```
-- file stack.mesa
-- last modified by Sweet, July 5, 1978  9:08 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [Address, BYTE, VMLimit, wordlength],
  Code: FROM "code" USING [acstack, ACStackOverflow, ACStackUnderflow, codeptr, curctxlvl, framesz, mai
**nBody, stking, tempcontext, tempstart],
  CodeDefs: FROM "codedefs" USING [CCIndex, CCNull, ChunkBase, EvalStackSize, Lexeme, StkIndex, StkItem
**, TempStateRecord, topostack],
  ComData: FROM "comdata" USING [bodyIndex, mainCtx, objectFrameSize],
  ErrorDefs: FROM "errordefs" USING [errorsei],
  FOpCodes: FROM "fopcodes" USING [qEXCH, qFREE],
  P5ADefs: FROM "p5adefs" USING [Ciout0, createlabel, deletecell, makeBDOItem, MinimalStack, PopEffect,
** PushEffect, sCassign, slCassign],
  P5BDefs: FROM "p5bdefs" USING [pushlex],
  StringDefs: FROM "stringdefs" USING [WordsForString],
  SymDefs: FROM "symdefs" USING [BitAddress, bodytype, ContextLevel, CSEIndex, CTXIndex, ctxtype, HTInd
**ex, HTNull, ISEIndex, ISENull, lG, lZ, SEIndex, SENull, SERecord, setype, typeANY],
  SymTabDefs: FROM "symtabdefs" USING [makenewctx, NextSe, setselink],
  SystemDefs: FROM "systemdefs" USING [AllocateHeapNode, FreeHeapNode],
  TableDefs: FROM "tabledefs" USING [Allocate, TableBase, TableNotifier],
  TreeDefs: FROM "treedefs" USING [treetype];


DEFINITIONS FROM CodeDefs;

Stack: PROGRAM
    IMPORTS MPtr: ComData, CPtr: Code, ErrorDefs, P5ADefs, P5BDefs, StringDefs, SymTabDefs, SystemDefs,
** TableDefs
    EXPORTS CodeDefs, P5ADefs =
  BEGIN
  OPEN P5ADefs, P5BDefs;


  -- imported definitions

  Address: TYPE = AltoDefs.Address;
  BYTE: TYPE = AltoDefs.BYTE;
  VMLimit: CARDINAL = AltoDefs.VMLimit;
  wordlength: CARDINAL = AltoDefs.wordlength;

  BitAddress: TYPE = SymDefs.BitAddress;
  ContextLevel: TYPE = SymDefs.ContextLevel;
  CSEIndex: TYPE = SymDefs.CSEIndex;
  CTXIndex: TYPE = SymDefs.CTXIndex;
  HTIndex: TYPE = SymDefs.HTIndex;
  HTNull: HTIndex = SymDefs.HTNull;
  ISEIndex: TYPE = SymDefs.ISEIndex;
  ISENull: ISEIndex = SymDefs.ISENull;
  lZ: ContextLevel = SymDefs.lZ;
  lG: ContextLevel = SymDefs.lG;
  SEIndex: TYPE = SymDefs.SEIndex;
  SENull: SEIndex = SymDefs.SENull;
  SERecord: TYPE = SymDefs.SERecord;
  typeANY: CSEIndex = SymDefs.typeANY;


  InvalidHeapRelease: SIGNAL = CODE;
  InvalidTempRelease: SIGNAL = CODE;

  seb: TableDefs.TableBase;              -- semantic entry base (local copy)
  ctxb: TableDefs.TableBase;             -- context entry base (local copy)
  cb: ChunkBase;                 -- code base (local copy)
  bb: TableDefs.TableBase;               -- body table base (local copy)

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


  pendtemplist, templistpool, templist, heaplist: ISEIndex;
```

```
  stklist: StkIndex;
  stkptr: StkIndex;
  stkUB: INTEGER ← EvalStackSize - 2;
  StackModelingError: SIGNAL = CODE;

  StkNull: StkIndex = NIL;
  ItemStkIndex: TYPE = POINTER TO item StkItem;

  StackInit: PUBLIC PROCEDURE =
    BEGIN -- called at beginning of MODULE to init stack stuff
    pendtemplist ← templistpool ← templist ← heaplist ← ISENull;
    stkptr ← stklist ← StkNull;
    CPtr.tempcontext ← SymTabDefs.makenewctx[1Z];
    stkUB ← EvalStackSize - 1;
    CPtr.stking ← FALSE;
    RETURN
    END;


  StackFinal: PUBLIC PROCEDURE =
    BEGIN -- called at end of MODULE to release stack items
    s: StkIndex;

    UNTIL stklist = StkNull DO
      s ← stklist;
      stklist ← stklist.uplink;
      SystemDefs.FreeHeapNode[s];
      ENDLOOP;
    RETURN
    END;


  stkerror: PROCEDURE = BEGIN SIGNAL StackModelingError; RETURN END;

  pushtempstate: PUBLIC PROCEDURE [p: POINTER TO TempStateRecord, newfs: CARDINAL] =
    BEGIN
    p↑ ← TempStateRecord[pendtemplist: pendtemplist, templist: templist,
            heaplist: heaplist, tempctxlvl: (ctxb+CPtr.tempcontext).ctxlevel,
            tempstart: CPtr.tempstart, framesz: CPtr.framesz];
    pendtemplist ← templist ← heaplist ← ISENull;
    (ctxb+CPtr.tempcontext).ctxlevel ← CPtr.curctxlvl;
    CPtr.tempstart ← CPtr.framesz ← newfs;
    RETURN
    END;



  poptempstate:  PUBLIC PROCEDURE [p: POINTER TO TempStateRecord] =
    BEGIN
    purgependtemplist[];
    [pendtemplist: pendtemplist, templist: templist,
      heaplist: heaplist, tempctxlvl: (ctxb+CPtr.tempcontext).ctxlevel,
      tempstart: CPtr.tempstart, framesz: CPtr.framesz] ← p↑;
    RETURN
    END;



  stackoff:  PUBLIC PROCEDURE =
    BEGIN -- turns stack modelling off
    CPtr.stking ← FALSE;
    RETURN
    END;


  stackon:  PUBLIC PROCEDURE =
    BEGIN -- turns stack modelling on
    CPtr.stking ← TRUE;
    RETURN
    END;


  push: PROCEDURE [l: se Lexeme] =
    BEGIN -- adds item to stack
    s: StkIndex ← stackalloc[];
```

```
    s↑ ← StkItem[downlink: stkptr, uplink: , stkvalue: item[1]];
    IF stkptr # StkNull THEN
      BEGIN
      s.uplink ← stkptr.uplink;
      stkptr.uplink ← s;
      END
    ELSE s.uplink ← StkNull;
    stkptr ← s;
    RETURN
    END;


  pop: PUBLIC PROCEDURE =
    BEGIN -- moves stkptr down the stack
    s: StkIndex ← stkptr;

    IF stkptr = StkNull THEN BEGIN stkerror[]; RETURN END;
    stkptr ← stkptr.downlink;
    delstkitems[s,1];
    RETURN
    END;


  delstkitems: PROCEDURE [s: StkIndex, n: CARDINAL] =
    BEGIN -- removes n items from s upward (including s)
    ns, ds: StkIndex;

    ds ← s.downlink;
    THROUGH [1..n] DO
      ns ← s.uplink;
      stackfree[s];
      s ← ns;
      ENDLOOP;
    IF ds # StkNull THEN ds.uplink ← s;
    IF s # StkNull THEN s.downlink ← ds;
    RETURN
    END;


  stackalloc: PROCEDURE RETURNS [s: StkIndex] =
    BEGIN -- allocates a stkitem
    IF (s ← stklist) # StkNull THEN
      BEGIN
      stklist ← s.uplink;
      RETURN;
      END;
    s ← SystemDefs.AllocateHeapNode[SIZE[StkItem]];
    RETURN
    END;


  stackfree: PROCEDURE [s: StkIndex] =
    BEGIN -- frees a stkitem
    WITH s SELECT FROM
      MARK =>
        BEGIN
        IF CPtr.codeptr = label THEN CPtr.codeptr ← cb[label].blink;
        deletecell[label];
        END;
      ENDCASE;
    s.uplink ← stklist; stklist ← s;
    RETURN
    END;


  clearstack:  PUBLIC PROCEDURE =
    BEGIN -- clears out the entire stack
    s: StkIndex ← stkptr;

    UNTIL s = StkNull DO
      stkptr ← s.downlink;
      stackfree[s];
      s ← stkptr;
      ENDLOOP;
    RETURN
    END;
```

```
newstack:  PUBLIC PROCEDURE RETURNS [s: StkIndex] =
  BEGIN -- sets up a new (empty) stack returning old stkptr
  s ← stkptr;
  stkptr ← StkNull;
  RETURN
  END;

restoreoldstack:  PUBLIC PROCEDURE [s: StkIndex] =
  BEGIN -- inverse of newstack
  clearstack[];
  stkptr ← s;
  RETURN
  END;


markstack:  PUBLIC PROCEDURE =
  BEGIN -- marks stack for fork reset
  s: StkIndex ← stackalloc[];

  s↑ ← StkItem[uplink: , downlink: stkptr, stkvalue: MARK[label: createlabel[]]];
  IF stkptr # StkNull THEN
    BEGIN
    s.uplink ← stkptr.uplink;
    stkptr.uplink ← s;
    END
  ELSE s.uplink ← StkNull;
  stkptr ← s;
  RETURN
  END;


resettomark:  PUBLIC PROCEDURE =
  BEGIN -- resets stkptr to nearest mark making sure all
        -- intervening items are on the stack
  s: StkIndex ← stkptr;
  n: CARDINAL ← 0;

  DO
    IF s = StkNull THEN BEGIN stkerror[]; RETURN END;
    WITH s SELECT FROM
      MARK =>
        BEGIN chkrandsonstack[n]; stkptr ← s; RETURN END;
      ENDCASE;
    n ← n+1;
    s ← s.downlink;
    ENDLOOP
  END;


unmarkstack: PUBLIC PROCEDURE =
  BEGIN -- ensures all items down to nearest mark are on stack
        -- and removes mark (called by last branch of expression forks)
        -- does NOT change stkptr
  s: StkIndex ← stkptr;
  n: CARDINAL ← 0;

  DO
    IF s = StkNull THEN BEGIN stkerror[]; RETURN END;
    WITH s SELECT FROM
      MARK =>
        BEGIN putrandsonstack[n]; delstkitems[s,1]; RETURN END;
      ENDCASE;
    n ← n+1;
    s ← s.downlink;
    ENDLOOP
  END;


deletetomark:  PUBLIC PROCEDURE =
  BEGIN -- like resettomark, but also deletes mark
  s: StkIndex;

  resettomark[];
  IF (s ← stkptr) = StkNull THEN BEGIN stkerror[]; RETURN END;
```

```
    stkptr ← stkptr.downlink;
    delstkitems[s, 1];
    RETURN;
    END;


  incrstack:  PUBLIC PROCEDURE [n: CARDINAL] =
    BEGIN -- pushes n items onto the stack
    THROUGH [1..n] DO push[topostack] ENDLOOP;
    RETURN
    END;


  dumpstack:  PUBLIC PROCEDURE =
    BEGIN -- puts all stkitems into temps
    s: StkIndex ← stkptr;
    ss: StkIndex ← stkptr;
    savcodeptr: CCIndex ← CPtr.codeptr;
    n: CARDINAL ← 0;

    stackoff[];
    DO
      IF s = StkNull THEN
        BEGIN
        unloadstack[ss,n];
        stackon[]; CPtr.codeptr ← savcodeptr;
        UNTIL cb[CPtr.codeptr].flink = CCNull
          DO CPtr.codeptr ← cb[CPtr.codeptr].flink ENDLOOP;
        RETURN
        END;
      WITH s SELECT FROM
        MARK =>
          BEGIN unloadstack[ss,n]; n ← 0; ss ← downlink; CPtr.codeptr ← label; END;
        item =>
          IF lexeme # topostack THEN
            BEGIN unloadstack[ss,n]; EXIT END
          ELSE n ← n+1;
        ENDCASE;
      s ← s.downlink;
      ENDLOOP;
    UNTIL (s ← s.downlink) = StkNull
      DO
      WITH s SELECT FROM
        item =>
          IF lexeme = topostack THEN stkerror[];
        ENDCASE;
      ENDLOOP;
    stackon[];
    CPtr.codeptr ← savcodeptr;
    UNTIL cb[CPtr.codeptr].flink = CCNull
      DO CPtr.codeptr ← cb[CPtr.codeptr].flink ENDLOOP;
    RETURN
    END;


  bltnwordsfromstack: PUBLIC PROCEDURE [n: CARDINAL] RETURNS [tlex: se Lexeme] =
    BEGIN -- put n top words of stack in contiguous storage
    a: BitAddress;

    IF n = 0 THEN BEGIN stkerror[]; RETURN END;
    stackoff[];
    unloadstackcontiguous[stkptr,n];
    WITH stkptr SELECT FROM
      item =>
        BEGIN
        a ← (seb+lexeme.lexsei).idvalue;
        tlex ← createtemplex[a.wd+1-n, n];
        END;
      ENDCASE;
    stackon[];
    RETURN
    END;


  unloadstack: PROCEDURE [s: StkIndex, n: CARDINAL] =
    BEGIN -- main subr for dumpstack
```

```
    tlex: se Lexeme;
    tempsneeded: CARDINAL ← 0;
    ts: CARDINAL;
    ss: StkIndex ← s;
    nn: CARDINAL ← n;

    DO
      IF nn = 0 THEN EXIT;
      IF ss = StkNull THEN BEGIN stkerror[]; RETURN END;
      WITH ss SELECT FROM
        MARK =>
          BEGIN stkerror[]; RETURN; END;
        item =>
          IF lexeme = topostack THEN tempsneeded ← tempsneeded+1;
        ENDCASE;
      nn ← nn-1;
      ss ← ss.downlink;
      ENDLOOP;
    ts ← CPtr.tempstart;
    bumptemps[tempsneeded];
    DO
      IF n = 0 THEN RETURN;
      WITH s SELECT FROM
        item =>
          IF lexeme = topostack THEN
            BEGIN
            tempsneeded ← tempsneeded-1;
            tlex ← createtemplex[ts+tempsneeded,1];
            releasetemplex[tlex];
            sCassign[tlex.lexsei];
            lexeme ← tlex;
            END;
        ENDCASE;
      n ← n-1;
      s ← s.downlink;
      ENDLOOP;
    END;


  unloadstackcontiguous: PROCEDURE [ss: StkIndex, n: CARDINAL] =
    BEGIN -- main subr for bltnwordsfromstack
    tlex, l: se Lexeme;
    s: StkIndex ← ss;
    i: CARDINAL ← 0;
    k, w, ts: CARDINAL;
    a: BitAddress;

    IF n = 0 THEN RETURN;
    DO
      IF s = StkNull THEN BEGIN stkerror[]; RETURN END;
      WITH s SELECT FROM
        MARK =>
          BEGIN stkerror[]; RETURN; END;
        item =>
          IF lexeme # topostack THEN EXIT;
        ENDCASE;
      IF (i ← i+1) = n THEN EXIT;
      s ← s.downlink;
      ENDLOOP;
    k ← i; -- k = # actually on stack;
    IF i # n THEN
      WITH s SELECT FROM
        item =>
          BEGIN
          a ← (seb+lexeme.lexsei).idvalue;
          IF (w ← a.wd) = (CPtr.tempstart-1) THEN
            UNTIL (i ← i+1) = n DO
              s ← downlink;
              IF s = StkNull THEN BEGIN stkerror[]; RETURN END;
              WITH s SELECT FROM
                MARK =>
                  BEGIN stkerror[]; RETURN END;
                item =>
                  BEGIN
                  a ← (seb+lexeme.lexsei).idvalue;
                  IF w+1 # a.wd THEN EXIT;
```

```
            END;
          ENDCASE;
        w ← w-1;
        ENDLOOP;
      END; -- i = # already properly contiguous with free temps
    ENDCASE;
  IF i = n AND k = 0 THEN RETURN;      -- already in contiguous locations
  s ← ss;
  ts ← CPtr.tempstart;
  IF i = n THEN
    BEGIN -- ones not on the stack are in the right place.  Store others.
    bumptemps[k];
    FOR i DECREASING IN [0..k) DO
      tlex ← createtemplex[ts+i,1]; -- peephole will take care of SLD's
      releasetemplex[tlex];
      sCassign[tlex.lexsei];
      WITH s SELECT FROM
        item => lexeme ← tlex;
        ENDCASE;
      s ← s.downlink;
      ENDLOOP;
    RETURN;
    END;
  bumptemps[n]; -- they're not contiguous, must copy to a new big temp
  IF (n MOD 2) # 0 THEN
    BEGIN
    tlex ← createtemplex[ts+n-1,1];
    releasetemplex[tlex];
    WITH s SELECT FROM
      item =>
        BEGIN
        IF lexeme = topostack THEN sCassign[tlex.lexsei]
        ELSE slCassign[tlex.lexsei, lexeme, FALSE, 1];
        lexeme ← tlex;
        END;
      ENDCASE;
    s ← s.downlink;
    END;
  FOR i DECREASING IN [0..n/2)
    DO
    IF s = StkNull THEN BEGIN stkerror[]; RETURN; END;
    WITH s SELECT FROM
      MARK =>
        BEGIN stkerror[]; RETURN; END;
      item =>
        BEGIN
        tlex ← createtemplex[ts+2*i,2];
        releasetemplex[tlex];
        ss ← downlink; l ← lexeme;
        IF l # topostack THEN
          WITH ss SELECT FROM
            item =>
              IF lexeme # topostack THEN
                BEGIN
                IF trydoubleload[lexeme, l] = 1 THEN pushlex[l];
                END;
            ENDCASE
        ELSE
          WITH ss SELECT FROM
            item =>
              IF lexeme # topostack THEN
                BEGIN
                pushlex[lexeme];
                Ciout0[FOpCodes.qEXCH];
                END;
            ENDCASE;
        END;
      ENDCASE;
    sCassign[tlex.lexsei];
    s ← setstklex2[ts+2*i, s];
    ENDLOOP;
  RETURN
  END;

setstklex2: PROCEDURE [i: Address, s: StkIndex] RETURNS [StkIndex] =
```

```
  BEGIN
  i ← i+1;
  THROUGH [0..1] DO
    WITH s SELECT FROM
      item =>
        BEGIN
        lexeme ← createtemplex[i, 1];
        releasetemplex[lexeme];
        END;
      ENDCASE;
    i ← i-1;
    s ← s.downlink;
    ENDLOOP;
  RETURN[s]
  END;


trydoubleload: PROCEDURE [l1, l2: se Lexeme] RETURNS [CARDINAL] =
  BEGIN
  a1, a2: BitAddress;
  l: bdo Lexeme;

  a1 ← (seb+l1.lexsei).idvalue; a2 ← (seb+l2.lexsei).idvalue;
  IF a2.wd = (a1.wd + 1) THEN
    BEGIN
    l ← makeBDOItem[l1];
    cb[l.lexbdoi].offset.size ← 2*wordlength;
    pushlex[l];
    RETURN[2]
    END;
  pushlex[l1];
  RETURN[1]
  END;



putrandsonstack: PUBLIC PROCEDURE [n: CARDINAL] =
  BEGIN -- ensures that the n items on stack are in acstack
  stkary: ARRAY [0..EvalStackSize) OF se Lexeme;
  i, k: CARDINAL;
  s: StkIndex ← stkptr;
  ais: BOOLEAN ← TRUE;

  k ← 0;
  FOR i IN [0..n) DO
    IF s = StkNull THEN BEGIN stkerror[]; RETURN END;
    WITH s SELECT FROM
      MARK => stkerror[];
      item =>
        BEGIN
        stkary[i] ← lexeme;
        IF stkary[i] = topostack THEN
          BEGIN
          IF ~ais THEN stkerror[];
          k ← k + 1;
          END
        ELSE ais ← FALSE;
        END;
      ENDCASE;
    s ← s.downlink;
    ENDLOOP;
  IF ais THEN RETURN;
  stackoff[];
  IF n = 2 AND (stkary[0] = topostack) THEN
    BEGIN
    pushlex[stkary[1]];
    WITH stkptr.downlink SELECT FROM
      item => lexeme ← topostack;
      ENDCASE;
    Ciout0[FOpCodes.qEXCH];
    stackon[];
    RETURN
    END;
  unloadstack[stkptr, k];
  s ← stkptr;
  FOR i IN [0..n) DO
```

```
    WITH s SELECT FROM
      MARK => stkerror[];
      item =>
        BEGIN
        stkary[i] ← lexeme;
        lexeme ← topostack;
        END;
      ENDCASE;
    s ← s.downlink;
    ENDLOOP;
  UNTIL n < 2 DO
    n ← n - trydoubleload[stkary[n-1], stkary[n-2]] ENDLOOP;
  IF n = 1 THEN pushlex[stkary[0]];
  stackon[];
  RETURN
  END;


chkrandsonstack: PUBLIC PROCEDURE [n: CARDINAL] =
  BEGIN -- ensures n items on stack and deletes them
  s: StkIndex ← stkptr;

  IF n = 0 THEN RETURN;
  putrandsonstack[n];
  THROUGH [1..n) DO s ← s.downlink ENDLOOP;
  stkptr ← s.downlink;
  delstkitems[s,n];
  RETURN
  END;


gentemplex:  PUBLIC PROCEDURE [nwords: CARDINAL] RETURNS [l: se Lexeme] =
  BEGIN
  l ← createtemplex[CPtr.tempstart, nwords];
  releasetemplex[l];
  bumptemps[nwords];
  RETURN
  END;


genanonlex:  PUBLIC PROCEDURE [nwords: CARDINAL] RETURNS [l: se Lexeme] =
  BEGIN
  l ← createtemplex[CPtr.tempstart, nwords];
  bumptemps[nwords];
  RETURN
  END;


genstringbodylex:  PUBLIC PROCEDURE [nchars: CARDINAL] RETURNS [l: se Lexeme] =
  BEGIN
  nwords: CARDINAL ← StringDefs.WordsForString[nchars];
  IF ~CPtr.mainBody THEN RETURN [genanonlex[nwords]];
  l ← createtemplex[MPtr.objectFrameSize, nwords];
  (seb+l.lexsei).ctxnum ← MPtr.mainCtx;
  MPtr.objectFrameSize ← MPtr.objectFrameSize + nwords;
  RETURN
  END;


bumptemps: PROCEDURE [n: CARDINAL] =
  BEGIN -- updates CPtr.tempstart (and CPtr.framesz, if necessary)
  CPtr.framesz ← MAX[CPtr.tempstart ← CPtr.tempstart+n, CPtr.framesz];
  IF CPtr.framesz > VMLimit/wordlength THEN
    ErrorDefs.errorsei[addressOverflow, (bb+MPtr.bodyIndex).id];
  RETURN
  END;


purgependtemplist:  PUBLIC PROCEDURE =
  BEGIN -- after each statment the temp sei's are released
  sei: ISEIndex ← pendtemplist;
  nsei: ISEIndex;

  WHILE sei # ISENull DO
    nsei ← SymTabDefs.NextSe[sei];
    releasetempsei[sei];
```

```
    sei ← nsei;
    ENDLOOP;
  pendtemplist ← ISENull;
  RETURN
  END;


purgeheaplist: PUBLIC PROCEDURE[oldheaplist: ISEIndex] =
  BEGIN -- after each statment the heap chunks are freed
  sei: ISEIndex ← heaplist;
  nsei: ISEIndex;
  l: se Lexeme ← Lexeme[se[]];

  WHILE sei # ISENull DO
    nsei ← SymTabDefs.NextSe[sei];
    l.lexsei ← sei;
    freeheaplex[l];
    sei ← nsei;
    ENDLOOP;
  heaplist ← oldheaplist;
  RETURN
  END;


freeheaplex: PUBLIC PROCEDURE [l: se Lexeme] =
  BEGIN
  pushlex[l];
  Ciout0[FOpCodes.qFREE];
  releasetempsei[l.lexsei];
  RETURN
  END;

pushheaplist: PUBLIC PROCEDURE RETURNS[oldheaplist: ISEIndex] =
  BEGIN
  oldheaplist ← heaplist;
  heaplist ← ISENull;
  RETURN
  END;


genheaplex: PUBLIC PROCEDURE RETURNS[l: se Lexeme] =
  BEGIN
  l ← genanonlex[1];
  SymTabDefs.setselink[l.lexsei, heaplist];
  heaplist ← l.lexsei;
  RETURN
  END;


freetempsei: PUBLIC PROCEDURE [sei: ISEIndex] =
  BEGIN -- de-links a temp sei from its chain
  SymTabDefs.setselink[sei, templistpool];
  templistpool ← sei;
  RETURN
  END;


releasetempsei:  PROCEDURE [sei: ISEIndex] =
  BEGIN -- puts a temp sei on the templist
  a: BitAddress ← (seb+sei).idvalue;

  CPtr.tempstart ← MIN[CPtr.tempstart, a.wd];
  freetempsei[sei];
  RETURN
  END;


createtemplex: PROCEDURE [wdoffset, nwords: INTEGER] RETURNS [l: se Lexeme] =
  BEGIN -- inits (if nec) a new temp sei cell
  sei: ISEIndex;
  a: BitAddress;

  IF templistpool # SENull THEN
    BEGIN
    sei ← templistpool;
    templistpool ← SymTabDefs.NextSe[sei];
```

```
    (seb+sei).ctxnum ← CPtr.tempcontext;
    END
  ELSE
    BEGIN
    sei ← TableDefs.Allocate[SymDefs.setype, SIZE[linked id SERecord]];
    (seb+sei)↑ ← SERecord[mark3: , mark4: ,
      sebody: id[extended: FALSE, public: , writeonce: , linkSpace: FALSE,
      constant: FALSE, ctxnum: CPtr.tempcontext, htptr:HTNull ,
      idtype: typeANY, idinfo: , idvalue: , ctxlink: linked[LOOPHOLE[0]]]];
    END;
  SymTabDefs.setselink[sei, LOOPHOLE[0]];
  a ← BitAddress[wd: wdoffset, bd: 0];
  (seb+sei).idvalue ← a;
  (seb+sei).idinfo ← nwords*wordlength;
  l ← Lexeme[se[lexsei: sei]];
  RETURN
  END;


releasetemplex: PUBLIC PROCEDURE [l: se Lexeme] =
  BEGIN -- releases a cell of temporary storage
  IF SymTabDefs.NextSe[l.lexsei] # LOOPHOLE[0, ISEIndex] THEN RETURN;
  SymTabDefs.setselink[l.lexsei, pendtemplist];
  pendtemplist ← l.lexsei;
  RETURN
  END;


freetemplist:  PUBLIC PROCEDURE =
  BEGIN -- at end of body puts se-entries of temp cells on list from templistpool
  sei: ISEIndex ← templist;
  nsei: ISEIndex;

  UNTIL sei = ISENull DO
    nsei ← SymTabDefs.NextSe[sei];
    freetempsei[sei];
    sei ← nsei;
    ENDLOOP;
  templist ← ISENull;
  RETURN
  END;


chkacstack: PUBLIC PROCEDURE [b: BYTE] =
  BEGIN -- checks AC stack for over/underflow
  pusheffect: INTEGER = PushEffect[b];
  popeffect: INTEGER = PopEffect[b];
  neteffect: INTEGER = pusheffect - popeffect;

  IF (CPtr.acstack + neteffect) > stkUB THEN
    BEGIN
    IF (stkUB ← stkUB+1) > EvalStackSize THEN
      BEGIN SIGNAL CPtr.ACStackOverflow; RETURN; END;
    dumpstack[];
    CPtr.acstack ← 0;
    stkUB ← EvalStackSize - 2;
    END;
  IF CPtr.stking THEN
    BEGIN chkrandsonstack[popeffect]; incrstack[pusheffect]; END;
  IF CPtr.acstack # popeffect AND MinimalStack[b] THEN
    SIGNAL StackModelingError;
  IF (CPtr.acstack ← CPtr.acstack + neteffect) < 0 THEN
    SIGNAL CPtr.ACStackUnderflow;
  RETURN
  END;


adjustacstack:  PUBLIC PROCEDURE [x: INTEGER] =
  BEGIN -- used to adjust acstack at fork join points
  CPtr.acstack ← CPtr.acstack+x;
  SELECT CPtr.acstack FROM
    >stkUB => SIGNAL CPtr.ACStackOverflow;
    < 0 => SIGNAL CPtr.ACStackUnderflow;
    ENDCASE;
  RETURN
  END;
```

```
END...
```