  SystemDefs: FROM "systemdefs",
  TableDefs: FROM "tabledefs",
  TreeDefs: FROM "treedefs";

Pass4L: PROGRAM
    IMPORTS
        CompilerDefs, ErrorDefs, SymTabDefs, SystemDefs, TreeDefs,
        dataPtr: ComData
    EXPORTS P4Defs =
  BEGIN
  OPEN SymTabDefs, SymDefs;

  tb: TableDefs.TableBase;      -- tree base (local copy)
  seb: TableDefs.TableBase;     -- se table base (local copy)
  ctxb: TableDefs.TableBase;    -- context table base (local copy)
  bb: TableDefs.TableBase;      -- body table base (local copy)

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


 -- address assignment (machine sensitive and subject to change)

  WordLength: CARDINAL = AltoDefs.wordlength;
  WordFill: CARDINAL = WordLength-1;
  ByteLength: CARDINAL = AltoDefs.charlength;
  BytesPerWord: CARDINAL = WordLength/ByteLength;

  LocalOrigin: CARDINAL = ControlDefs.localbase*WordLength;
  GlobalOrigin: CARDINAL = ControlDefs.globalbase*WordLength;
  FrameLimit: CARDINAL = ControlDefs.MaxFrameSize*WordLength;

  EntryLimit: CARDINAL = ControlDefs.MaxNGfi * ControlDefs.EPRange;


  BitsForType: PUBLIC PROCEDURE [type: SEIndex] RETURNS [nBits: CARDINAL] =
    BEGIN  -- assumes (an attempt at) prior processing by P4declitem
    b, nW: CARDINAL;
    sei: CSEIndex ← UnderType[type];
    WITH (seb+sei) SELECT FROM
      basic =>  nBits ← length;
      pointer =>  nBits ← WordLength;
      transfer =>  nBits ← IF mode = port THEN 2*WordLength ELSE WordLength;
      arraydesc =>  nBits ← 2*WordLength;
      relative =>  nBits ← BitsForType[offsetType];
      long =>
        BEGIN
        nW ← (BitsForType[rangetype] + WordFill)/WordLength;
        nBits ← (nW + 1)*WordLength;
        END;
      real =>  nBits ← 2*WordLength;
      ENDCASE =>  -- processing of se entry must be complete
        BEGIN
        IF ~mark4
          THEN
            BEGIN  -- P4declitem has not been able to complete
            ErrorDefs.errorsei[typeLength,
                IF (seb+type).setag = id
                  THEN LOOPHOLE[type, ISEIndex]
                  ELSE ISENull];
```

```
          RETURN [0]
          END;
        WITH (seb+sei) SELECT FROM
          enumerated => nBits ← BitsForRange[Cardinality[sei]-1];
          record => BEGIN nBits ← length; lengthUsed ← TRUE END;
          array =>
            BEGIN
            b ← BitsForType[componenttype];
            nW ← IF packed AND b <= ByteLength
                THEN (Cardinality[indextype] + (BytesPerWord-1))/BytesPerWord
                ELSE Cardinality[indextype] * ((b + WordFill)/WordLength);
            IF nW > AltoDefs.maxword/WordLength
              THEN ErrorDefs.error[fieldSize];
            nBits ← nW*WordLength; lengthUsed ← TRUE;
            END;
          subrange =>
            nBits ← IF empty THEN 0 ELSE BitsForRange[Cardinality[sei]-1];
          ENDCASE => nBits ← 0;
        END;
    RETURN
    END;

  ArgLength: PUBLIC PROCEDURE [type: SEIndex] RETURNS [length: CARDINAL] =
    BEGIN
    sei: CSEIndex = UnderType[type];
    length ← 0;
    WITH (seb+sei) SELECT FROM
      transfer =>
        BEGIN
        IF inrecord # SENull THEN
          BEGIN
          length ← length + (seb+inrecord).length;
          (seb+inrecord).lengthUsed ← TRUE;
          END;
        IF outrecord # SENull THEN
          BEGIN
          length ← length + (seb+outrecord).length;
          (seb+outrecord).lengthUsed ← TRUE;
          END;
        END;
      definition => NULL;
      ENDCASE => ERROR;
    RETURN
    END;


-- profile utilities

  VarInfo: TYPE = RECORD [
    link: ISEIndex,
    nRefs: CARDINAL];

  Profile: TYPE = DESCRIPTOR FOR ARRAY OF VarInfo;

  AllocateProfile: PROCEDURE [n: INTEGER] RETURNS [profile: Profile] =
    BEGIN
    k: INTEGER;
    profile ← DESCRIPTOR [SystemDefs.AllocateHeapNode[n*SIZE[VarInfo]], n];
    FOR k IN [0 .. n) DO profile[k].link ← ISENull ENDLOOP;
    RETURN
    END;

  ReleaseProfile: PROCEDURE [profile: Profile] =
    BEGIN
    SystemDefs.FreeHeapNode[BASE[profile]];
    RETURN
    END;

  SortProfile: PROCEDURE [v: Profile] =
    BEGIN -- Shell sort --
    h, i, j: INTEGER;
    k: CARDINAL;
    t: VarInfo;
    h ← LENGTH [v];
      DO
      h ← h/2;
```

```
    FOR j IN [h .. LENGTH[v])
      DO
      i ← j-h;  k ← v[j].nRefs;  t ← v[j];
      WHILE k > v[i].nRefs
        DO
        v[i+h] ← v[i];
        IF (i ← i-h) < 0 THEN EXIT;
        ENDLOOP;
      v[i+h] ← t;
      ENDLOOP;
    IF h <=1 THEN EXIT;
    ENDLOOP;
  RETURN
  END;


-- entry point assignment

 GenBodies: PROCEDURE [root: BTIndex, proc: PROCEDURE [CBTIndex]] =
  BEGIN
  bti, next: BTIndex;
  FOR bti ← root, next UNTIL bti = BTNull
    DO
    WITH (bb+bti) SELECT FROM
      Callable => proc[LOOPHOLE[bti]];
      ENDCASE => NULL;
    IF (bb+bti).firstSon # BTNull
      THEN  next ← (bb+bti).firstSon
      ELSE
        DO
        next ← (bb+bti).link.index;
        IF next = BTNull OR (bb+bti).link.which # parent THEN EXIT;
        bti ← next;
        ENDLOOP;
    ENDLOOP;
  RETURN
  END;


 BodyRefs: PROCEDURE [bti: CBTIndex] RETURNS [count: CARDINAL] =
  BEGIN
  sei: ISEIndex;
  node: TreeDefs.TreeIndex;

  CountRefs: TreeDefs.TreeScan =
    BEGIN
    WITH t SELECT FROM
      symbol =>  count ← count + (seb+index).idinfo;
      ENDCASE =>  ERROR;
    RETURN
    END;

  count ← 0;
  sei ← (bb+bti).id;
  IF sei # SENull AND (bb+bti).nesting = Outer
    THEN
      BEGIN  node ← (seb+sei).idvalue;
      TreeDefs.scanlist[(tb+node).son1, CountRefs];
      END;
  RETURN
  END;


 AssignEntries: PUBLIC PROCEDURE [rootBti: BTIndex] =
  BEGIN
  i, j, k: INTEGER;
  profile: Profile;
  bti: CBTIndex;

  AssignSlot: PROCEDURE [bti: CBTIndex] =
    BEGIN
    IF (bb+bti).info.mark = Internal
      THEN
        BEGIN
        profile[k] ← VarInfo[
              link: LOOPHOLE[bti],
```

```
            nRefs: IF (bb+bti).nesting = Inner
                  THEN 0
                  ELSE BodyRefs[LOOPHOLE[bti]]];
        k ← k+1;
        END;
    RETURN
    END;

  IF MAX[dataPtr.nBodies, dataPtr.nSigCodes] > EntryLimit
    THEN ErrorDefs.error[bodyEntries];
  profile ← AllocateProfile[dataPtr.nBodies];
  k ← 0;  GenBodies[rootBti, AssignSlot];
  IF dataPtr.sort THEN SortProfile[profile];
  i ← 1;
  FOR j IN [0..LENGTH[profile])
    DO
    bti ← LOOPHOLE[profile[j].link];
    IF bti = dataPtr.mainBody
      THEN (bb+bti).entryIndex ← 0
      ELSE  BEGIN  (bb+bti).entryIndex ← i;  i ← i+1  END;
    ENDLOOP;
  ReleaseProfile[profile];  RETURN
  END;


-- frame layout

 VarScan: TYPE = PROCEDURE [sei: ISEIndex, output: BOOLEAN];

 GenCtxVars: PROCEDURE [ctx: CTXIndex, p: VarScan, output: BOOLEAN] =
   BEGIN
   sei: ISEIndex;
   IF ctx # CTXNull THEN
     FOR sei ← (ctxb+ctx).selist, NextSe[sei] UNTIL sei = SENull
       DO
       IF ~(seb+sei).constant THEN p[sei, output];
       ENDLOOP;
   RETURN
   END;

 GenBodyVars: PROCEDURE [bti: CBTIndex, p: VarScan] =
   BEGIN
   type: SEIndex = (bb+bti).ioType;
   WITH se: (seb+type) SELECT FROM
     constructor =>
       WITH se SELECT FROM
         transfer =>
           BEGIN
           IF inrecord # SENull
             THEN GenCtxVars[(seb+inrecord).fieldctx, p, FALSE];
           IF outrecord # SENull
             THEN GenCtxVars[(seb+outrecord).fieldctx, p, TRUE];
           END;
         ENDCASE;
     ENDCASE;
   GenCtxVars[(bb+bti).localCtx, p, FALSE];
   RETURN
   END;

 GenImportedVars: PROCEDURE [p: VarScan] =
   BEGIN
   sei: ISEIndex;
   type: CSEIndex;
   ctx: CTXIndex = dataPtr.importCtx;
   IF ctx # CTXNull THEN
     FOR sei ← (ctxb+ctx).selist, NextSe[sei] UNTIL sei = SENull
       DO
       IF ~(seb+sei).constant
         THEN p[sei, FALSE]
         ELSE
           BEGIN  type ← UnderType[(seb+sei).idtype];
           WITH (seb+type) SELECT FROM
             definition =>  GenCtxVars[defCtx, p, FALSE];
             ENDCASE;
           END;
       ENDLOOP;
```

```
    RETURN
    END;


  BumpArgRefs: PROCEDURE [rsei: recordCSEIndex] =
    BEGIN
    sei: ISEIndex;
--  node: TreeDefs.TreeIndex;
    saveIndex: CARDINAL = dataPtr.textIndex;
    IF rsei # SENull
      THEN
        FOR sei ← (ctxb+(seb+rsei).fieldctx).selist, NextSe[sei] UNTIL sei = SENull
          DO
--        IF (seb+sei).idinfo = 0 AND (seb+sei).htptr # HTNull
--          THEN
--            BEGIN  node ← LOOPHOLE[(seb+sei).idvalue];
--            dataPtr.textIndex ← (tb+node).info;
--            ErrorDefs.WarningSei[unusedId, sei];
--            END;
          (seb+sei).idinfo ← (seb+sei).idinfo + 1;
          ENDLOOP;
    dataPtr.textIndex ← saveIndex;  RETURN
    END;

  CheckArguments: PROCEDURE [bti: CBTIndex] =
    BEGIN
    bodyType: SEIndex = (bb+bti).ioType;
    node: TreeDefs.TreeIndex;
    WITH type: (seb+bodyType) SELECT FROM
      constructor =>
        WITH type SELECT FROM
          transfer =>
            BEGIN
            WITH (bb+bti).info SELECT FROM
              Internal => node ← bodyTree;
              ENDCASE => ERROR;
            BumpArgRefs[inrecord];
--          IF (tb+node).attr2 THEN BumpArgRefs[outrecord];    ** field?
            END;
          ENDCASE => ERROR;
      ENDCASE;
    RETURN
    END;


  LayoutLocals: PUBLIC PROCEDURE [bti: CBTIndex] RETURNS [length: CARDINAL] =
    BEGIN
    vProfile: Profile;
    vI: CARDINAL;

    CountVar: VarScan =
      BEGIN
      IF (seb+sei).htptr # HTNull OR ~output THEN vI ← vI + 1;
      RETURN
      END;

    InsertVar: VarScan =
      BEGIN
      saveIndex: CARDINAL = dataPtr.textIndex;
      node: TreeDefs.TreeIndex = LOOPHOLE[(seb+sei).idvalue];
      nW: CARDINAL;
      IF node # TreeDefs.nullTreeIndex
        THEN dataPtr.textIndex ← (tb+node).info;
      IF (seb+sei).htptr # HTNull OR ~output
        THEN
          BEGIN
          vProfile[vI] ← [link:sei, nRefs:(seb+sei).idinfo];  vI ← vI+1;
          END;
      IF (seb+sei).idinfo = 0 AND (seb+sei).htptr # HTNull
       AND ~output      -- suppress message for return record
        THEN ErrorDefs.WarningSei[unusedId, sei];
      nW ← (BitsForType[(seb+sei).idtype] + WordFill)/WordLength;
      (seb+sei).idinfo ← nW*WordLength;  (seb+sei).idvalue ← 0;
      dataPtr.textIndex ← saveIndex;  RETURN
      END;
```

```
    origin: CARDINAL;
    CheckArguments[bti];
    vI ← 0;  GenBodyVars[bti, CountVar];
    vProfile ← AllocateProfile[vI];
    vI ← 0;  GenBodyVars[bti, InsertVar];
    SortProfile[vProfile];
    origin ← IF (bb+bti).level = 1L
        THEN LocalOrigin
        ELSE LocalOrigin + WordLength;
    IF (seb+(bb+bti).ioType).setag = id
      THEN      -- must be a procedure type
        BEGIN
        IF origin = LocalOrigin
          THEN
            BEGIN       -- fill link word
            [] ← AssignVars[vProfile, LocalOrigin, LocalOrigin+WordLength];
            origin ← origin+WordLength;
            END;
        origin ← origin + ArgLength[(bb+bti).ioType];
        END;
    origin ← AssignVars[vProfile, origin, LocalOrigin + ControlDefs.localslots*WordLength];
    length ← AssignVars[vProfile, origin, FrameLimit];
    CheckFrameOverflow[vProfile];  ReleaseProfile[vProfile];
    IF (bb+bti).level > 1L
     AND length > ControlDefs.MaxSmallFrameSize*WordLength
      THEN ErrorDefs.errorsei[addressOverflow, (bb+bti).id];
    RETURN
    END;


  LayoutGlobals: PUBLIC PROCEDURE [bti: CBTIndex] RETURNS [length: CARDINAL] =
    BEGIN
    vProfile, xProfile: Profile;
    vI, xI: CARDINAL;

    CountVar: VarScan =
      BEGIN
      ctx: CTXIndex = (seb+sei).ctxnum;
      IF (ctxb+ctx).ctxType = imported OR ctx = dataPtr.importCtx
        THEN  xI ← xI + 1
        ELSE
          IF (seb+sei).htptr # HTNull OR ~output THEN  vI ← vI + 1;
      RETURN
      END;

    InsertVar: VarScan =
      BEGIN
      saveIndex: CARDINAL;
      ctx: CTXIndex = (seb+sei).ctxnum;
      node: TreeDefs.TreeIndex;
      nW: CARDINAL;
      IF (ctxb+ctx).ctxType = imported OR ctx = dataPtr.importCtx
        THEN
          BEGIN
          xProfile[xI] ← [link:sei, nRefs:(seb+sei).idinfo];  xI ← xI+1;
          IF (seb+sei).idinfo = 0 AND ~(seb+sei).public
            THEN ErrorDefs.WarningSei[unusedId, sei];
          (seb+sei).idinfo ←
            ((BitsForType[(seb+sei).idtype]+WordFill)/WordLength)*WordLength;
          END
        ELSE
          BEGIN  saveIndex ← dataPtr.textIndex;
          node ← LOOPHOLE[(seb+sei).idvalue];
          IF node # TreeDefs.nullTreeIndex
            THEN dataPtr.textIndex ← (tb+node).info;
          IF (seb+sei).htptr # HTNull OR ~output
            THEN
              BEGIN
              vProfile[vI] ← [link:sei, nRefs:(seb+sei).idinfo];  vI ← vI + 1;
              END;
          IF (seb+sei).idinfo = 0
           AND ~(seb+sei).public AND (seb+sei).htptr # HTNull
            THEN ErrorDefs.WarningSei[unusedId, sei];
          nW ← (BitsForType[(seb+sei).idtype] + WordFill)/WordLength;
          (seb+sei).idinfo ← nW*WordLength;  (seb+sei).idvalue ← 0;
          dataPtr.textIndex ← saveIndex;
```

```
        END;
      RETURN
      END;

    origin: CARDINAL;
    IF (seb+(bb+bti).ioType).setag = id THEN ERROR;
    CheckArguments[bti];
    vI ← xI ← 0;  GenBodyVars[bti, CountVar];  GenImportedVars[CountVar];
    vProfile ← AllocateProfile[vI];  xProfile ← AllocateProfile[xI];
    vI ← xI ← 0;  GenBodyVars[bti, InsertVar];  GenImportedVars[InsertVar];
    IF dataPtr.sort
      THEN  BEGIN  SortProfile[vProfile];  SortProfile[xProfile]  END;
    origin ← IF dataPtr.stopping THEN GlobalOrigin+WordLength ELSE GlobalOrigin;
    AssignXfers[xProfile, 0, 256*WordLength];
    origin ← AssignVars[vProfile, origin, FrameLimit];
    length ← MAX[origin, GlobalOrigin+WordLength];
    CheckFrameOverflow[vProfile];  ReleaseProfile[vProfile];
    CheckFrameOverflow[xProfile];  ReleaseProfile[xProfile];
    RETURN
    END;


  LayoutBlock: PUBLIC PROCEDURE [bti: BTIndex, origin: CARDINAL] RETURNS [length: CARDINAL] =
    BEGIN
    vProfile: Profile;
    vI: CARDINAL;

    CountVar: VarScan =
      BEGIN
      vI ← vI + 1;  RETURN
      END;

    InsertVar: VarScan =
      BEGIN
      saveIndex: CARDINAL = dataPtr.textIndex;
      node: TreeDefs.TreeIndex = LOOPHOLE[(seb+sei).idvalue];
      nW: CARDINAL;
      IF node # TreeDefs.nullTreeIndex
        THEN dataPtr.textIndex ← (tb+node).info;
      vProfile[vI] ← [link:sei, nRefs:(seb+sei).idinfo];  vI ← vI+1;
      IF (seb+sei).idinfo = 0 THEN ErrorDefs.WarningSei[unusedId, sei];
      nW ← (BitsForType[(seb+sei).idtype] + WordFill)/WordLength;
      (seb+sei).idinfo ← nW*WordLength;  (seb+sei).idvalue ← 0;
      dataPtr.textIndex ← saveIndex;  RETURN
      END;

    vI ← 0;  GenCtxVars[(bb+bti).localCtx, CountVar, FALSE];
    vProfile ← AllocateProfile[vI];
    vI ← 0;  GenCtxVars[(bb+bti).localCtx, InsertVar, FALSE];
    SortProfile[vProfile];
    length ← AssignVars[vProfile, origin, FrameLimit];
    CheckFrameOverflow[vProfile];  ReleaseProfile[vProfile];
    IF (bb+bti).level > 1L
     AND length > ControlDefs.MaxSmallFrameSize*WordLength
      THEN ErrorDefs.errorsei[addressOverflow, dataPtr.seAnon];
    RETURN
    END;


  LayoutInterface: PUBLIC PROCEDURE [bti: CBTIndex] RETURNS [nEntries: CARDINAL] =
    BEGIN
    sei: ISEIndex;
    epN: CARDINAL;
    nW: CARDINAL;
    node: TreeDefs.TreeIndex;
    saveIndex: CARDINAL;
    epN ← 0;
    FOR sei ← (ctxb+(bb+bti).localCtx).selist, NextSe[sei] UNTIL sei = SENull
      DO
      IF ~(seb+sei).constant
        THEN
          BEGIN  saveIndex ← dataPtr.textIndex;
          node ← LOOPHOLE[(seb+sei).idvalue];
          IF node # TreeDefs.nullTreeIndex
            THEN dataPtr.textIndex ← (tb+node).info;
          nW ← (BitsForType[(seb+sei).idtype] + WordFill)/WordLength;
```

```
        (seb+sei).idinfo ← nW*WordLength;
        SELECT XferMode[(seb+sei).idtype] FROM
          procedure, signal, error, program =>
            BEGIN  (seb+sei).linkSpace ← TRUE;
            (seb+sei).idvalue ← epN;  epN ← epN + 1;
            END;
          ENDCASE => (seb+sei).idvalue ← 0;
        dataPtr.textIndex ← saveIndex;
        END;
    ENDLOOP;
  IF (nEntries←epN) > EntryLimit THEN ErrorDefs.error[interfaceEntries];
  RETURN
  END;


CheckFrameOverflow: PROCEDURE [profile: Profile] =
  BEGIN
  i: INTEGER;
  FOR i IN [0 .. LENGTH[profile])
    DO
    IF profile[i].link # SENull
      THEN ErrorDefs.errorsei[addressOverflow, profile[i].link];
    ENDLOOP;
  RETURN
  END;


Align: PROCEDURE [offset: CARDINAL, type: SEIndex] RETURNS [CARDINAL] =
  BEGIN
  RETURN [SELECT XferMode[type] FROM
    port =>
      (offset+WordLength)/(4*WordLength)*(4*WordLength) + (2*WordLength),
    ENDCASE => offset]
  END;

AssignVars: PROCEDURE [profile: Profile, origin, limit: CARDINAL] RETURNS [CARDINAL] =
  BEGIN
  start, base, remainder, delta: CARDINAL;
  i, j, next: INTEGER;
  sei, t: ISEIndex;
  found, skips: BOOLEAN;
  size, nRefs: CARDINAL;
  next ← 0;  start ← origin;  remainder ← limit - origin;
  WHILE next < LENGTH[profile]
    DO
    i ← next;  found ← skips ← FALSE;
    WHILE ~found AND i < LENGTH[profile]
      DO
      sei ← profile[i].link;
      IF sei # SENull
        THEN
          BEGIN  OPEN (seb+sei);
          base ← Align[start, idtype];  delta ← base - start;
          IF idinfo + delta <= remainder
            THEN
              BEGIN  nRefs ← 0;  size ← 0;
              FOR j ← i+1, j+1 WHILE j < LENGTH[profile]
                DO
                t ← profile[j].link;
                IF t # SENull
                  THEN
                    BEGIN  size ← size +  (seb+t).idinfo;
                    IF size > (seb+sei).idinfo THEN EXIT;
                    nRefs ← nRefs + profile[j].nRefs;
                    END;
              ENDLOOP;
            IF nRefs <= profile[i].nRefs OR ~dataPtr.sort
              THEN
                BEGIN
                found ← TRUE;
                idvalue ← BitAddress[wd:base/WordLength, bd:0];
                mark4 ← TRUE;  profile[i].link ← ISENull;
                IF base # start AND dataPtr.sort
                  THEN  [] ← AssignVars[profile, start, base];
                start ← base + idinfo;
                remainder ← remainder - (idinfo+delta);
```

```
                END
              ELSE
                IF ~skips THEN  BEGIN  skips ← TRUE;  next ← i  END;
            END;
        END;
      i ← i+1;
      IF ~skips THEN next ← i;
      ENDLOOP;
    ENDLOOP;
  RETURN [start]
  END;

AssignXfers: PROCEDURE [profile: Profile, origin, limit: CARDINAL] =
  BEGIN
  nProcs: CARDINAL;
  next: CARDINAL;
  i, j: CARDINAL;
  sei: ISEIndex;
  t: VarInfo;
  i ← nProcs ← LENGTH[profile];
  UNTIL i = 0
    DO
    i ← i-1;
    IF XferMode[(seb+profile[i].link).idtype] # procedure
      THEN
        BEGIN
        nProcs ← nProcs-1;  t ← profile[i];
        FOR j IN [i..nProcs) DO profile[j] ← profile[j+1] ENDLOOP;
        profile[nProcs] ← t;
        END;
    ENDLOOP;
  -- the xfer frame fragment begins at origin
    dataPtr.linkBase ← origin/WordLength;
    CompilerDefs.AppendBCDWord[dataPtr.linkCount ← LENGTH[profile]];
  i ← LENGTH[profile];
  next ← MIN[origin + LENGTH[profile]*WordLength, limit];
  UNTIL i = 0 OR next = origin
    DO
    i ← i-1;  sei ← profile[i].link;  profile[i].link ← ISENull;
    next ← next - (seb+sei).idinfo;
    CompilerDefs.AppendBCDWord[(seb+sei).idvalue];
    (seb+sei).idvalue ← BitAddress[wd: next/WordLength, bd: 0];
    (seb+sei).linkSpace ← TRUE;
    ENDLOOP;
  RETURN
  END;


GenLocalProcs: PROCEDURE [firstBti: BTIndex, proc: PROCEDURE [CBTIndex]] =
  BEGIN
  bti: BTIndex;
  IF (bti ← firstBti) # BTNull
    THEN
      DO
      WITH body: (bb+bti) SELECT FROM
        Callable => proc[LOOPHOLE[bti]];
        ENDCASE => NULL;
      IF (bb+bti).link.which = parent THEN EXIT;
      bti ← (bb+bti).link.index;
      ENDLOOP;
  RETURN
  END;

AssignLocalDescriptors: PUBLIC PROCEDURE [first: BTIndex, origin: CARDINAL] RETURNS [CARDINAL] =
  BEGIN
  i, j, k, n: INTEGER;
  w: CARDINAL;
  profile: Profile;
  bti: CBTIndex;

  GenLocals: PROCEDURE [proc: PROCEDURE [CBTIndex]] =
    BEGIN
    bti: BTIndex;
    IF (bti ← first) # BTNull
      THEN
        DO
```

```
        WITH (bb+bti) SELECT FROM
          Callable => proc[LOOPHOLE[bti]];
          ENDCASE => NULL;
        IF (bb+bti).link.which = parent THEN EXIT;
        bti ← (bb+bti).link.index;
        ENDLOOP;
    RETURN
    END;

  CountLocal: PROCEDURE [bti: CBTIndex] =
    BEGIN
    IF (bb+bti).info.mark = Internal THEN n ← n+1;
    RETURN
    END;

  AssignLocal: PROCEDURE [bti: CBTIndex] =
    BEGIN
    IF (bb+bti).info.mark = Internal
      THEN
        BEGIN
        profile[k] ← VarInfo[link:LOOPHOLE[bti], nRefs:BodyRefs[bti]];
        k ← k+1;
        END;
    RETURN
    END;

  n ← 0;  GenLocals[CountLocal];
  profile ← AllocateProfile[n];
  k ← 0;  GenLocals[AssignLocal];
  SortProfile[profile];
  w ← (origin + WordFill)/WordLength;
  i ← 1;
  FOR j IN [0..LENGTH[profile])
    DO
    bti ← LOOPHOLE[profile[j].link];
    WITH (bb+bti) SELECT FROM
      Inner =>
        BEGIN -- align to 4n+2
        frameOffset ← w ← ((w+1)/4)*4 + 2;  w ← w+1;
        END;
      ENDCASE =>  ERROR;
    ENDLOOP;
  ReleaseProfile[profile];
  RETURN [w*WordLength]
  END;


-- parameter record  layout

 LayoutArgs: PUBLIC PROCEDURE [argRecord: recordCSEIndex, origin: CARDINAL, body: BOOLEAN]
    RETURNS [CARDINAL] =
  BEGIN
  w, nW: CARDINAL;
  ctx: CTXIndex;
  sei: ISEIndex;
  w ← origin;
  IF argRecord # SENull
    THEN
      BEGIN  ctx ← (seb+argRecord).fieldctx;
      FOR sei ← (ctxb+ctx).selist, NextSe[sei] UNTIL sei = SENull
        DO
        OPEN (seb+sei);
        nW ← (BitsForType[idtype] + WordFill)/WordLength;
        IF ~body
          THEN
            BEGIN idinfo ← nW*WordLength; idvalue ← BitAddress[wd:w, bd:0];
            END;
        w ← w + nW;
        ENDLOOP;
      END;
  RETURN [w]
  END;


-- record layout
```

```
  ScanVariants: PROCEDURE
      [caseCtx: CTXIndex, proc: PROCEDURE [recordCSEIndex] RETURNS [BOOLEAN]]
      RETURNS [BOOLEAN] =
    BEGIN
    sei: ISEIndex;
    rSei: SEIndex;
    FOR sei ← (ctxb+caseCtx).selist, NextSe[sei] UNTIL sei = SENull
      DO
      rSei ← (seb+sei).idinfo;
      WITH variant: (seb+rSei) SELECT FROM
        constructor =>
          WITH variant SELECT FROM
            record => IF proc[LOOPHOLE[rSei]] THEN RETURN [TRUE];
            ENDCASE => ERROR;
        ENDCASE =>  NULL;       -- skip multiple identifiers
      ENDLOOP;
    RETURN [FALSE]
    END;

  LayoutFields: PUBLIC PROCEDURE [rSei: recordCSEIndex, offset: CARDINAL] =
    BEGIN
    w, b: CARDINAL;
    lastFillable: BOOLEAN;
    lastSei: ISEIndex;

    AssignField: PROCEDURE [sei: ISEIndex] =
      BEGIN  OPEN id: (seb+sei);
      n, nW, nB: CARDINAL;
      saveIndex: CARDINAL = dataPtr.textIndex;
      dataPtr.textIndex ← (tb+LOOPHOLE[id.idvalue, TreeDefs.TreeIndex]).info;
--    IF id.idinfo = 0 AND ~id.public AND id.htptr # HTNull
--      THEN ErrorDefs.WarningSei[unusedId, sei];
      n ← BitsForType[id.idtype];
      nW ← n/WordLength;  nB ← n MOD WordLength;
      IF nW > 0 AND nB # 0
        THEN  BEGIN  nW ← nW+1;  nB ← 0  END;
      IF (nW > 0 OR b+nB > WordLength) AND b # 0
        THEN  BEGIN  w ← w+1;  b ← 0  END;
      dataPtr.textIndex ← saveIndex;
      IF b = 0 AND lastFillable THEN FillWord[lastSei];
      id.idinfo ← nW*WordLength + nB;
      id.idvalue ← BitAddress[wd:w, bd:b];
      lastSei ← sei;  lastFillable ← (nW = 0);
      w ← w + nW;  b ← b + nB;
      IF b >= WordLength THEN  BEGIN  w ← w+1;  b ← b - WordLength  END;
      RETURN
      END;

    FillWord: PROCEDURE [sei: ISEIndex] =
      BEGIN
      t: BitAddress = (seb+sei).idvalue;
      width: CARDINAL = WordLength - t.bd;
      IF (seb+rSei).machineDep AND width # (seb+sei).idinfo
        THEN ErrorDefs.WarningSei[recordGap, sei];
      (seb+sei).idinfo ← width;
      RETURN
      END;

    FindFit: PROCEDURE [vSei: recordCSEIndex] RETURNS [BOOLEAN] =
      BEGIN
      sei: ISEIndex;
      type: CSEIndex;
      sei ← (ctxb+(seb+vSei).fieldctx).selist;
      IF sei = SENull THEN RETURN [FALSE];
      type ← UnderType[(seb+sei).idtype];
      WITH (seb+type) SELECT FROM
        union =>
          IF controlled
            THEN   sei ← tagsei
            ELSE RETURN [ScanVariants[casectx, FindFit]];
        ENDCASE => NULL;
      RETURN [BitsForType[(seb+sei).idtype] + b <= WordLength]
      END;

    vOrigin: CARDINAL;
    maxLength: CARDINAL;
```

```
AssignVariant: PROCEDURE [vSei: recordCSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  LayoutFields[vSei, vOrigin];
  maxLength ← MAX[(seb+vSei).length, maxLength];
  RETURN [FALSE]
  END;

eqLengths: BOOLEAN;
padEnd: CARDINAL;

PadVariant: PROCEDURE [vSei: recordCSEIndex] RETURNS [BOOLEAN] =
  BEGIN
  sei, fillSei: ISEIndex;
  type: CSEIndex;
  fillOrigin: CARDINAL;
  t: BitAddress;
  ctx: CTXIndex = (seb+vSei).fieldctx;
  fillSei ← ISENull;
  FOR sei ← (ctxb+ctx).selist, NextSe[sei] UNTIL sei = SENull
      DO
      IF LOOPHOLE[(seb+sei).idvalue, BitAddress].wd # w THEN EXIT;
      fillSei ← sei;
      ENDLOOP;
  IF fillSei # SENull
    THEN
      BEGIN
      t ← (seb+fillSei).idvalue;  fillOrigin ← t.wd*WordLength + t.bd;
      IF fillOrigin + (seb+fillSei).idinfo < padEnd
        THEN
          BEGIN
          type ← UnderType[(seb+fillSei).idtype];
          WITH (seb+type) SELECT FROM
            union =>
              BEGIN
              saveLastSei: ISEIndex = lastSei;
              IF controlled THEN lastSei ← tagsei;  -- for messages only
              [] ← ScanVariants[casectx, PadVariant];
              lastSei ← saveLastSei;
              END;
            ENDCASE =>
              IF (seb+rSei).machineDep
                THEN ErrorDefs.WarningSei[recordGap, fillSei];
          (seb+fillSei).idinfo ←  padEnd - fillOrigin;
          END;
      END
    ELSE
      IF vOrigin < padEnd AND (vOrigin # 0 OR maxLength < WordLength)
        THEN
          BEGIN
          IF (seb+rSei).machineDep
            THEN ErrorDefs.WarningSei[recordGap, lastSei];
          fillSei ← makectxse[HTNull, CTXNull];
          (seb+fillSei).public ← TRUE;  (seb+fillSei).extended ← FALSE;
          (seb+fillSei).constant ← (seb+fillSei).writeonce ← FALSE;
          (seb+fillSei).linkSpace ← FALSE;
          (seb+fillSei).idtype ← dataPtr.idANY;
          (seb+fillSei).idvalue ← BitAddress[wd:w, bd:b];
          (seb+fillSei).idinfo ← padEnd - vOrigin;
          (seb+fillSei).mark3 ← (seb+fillSei).mark4 ← TRUE;
          WITH (seb+fillSei) SELECT FROM
            linked => link ← (ctxb+ctx).selist;
            ENDCASE => ERROR;
          (ctxb+ctx).selist ← fillSei;
          END;
  (seb+vSei).length ← MIN[
            maxLength,
            ((seb+vSei).length + WordFill)/WordLength * WordLength];
  IF (seb+vSei).length # maxLength THEN eqLengths ← FALSE;
  RETURN [FALSE]
  END;

sei: ISEIndex;
type: CSEIndex;
ctx: CTXIndex = (seb+rSei).fieldctx;
w ← offset/WordLength;  b ← offset MOD WordLength;
```

```
    lastFillable ← FALSE;  lastSei ← ISENull;
    FOR sei ← (ctxb+ctx).selist, NextSe[sei] UNTIL sei = SENull
      DO
      IF ~(seb+sei).constant
        THEN
          BEGIN
          type ← UnderType[(seb+sei).idtype];
          WITH (seb+type) SELECT FROM
            union =>
              BEGIN
              IF ~controlled
                THEN  (seb+sei).idvalue ← BitAddress[wd:w, bd:b]
                ELSE
                  BEGIN
                  AssignField[tagsei];
                  (seb+sei).idvalue ← (seb+tagsei).idvalue;
                  END;
              IF lastFillable AND b # 0 AND ~ScanVariants[casectx, FindFit]
                THEN
                  BEGIN  FillWord[lastSei];  w ← w+1;  b ← 0  END;
              maxLength ← vOrigin ← w*WordLength + b;
              [] ← ScanVariants[casectx, AssignVariant];
              padEnd ← IF maxLength < WordLength
                THEN maxLength
                ELSE MAX[(vOrigin + WordFill)/WordLength, 1]*WordLength;
              eqLengths ← TRUE;
              [] ← ScanVariants[casectx, PadVariant];
              equalLengths ← eqLengths;
              (seb+sei).idinfo ←
                (maxLength - vOrigin) +
                  (IF controlled THEN (seb+tagsei).idinfo ELSE 0);
              w ← maxLength/WordLength;  b ← maxLength MOD WordLength;
              lastFillable ← FALSE;
              END;
            ENDCASE =>  AssignField[sei];
          END;
      ENDLOOP;
    IF lastFillable AND b # 0 AND w > 0
      THEN  BEGIN  FillWord[lastSei];  b ← 0;  w ← w + 1  END;
    (seb+rSei).length ← w*WordLength + b;  RETURN
    END;

  END.
```