```
-- file Parser.Mesa
-- last modified by Satterthwaite, August 1, 1978  2:50 PM

DIRECTORY
  IODefs: FROM "iodefs" USING [CR, TAB, WriteChar, WriteDecimal, WriteString],
  SystemDefs: FROM "systemdefs"
    USING [AllocateHeapNode, AllocateSegment, FreeHeapNode, FreeSegment],
  StreamDefs: FROM "streamdefs" USING [StreamHandle],
  LALRDefs: FROM "lalrdefs"
    USING [
      ActionEntry, ActionTag, AsstlEntry, LALRTable, ProductionInfo,
      State, Symbol, SymbolRecord,
      endmarker, InitialSymbol, lastntstate],
  P1Defs: FROM "p1defs"
    USING [
      AssignDescriptors, Atom, ErrorContext, ProcessQueue, ResetScanIndex,
      ScanInit, ScanReset, TokenValue];

Parser: PROGRAM
    IMPORTS IODefs, SystemDefs, P1Defs
    EXPORTS P1Defs SHARES LALRDefs =
  BEGIN  -- Mesa parser with error recovery
  OPEN LALRDefs;

  ErrorLimit: CARDINAL = 25;

  InitialState: State = 1;
  FinalState: State = 0;
  Scan: ActionTag = [FALSE, 0];

  currentState: State;
  inputSymbol, lhs: Symbol;
  DefaultMarker: Symbol = endmarker+1;

  input: PROCEDURE RETURNS [symbol: SymbolRecord];
  inputLoc: CARDINAL;
  inputValue: UNSPECIFIED;

  lastSymbol: SymbolRecord;
  NullSymbol: Symbol = 0;

  s: DESCRIPTOR FOR ARRAY OF State;
  l: DESCRIPTOR FOR ARRAY OF CARDINAL;
  v: DESCRIPTOR FOR ARRAY OF UNSPECIFIED;
  top: CARDINAL;
  stackSize: CARDINAL;

  q: DESCRIPTOR FOR ARRAY OF ActionEntry;
  qI: CARDINAL;
  queueSize: CARDINAL;

  lalrTable: POINTER TO LALRTable;

  -- transition tables for terminal input symbols

  tState: DESCRIPTOR FOR ARRAY OF State;
  asstl: DESCRIPTOR FOR ARRAY OF AsstlEntry;
  tSymbol: DESCRIPTOR FOR ARRAY OF Symbol;
  tAction: DESCRIPTOR FOR ARRAY OF ActionEntry;

  -- transition tables for nonterminal input symbols

  nState: DESCRIPTOR FOR ARRAY OF State;
  nLength: DESCRIPTOR FOR ARRAY OF CARDINAL;
  nSymbol: DESCRIPTOR FOR ARRAY OF Symbol;
  nAction: DESCRIPTOR FOR ARRAY OF ActionEntry;
  nDefaults: DESCRIPTOR FOR ARRAY OF ActionEntry;

  -- production information

  prodData: DESCRIPTOR FOR ARRAY OF ProductionInfo;


-- initialization/termination

  ParseInit: PROCEDURE [stream: StreamDefs.StreamHandle, tablePtr: POINTER TO LALRTable] =
```

```
    BEGIN
    lalrTable ← tablePtr;       -- for error reporting
    P1Defs.ScanInit[stream, tablePtr];

      BEGIN  OPEN tablePtr;
      tState ← DESCRIPTOR[parsetable.tstate];
      asst1 ← DESCRIPTOR[parsetable.asst1];
      tSymbol ← DESCRIPTOR[parsetable.tsym];
      tAction ← DESCRIPTOR[parsetable.tact];
      nState ←DESCRIPTOR[parsetable.nstate];
      nLength ← DESCRIPTOR[parsetable.nlen];
      nSymbol ← DESCRIPTOR[parsetable.nsym];
      nAction ← DESCRIPTOR[parsetable.nact];
      nDefaults ← DESCRIPTOR[parsetable.ntdefaults];
      prodData ← DESCRIPTOR[parsetable.proddata];
      END;

    stackSize ← queueSize ← 0;  ExpandStack[512];  ExpandQueue[256];
    RETURN
    END;

  InputLoc: PUBLIC PROCEDURE RETURNS [CARDINAL] =
    BEGIN
    RETURN [inputLoc]
    END;


-- * * * *  Main Parsing Procedures * * * * --

  Parse: PUBLIC PROCEDURE [
        stream: StreamDefs.StreamHandle,
        table: POINTER TO LALRTable]
      RETURNS [complete: BOOLEAN, nErrors: CARDINAL] =
    BEGIN
    i, valid, k, m: CARDINAL;           -- stack pointers
    j, j0: CARDINAL;
    tj: ActionEntry;

    ParseInit[stream, table];  input ← P1Defs.Atom;
    nErrors ← 0;  complete ← TRUE;
    i ← top ← valid ← 0;  qI ← 0;
    s[0] ← currentState ← InitialState;  lastSymbol.class ← NullSymbol;
    inputSymbol ← InitialSymbol;  inputValue ← 0;  inputLoc ← 0;

    WHILE currentState # FinalState DO
      BEGIN
      j0 ← tState[currentState];
      FOR j IN [j0 .. j0 + asst1[currentState].tlen)
        DO
        SELECT tSymbol[j] FROM
          inputSymbol, DefaultMarker => EXIT;
          ENDCASE;
        REPEAT
          FINISHED => GO TO SyntaxError;
        ENDLOOP;

      tj ← tAction[j];
      IF ~tj.rtag.reduce        -- scan or scan reduce entry
        THEN
          BEGIN
          IF qI > 0
            THEN
              BEGIN
              FOR k IN (valid..i] DO s[k] ← s[top+(k-valid)] ENDLOOP;
              P1Defs.ProcessQueue[qI, top];  qI ← 0;
              END;
          IF (top ← valid ← i ← i+1) >= stackSize THEN ExpandStack[256];
          lastSymbol.class ← inputSymbol; v[i] ← inputValue; l[i] ← inputLoc;
          [inputSymbol, inputValue, inputLoc]  ← input[].symbol;
          END;

      WHILE tj.rtag # Scan
        DO
        IF qI >= queueSize THEN ExpandQueue[256];
        q[qI] ← tj;  qI ← qI + 1;
        i ← i-tj.rtag.plength;
```

```
      currentState ← s[IF i > valid THEN top+(i-valid) ELSE (valid ← i)];
      lhs ← prodData[tj.transition].lhs;
        BEGIN
        IF currentState <= lastntstate
          THEN
            BEGIN  j ← nState[currentState];
            FOR j IN [j..j+nLength[currentState])
              DO
              IF lhs = nSymbol[j] THEN
                BEGIN  tj ← nAction[j]; GO TO nfound  END;
              ENDLOOP;
            END;
        tj ← nDefaults[lhs];
        EXITS
          nfound => NULL;
        END;
      i ← i+1;
      ENDLOOP;
    IF (m ← top+(i-valid)) >= stackSize THEN ExpandStack[256];
    s[m] ← currentState ← tj.transition;
    EXITS
      SyntaxError =>
        BEGIN
        lastSymbol.value ← v[top];  lastSymbol.index ← l[top];
        top ← top - 1;
        complete ← SyntaxError[(nErrors←nErrors+1)>ErrorLimit];
        i ← valid ← top;  qI ← 0;  lastSymbol.class ← NullSymbol;
        currentState ← s[i];
        [inputSymbol, inputValue, inputLoc] ← input[].symbol;
        IF ~complete THEN EXIT
        END;
    END;
  ENDLOOP;

  P1Defs.ProcessQueue[qI, top];
  EraseQueue[];  EraseStack[];
  nErrors ← nErrors + P1Defs.ScanReset[];
  RETURN [complete, nErrors]
  END;


ExpandStack: PROCEDURE [delta: CARDINAL] =
  BEGIN  OPEN SystemDefs;
  i: CARDINAL;
  newS: DESCRIPTOR FOR ARRAY OF State;
  newL: DESCRIPTOR FOR ARRAY OF CARDINAL;
  newV: DESCRIPTOR FOR ARRAY OF UNSPECIFIED;
  newSize: CARDINAL = stackSize + delta;
  newS ← DESCRIPTOR[AllocateSegment[newSize*SIZE[State]], newSize];
  newL ← DESCRIPTOR[AllocateSegment[newSize*SIZE[CARDINAL]], newSize];
  newV ← DESCRIPTOR[AllocateSegment[newSize*SIZE[UNSPECIFIED]], newSize];
  FOR i IN [0..stackSize)
    DO  newS[i] ← s[i];  newL[i] ← l[i];  newV[i] ← v[i]  ENDLOOP;
  EraseStack[];
  s ← newS;  l ← newL;  v ← newV;  stackSize ← newSize;
  P1Defs.AssignDescriptors[qd:q, vd:v, ld:l, pd:prodData];
  RETURN
  END;

EraseStack: PROCEDURE =
  BEGIN
  IF stackSize # 0
    THEN
      BEGIN  OPEN SystemDefs;
      FreeSegment[BASE[v]];  FreeSegment[BASE[l]];  FreeSegment[BASE[s]];
      END;
  RETURN
  END;

ExpandQueue: PROCEDURE [delta: CARDINAL] =
  BEGIN  OPEN SystemDefs;
  i: CARDINAL;
  newQ: DESCRIPTOR FOR ARRAY OF ActionEntry;
  newSize: CARDINAL = queueSize + delta;
  newQ ← DESCRIPTOR[AllocateSegment[newSize*SIZE[ActionEntry]], newSize];
  FOR i IN [0..queueSize)  DO  newQ[i] ← q[i]  ENDLOOP;
```

```
    EraseQueue[];
    q ← newQ;  queueSize ← newSize;
    P1Defs.AssignDescriptors[qd:q, vd:v, ld:l, pd:prodData];
    RETURN
    END;

  EraseQueue: PROCEDURE =
    BEGIN
    IF queueSize # 0 THEN SystemDefs.FreeSegment[BASE[q]];
    RETURN
    END;


-- * * * * Error Recovery Section * * * * --

 -- parameters of error recovery

  MinScanLimit: CARDINAL = 4;
  MaxScanLimit: CARDINAL = 12;
  InsertLimit: CARDINAL = 2;
  DiscardLimit: CARDINAL = 10;
  TreeSize: CARDINAL = 256;
  CheckSize: CARDINAL = MaxScanLimit+InsertLimit+2;


 -- debugging

  ParserID: PUBLIC PROCEDURE RETURNS [STRING] =
    BEGIN
    RETURN ["Standard 4.0"]
    END;

  track: BOOLEAN = FALSE;

  DisplayNode: PROCEDURE [n: NodeIndex] =
    BEGIN  OPEN IODefs;
    IF track THEN
      BEGIN
      WriteString["::new node::"L];
      WriteChar[TAB]; WriteDecimal[n];
      WriteChar[TAB]; WriteDecimal[tree[n].father];
      WriteChar[TAB]; WriteDecimal[tree[n].last]; WriteChar[TAB];
      WriteDecimal[tree[n].state];  WriteChar[TAB]; TypeSym[tree[n].symbol];
      WriteChar[CR];
      END;
    RETURN
    END;


 -- tree management

  NodeIndex: TYPE = CARDINAL [0..TreeSize);
  NullIndex: NodeIndex = 0;

  StackNode: TYPE = RECORD[
    father: NodeIndex,
    last: NodeIndex,
    state:  State,
    symbol: Symbol,
    aLeaf, bLeaf: BOOLEAN,
    link: NodeIndex];

  tree: DESCRIPTOR FOR ARRAY OF StackNode;
  nextNode: NodeIndex;
  maxNode: NodeIndex;
  treeLimit: CARDINAL;
  TreeFull: SIGNAL = CODE;


  Allocate: PROCEDURE [parent, pred: NodeIndex, terminal: Symbol, stateNo: State]
      RETURNS [index: NodeIndex] =
    BEGIN
    IF (index ← nextNode) >= treeLimit THEN SIGNAL TreeFull;
    maxNode ← MAX[index, maxNode];
    tree[index] ← StackNode[
```

```
      father: parent,
      last: pred,
      state: stateNo,
      symbol: terminal,
      aLeaf: FALSE,
      bLeaf: FALSE,
      link: NullIndex];
  nextNode ← nextNode+1;  RETURN
  END;


HashSize: INTEGER = 256;      -- should depend on state count
hashTable: DESCRIPTOR FOR ARRAY OF NodeIndex;

ParsingMode: TYPE = {ATree, BTree, Checking};
parseMode: ParsingMode;

LinkHash: PROCEDURE [n: NodeIndex] =
  BEGIN
  htIndex: [0..HashSize) = tree[n].state MOD HashSize;
  tree[n].link ← hashTable[htIndex];  hashTable[htIndex] ← n;  RETURN
  END;

ExistingConfiguration: PROCEDURE [stack: StackRep] RETURNS [NodeIndex] =
  BEGIN
  n, n1, n2: NodeIndex;
  s1, s2: State;
  htIndex: [0..HashSize);
  aTree: BOOLEAN;
  SELECT parseMode FROM
    ATree =>  aTree ← TRUE;
    BTree =>  aTree ← FALSE;
    ENDCASE => RETURN [NullIndex];
  htIndex ← stack.extension MOD HashSize;
  FOR n ← hashTable[htIndex], tree[n].link UNTIL n = NullIndex
    DO
    IF (IF aTree THEN tree[n].aLeaf ELSE tree[n].bLeaf) THEN
      BEGIN
      s1 ← stack.extension;  s2 ← tree[n].state;
      n1 ← stack.leaf;       n2 ← tree[n].father;
        DO
        IF s1 # s2 THEN EXIT;
        IF n1 = n2 THEN RETURN [n];
        s1 ← tree[n1].state;  s2 ← tree[n2].state;
        n1 ← tree[n1].father;  n2 ← tree[n2].father;
        ENDLOOP;
      END;
    ENDLOOP;
  RETURN [NullIndex]
  END;

FindNode: PROCEDURE [parent, pred: NodeIndex, stateNo: State] RETURNS [index: NodeIndex] =
  BEGIN
  index ← ExistingConfiguration[[leaf:parent, extension:stateNo]];
  IF index = NullIndex
    THEN
      BEGIN
      index ← Allocate[parent, pred, 0, stateNo];
      SELECT parseMode FROM
        ATree =>  BEGIN  tree[index].aLeaf ← TRUE;  LinkHash[index]  END;
        BTree =>  BEGIN  tree[index].bLeaf ← TRUE;  LinkHash[index]  END;
        ENDCASE => NULL;
      END;
  RETURN
  END;


-- parsing simulation

NullState: State = LAST[State];

StackRep: TYPE = RECORD[
  leaf: NodeIndex,
  extension: State];
```

```
NTEntry: PROCEDURE [state: State, lhs: Symbol] RETURNS [ActionEntry] =
  BEGIN
  j: CARDINAL;
  IF state <= lastntstate THEN
    BEGIN
    j ← nState[state];
    FOR j IN [j..j+nLength[state])
      DO  IF lhs = nSymbol[j] THEN RETURN [nAction[j]]  ENDLOOP;
    END;
  RETURN [nDefaults[lhs]]
  END;

ActOnStack: PROCEDURE [stack: StackRep, action: ActionEntry, nScanned: [0..1]]
    RETURNS [StackRep] =
  BEGIN
  currentNode, thread: NodeIndex;
  currentState: State;
  count: CARDINAL;
  currentNode ← thread ← stack.leaf;  count ← nScanned;
  IF stack.extension = NullState
    THEN  currentState ← tree[currentNode].state
    ELSE  BEGIN  currentState ← stack.extension;  count ← count + 1  END;
  UNTIL action.rtag = Scan
    DO
    IF count > action.rtag.plength  -- can be one greater
      THEN
        BEGIN
        currentNode ← FindNode[currentNode, thread, currentState];
        count ← count - 1;
        END;
    UNTIL count = action.rtag.plength
      DO
      currentNode ← tree[currentNode].father;  count ← count + 1;
      ENDLOOP;
    currentState ← tree[currentNode].state;  count ← 1;
    action ← NTEntry[currentState, prodData[action.transition].lhs];
    ENDLOOP;
  IF count > 1
    THEN currentNode ← FindNode[currentNode, thread, currentState];
  stack.leaf ← currentNode;  stack.extension ← action.transition;
  RETURN [stack]
  END;


ParseStep: PROCEDURE [stack: StackRep, input: Symbol] RETURNS [StackRep] =
  BEGIN
  currentState: State;
  j, j0: CARDINAL;
  tj: ActionEntry;
  count: [0..1];
  scanned: BOOLEAN ← FALSE;
  currentState ← IF stack.extension = NullState
      THEN tree[stack.leaf].state
      ELSE stack.extension;
  WHILE ~scanned
    DO
    j0 ← tState[currentState];
    FOR j IN [j0..j0+asstl[currentState].tlen)
      DO
      SELECT tSymbol[j] FROM
        input, DefaultMarker => EXIT;
        ENDCASE;
      REPEAT
        FINISHED => RETURN [[NullIndex, NullState]];
      ENDLOOP;
    tj ← tAction[j];
    IF ~tj.rtag.reduce
      THEN  -- shift or shift reduce
        BEGIN  count ← 1;  scanned ← TRUE  END
      ELSE  count ← 0;
    stack ← ActOnStack[stack, tj, count];
    currentState ← stack.extension;
    ENDLOOP;
  RETURN [stack]
  END;
```

```
-- text buffer management

 newText: ARRAY [0 .. 1+InsertLimit) OF SymbolRecord;
 insertCount: CARDINAL;

 BufferSize: CARDINAL = 1 + DiscardLimit + (MaxScanLimit+InsertLimit);
 sourceText: ARRAY [0 .. BufferSize) OF SymbolRecord;
 scanBase, scanLimit: CARDINAL;


 Advance: PROCEDURE =
   BEGIN
   sourceText[scanLimit] ← input[];  scanLimit ← scanLimit + 1;
   RETURN
   END;

 Discard: PROCEDURE =
   BEGIN
   IF track THEN
     BEGIN  OPEN IODefs;
     WriteString["::discarding symbol -- "L];
     TypeSym[sourceText[scanBase].class];  WriteChar[CR];
     END;
   scanBase ← scanBase+1;
   RETURN
   END;

 UnDiscard: PROCEDURE =
   BEGIN
   scanBase ← scanBase-1;
   IF track THEN
     BEGIN  OPEN IODefs;
     WriteString["::recovering symbol -- "L];
     TypeSym[sourceText[scanBase].class];  WriteChar[CR];
     END;
   RETURN
   END;

 RecoverInput: PROCEDURE RETURNS [sym: SymbolRecord] =
   BEGIN
   IF insertCount <= InsertLimit
     THEN
       BEGIN  sym ← newText[insertCount];
       insertCount ← insertCount+1;
       END
     ELSE
       BEGIN  sym ← sourceText[scanBase];
       IF (scanBase ← scanBase+1) = scanLimit THEN input ← P1Defs.Atom;
       END;
   RETURN
   END;


-- acceptance checking

 best: RECORD [
   nAccepted: CARDINAL,
   nPassed: [0..1],
   node: NodeIndex,
   mode: ParsingMode,
   nDiscards: CARDINAL];

 RightScan: PROCEDURE [node: NodeIndex] RETURNS [stop: BOOLEAN] =
   BEGIN
   i: CARDINAL;
   stack: StackRep;
   state: State;
   nAccepted: CARDINAL;
   savedNextNode: NodeIndex = nextNode;
   savedMode: ParsingMode = parseMode;
   savedLimit: CARDINAL = treeLimit;
   parseMode ← Checking;  treeLimit ← LENGTH[tree];
   nAccepted ← 0;
   state ← tree[node].state;  stack ← [leaf:node, extension:NullState];
   FOR i IN [scanBase .. scanLimit)
```

```
      DO
      IF state = FinalState
        THEN
          BEGIN
          nAccepted ← IF (sourceText[i].class = endmarker)
            THEN scanLimit-scanBase
            ELSE 0;
          EXIT
          END;
      stack ← ParseStep[stack, sourceText[i].class];
      IF stack.leaf = NullIndex THEN EXIT;
      nAccepted ← nAccepted + 1;  state ← stack.extension;
      ENDLOOP;
    nextNode ← savedNextNode;  treeLimit ← savedLimit;
    SELECT (parseMode ← savedMode) FROM
      ATree =>
        IF nAccepted + 1 > best.nAccepted + best.nPassed
          THEN best ← [nAccepted, 1, node, ATree, scanBase-1];
      BTree =>
        IF nAccepted > best.nAccepted + best.nPassed
          THEN best ← [nAccepted, 0, node, BTree, scanBase];
      ENDCASE;
    RETURN [nAccepted >= MaxScanLimit]
    END;


-- strategy management

  RowRecord: TYPE = RECORD [
    index, limit: CARDINAL,
    stack: StackRep,
    next: RowHandle];

  RowHandle: TYPE = POINTER TO RowRecord;

  NextRow: PROCEDURE [list: RowHandle] RETURNS [row: RowHandle] =
    BEGIN
    r: RowHandle;
    s, t: Symbol;
    row ← NIL;
    FOR r ← list, r.next UNTIL r = NIL
      DO
      IF r.index < r.limit
        THEN
          BEGIN  s ← tSymbol[r.index];
          IF row = NIL OR s < t THEN  BEGIN  row ← r;  t ← s  END;
          END;
      ENDLOOP;
    RETURN
    END;

  FreeRowList: PROCEDURE [list: RowHandle] =
    BEGIN
    r, next: RowHandle;
    FOR r ← list, next UNTIL r = NIL
      DO  next ← r.next;  SystemDefs.FreeHeapNode[r]  ENDLOOP;
    RETURN
    END;


  Position: TYPE = {after, before};
  Length: TYPE = CARDINAL [0..InsertLimit];

  levelStart, levelEnd: ARRAY Position OF ARRAY Length OF NodeIndex;


  AddLeaf: PROCEDURE [stack: StackRep, s: Symbol, thread: NodeIndex] RETURNS [stop: BOOLEAN] =
    BEGIN
    newLeaf: NodeIndex;
    saveNextNode: NodeIndex = nextNode;
    stack ← ParseStep[stack, s];
    IF stack.leaf = NullIndex OR ExistingConfiguration[stack] # NullIndex
      THEN  BEGIN  nextNode ← saveNextNode;  stop ← FALSE  END
      ELSE
        BEGIN
        newLeaf ← Allocate[stack.leaf, thread, s, stack.extension];
```

```
      SELECT parseMode FROM
        ATree => tree[newLeaf].aLeaf ← TRUE;
        BTree => tree[newLeaf].bLeaf ← TRUE;
        ENDCASE => ERROR;
      LinkHash[newLeaf];
      IF track THEN DisplayNode[newLeaf];
      stop ← RightScan[newLeaf];
      END;
  RETURN
  END;


GrowTree: PROCEDURE [p: Position, n: Length] RETURNS [stop: BOOLEAN] =
  BEGIN
  i: NodeIndex;
  j, jLimit: CARDINAL;
  stack: StackRep;
  state: State;
  rowList, r: RowHandle;
  s: Symbol;
  IF track THEN
    BEGIN  OPEN IODefs;
    WriteString["::generating length -- "L];  WriteDecimal[n];
    WriteChar[IF p = before THEN 'B ELSE 'A];  WriteChar[CR];
    END;
  rowList ← NIL;
  FOR i IN [levelStart[p][n-1] .. levelEnd[p][n-1])
    DO
    IF tree[i].symbol # 0 OR n = 1
      THEN
        BEGIN
        ENABLE  UNWIND => FreeRowList[rowList];
        rowList ← NIL;
        stack ← [leaf:i, extension:NullState];  state ← tree[i].state;
          DO
          j ← tState[state];  jLimit ← j + asstl[state].tlen;
          s ← tSymbol[jLimit-1];
          r ← SystemDefs.AllocateHeapNode[SIZE[RowRecord]];
          r↑ ← RowRecord[index:j, limit:jLimit, stack:stack, next:rowList];
          rowList ← r;
          IF s # DefaultMarker THEN EXIT;
          r.limit ← r.limit - 1;
          stack ← ActOnStack[stack, tAction[jLimit-1], 0];
          state ← stack.extension;
          ENDLOOP;
        UNTIL (r ← NextRow[rowList]) = NIL
          DO
          IF AddLeaf[r.stack, tSymbol[r.index], i] THEN GO TO found;
          r.index ← r.index + 1;
          ENDLOOP;
        END;
    REPEAT
      found => stop ← TRUE;
      FINISHED =>  stop ← FALSE;
    ENDLOOP;
  FreeRowList[rowList];  rowList ← NIL;  RETURN
  END;

CheckTree: PROCEDURE [p: Position, n: Length] RETURNS [stop: BOOLEAN] =
  BEGIN
  i: NodeIndex;
  IF track THEN
    BEGIN  OPEN IODefs;
    WriteString["::checking length -- "L];  WriteDecimal[n];
    WriteChar[IF p = before THEN 'B ELSE 'A];  WriteChar[CR];
    END;
  FOR i IN [levelStart[p][n] .. levelEnd[p][n])
    DO
    ENABLE TreeFull => CONTINUE;
    IF RightScan[i] THEN GO TO found;
    REPEAT
      found => stop ← TRUE;
      FINISHED =>  stop ← FALSE;
    ENDLOOP;
  RETURN
  END;
```

```
  Accept: PROCEDURE =
    BEGIN
    j: CARDINAL;
    p: NodeIndex;
    s: Symbol;
    discardBase: CARDINAL = best.nPassed;
    insertCount ← 1+InsertLimit;
    FOR p ← best.node, tree[p].last WHILE p > rTop
      DO
      IF (s ← tree[p].symbol) # 0 THEN
        BEGIN
        insertCount ← insertCount-1;
        newText[insertCount] ← SymbolRecord[s, P1Defs.TokenValue[s], inputLoc];
        END;
      ENDLOOP;
    scanBase ← discardBase;
    IF best.nDiscards # 0
      THEN
        BEGIN  OPEN IODefs;
        WriteString["Text deleted is: "L];
        FOR j IN [1 .. best.nDiscards]
          DO
          TypeSym[sourceText[scanBase].class];  scanBase ← scanBase + 1;
          ENDLOOP;
        END;
    IF insertCount <= InsertLimit
      THEN
        BEGIN  OPEN IODefs;
        IF scanBase # discardBase THEN WriteChar[CR];
        WriteString["Text inserted is: "L];
        FOR j IN [insertCount .. InsertLimit]
          DO  TypeSym[newText[j].class]  ENDLOOP;
        END;
    IF discardBase = 1
      THEN
        BEGIN
        insertCount ← insertCount-1;  newText[insertCount] ← sourceText[0];
        END;
    IF scanBase + best.nAccepted < scanLimit
      THEN P1Defs.ResetScanIndex[sourceText[scanBase+best.nAccepted].index];
    scanLimit ← scanBase + best.nAccepted;
    input ← RecoverInput;
--  WriteChar[CR];
    RETURN
    END;

  TypeSym: PROCEDURE [sym: Symbol] =
    BEGIN
    OPEN IODefs, lalrTable.scantable;
    i: CARDINAL;
    vocab: STRING = LOOPHOLE[@vocabbody, STRING];
    WriteChar[' ];
    IF sym ~IN [1..endmarker)
      THEN WriteDecimal[sym]
      ELSE
        FOR i IN [vocabindex[sym-1]..vocabindex[sym])
          DO  WriteChar[vocab[i]]  ENDLOOP;
    RETURN
    END;


--stack node indices
  rTop: NodeIndex;


  Recover: PROCEDURE =
    BEGIN
    ModeMap: ARRAY Position OF ParsingMode = [ATree, BTree];
    i: CARDINAL;
    place: Position;
    level: Length;
    inserts, discards: CARDINAL;
    stack: StackRep;
    threshold: CARDINAL;
```

```
    treeLimit ← LENGTH[tree] - CheckSize;
    FOR i IN [0 .. HashSize) DO hashTable[i] ← NullIndex ENDLOOP;
    rTop ← NullIndex;  nextNode ← maxNode ← 1;

    best.nAccepted ← 0;  best.nPassed ← 1;  best.mode ← ATree;
    sourceText[0] ← lastSymbol;
    sourceText[1] ← SymbolRecord[inputSymbol, inputValue, inputLoc];
    scanBase ← 1;  scanLimit ← 2;
    THROUGH [1 .. MaxScanLimit) DO Advance[]  ENDLOOP;
    FOR i IN [0 .. top)
      DO
      rTop ← Allocate[rTop, rTop, 0, s[i]];
      IF track THEN DisplayNode[rTop];
      ENDLOOP;
    parseMode ← BTree;
    levelStart[before][0] ← rTop ← FindNode[rTop, rTop, s[top]];
    tree[rTop].bLeaf ← TRUE;
    levelEnd[before][0] ← nextNode;
    parseMode ← ATree;
    stack ← ParseStep[[leaf:rTop, extension:NullState], lastSymbol.class];
    rTop ← FindNode[stack.leaf, rTop, stack.extension];
    tree[rTop].symbol ← lastSymbol.class;
    tree[rTop].aLeaf ← tree[rTop].bLeaf ← TRUE;
    levelStart[after][0] ← rTop;  levelEnd[after][0] ← nextNode;
    IF track THEN DisplayNode[rTop];

    FOR level IN [1 .. LAST[Length]]
      DO
      FOR place IN Position
        DO
        parseMode ← ModeMap[place];
        IF place = before THEN UnDiscard[];
        -- try simple insertion (inserts=level)
        levelStart[place][level] ← nextNode;
        IF GrowTree[place, level !TreeFull => CONTINUE] THEN GO TO found;
        levelEnd[place][level] ← nextNode;
        -- try discards followed by 0 or more insertions
        FOR discards IN [1 .. level)
          DO
          Discard[];
          IF CheckTree[place, level] THEN GO TO found;
          ENDLOOP;
        Discard[];
        IF place = after THEN Advance[];
        FOR inserts IN [0 .. level]
          DO
          IF CheckTree[place, inserts] THEN GO TO found;
          ENDLOOP;
        -- undo discards at this level
        FOR discards DECREASING IN [1..level] DO UnDiscard[] ENDLOOP;
        IF place = before THEN Discard[];
        ENDLOOP;
      REPEAT
        found =>  NULL;
        FINISHED =>
          BEGIN
          threshold ← (MinScanLimit+MaxScanLimit)/2;
          FOR discards IN [1..LAST[Length]] DO Discard[]; Advance[] ENDLOOP;
          UNTIL scanBase > DiscardLimit
            DO
            IF best.nAccepted >= threshold THEN GO TO found;
            Discard[];
            FOR inserts IN Length
              DO
              FOR place IN Position
                DO
                parseMode ← ModeMap[place];
                IF place = before THEN UnDiscard[];
                IF CheckTree[place, inserts] THEN GO TO found;
                IF place = before THEN Discard[];
                ENDLOOP;
              ENDLOOP;
            Advance[];
            threshold ← IF threshold > MinScanLimit THEN threshold-1 ELSE MinScanLimit;
            REPEAT
```

```
            found => NULL;
            FINISHED =>
              IF best.nAccepted < MinScanLimit
                THEN  BEGIN  best.mode ← ATree;  best.nPassed ← 1  END;
          ENDLOOP;
        END;
    ENDLOOP;

  RETURN
  END;

SyntaxError: PROCEDURE [abort: BOOLEAN] RETURNS [success: BOOLEAN] =
  BEGIN
  IF abort
    THEN
      BEGIN  OPEN IODefs;
      P1Defs.ErrorContext["Syntax Error"L, inputLoc];
      WriteString["... Parse abandoned."L];  WriteChar[CR];
      success ← FALSE
      END
    ELSE
      BEGIN
      tree ← DESCRIPTOR[SystemDefs.AllocateSegment[TreeSize*SIZE[StackNode]], TreeSize];
      hashTable ← DESCRIPTOR[SystemDefs.AllocateSegment[HashSize*SIZE[NodeIndex]], HashSize];
      Recover[ ! TreeFull => CONTINUE];
      SystemDefs.FreeSegment[BASE[hashTable]];
      P1Defs.ErrorContext["Syntax Error"L,
        sourceText[IF best.mode=BTree THEN 0 ELSE 1].index];
      IF (success ← best.nAccepted >= MinScanLimit)
        THEN  Accept[]
        ELSE  IODefs.WriteString["No recovery found."L];
      SystemDefs.FreeSegment[BASE[tree]];
      BEGIN OPEN IODefs;
--    WriteString["   ("L];  WriteDecimal[maxNode];  WriteChar[')];
      WriteChar[CR];
      END;
      END;
  RETURN
  END;

END.
```