```
-- BcdParse.Mesa  Edited by Johnsson on August 24, 1977  8:22 AM

DIRECTORY
  BcdControlDefs: FROM "bcdcontroldefs",
  BcdLALRDefs: FROM "bcdlalrdefs",
  IODefs: FROM "iodefs",
  SystemDefs: FROM "systemdefs";

DEFINITIONS FROM BcdLALRDefs;

BcdParse: PROGRAM
  IMPORTS BcdLALRDefs, IODefs, SystemDefs
  EXPORTS BcdControlDefs, BcdLALRDefs
  SHARES BcdLALRDefs =
  BEGIN

  ErrorLimit: CARDINAL = 25;

  InitialState: State = 1;
  FinalState: State = 0;

  currentState: State;
  inputSymbol, lhs: Symbol;
  DefaultMarker: Symbol = endmarker+1;
  inputLoc: CARDINAL;
  inputValue: UNSPECIFIED;
  qptr, top: CARDINAL;

  s: DESCRIPTOR FOR ARRAY OF State;
  l: DESCRIPTOR FOR ARRAY OF CARDINAL;
  v: DESCRIPTOR FOR ARRAY OF UNSPECIFIED;
  q: DESCRIPTOR FOR ARRAY OF ActionEntry;

  StackSize: INTEGER = 512;
  QueueSize: INTEGER = 256;

  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;

  input: PROCEDURE RETURNS [symbol: SymbolRecord];


  -- initialization/termination

  ParseInit: PROCEDURE [table: POINTER TO LALRTable] =
    BEGIN
    OPEN SystemDefs;
    lalrTable ← table;  -- for error reporting
    ScanInit[table];

      BEGIN  OPEN table;
      tState ← DESCRIPTOR[parsetable.tstate];
      asstl ← DESCRIPTOR[parsetable.asstl];
      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;

    s ← DESCRIPTOR[AllocateSegment[StackSize*SIZE[State]], StackSize];
    l ← DESCRIPTOR[AllocateSegment[StackSize*SIZE[CARDINAL]], StackSize];
    v ← DESCRIPTOR[AllocateSegment[StackSize*SIZE[UNSPECIFIED]], StackSize];
    q ← DESCRIPTOR[AllocateSegment[QueueSize*SIZE[ActionEntry]], QueueSize];
    AssignDescriptors[qd:q, vd:v, ld:l, pd:prodData];
    RETURN
    END;

  ParseErase: PROCEDURE =
    BEGIN
    OPEN SystemDefs;
    FreeSegment[BASE[q]];
    FreeSegment[BASE[v]];
    FreeSegment[BASE[l]];  FreeSegment[BASE[s]];
    RETURN
    END;

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

--  the main parsing procedures

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

    ParseInit[table];  input ← Atom;
    nErrors ← 0;  complete ← TRUE; errors ← FALSE;
    i ← top ← valid ← 0;  qptr ← 0;
    s[0] ← currentState ← InitialState;
    [inputSymbol, inputValue, inputLoc] ← input[].symbol;

    WHILE currentState # FinalState DO
      BEGIN
      j0 ← tState[currentState];
      FOR j IN [j0 .. j0 + asstl[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 qptr > 0
            THEN
              BEGIN
              FOR k IN (valid..i] DO s[k] ← s[top+(k-valid)] ENDLOOP;
              ProcessQueue[qptr, top];  qptr ← 0;
              END;
          top ← valid ← i ← i+1;
          v[i] ← inputValue;  l[i] ← inputLoc;
          [inputSymbol,inputValue,inputLoc]  ← input[].symbol;
          END;

      WHILE tj.rtag # ActionTag[FALSE, 0]
        DO
        IF qptr >= QueueSize THEN ExpandQueue[];
        q[qptr] ← tj;  qptr ← qptr + 1;
        i ← i-tj.rtag.plength;  -- pop 1 state per rhs symbol
        currentState ← s[IF i > valid THEN top+(i-valid) ELSE (valid ← i)];
```

```
        lhs ← prodData[tj.transition].lhs;
          BEGIN
          IF currentState <= laststate
            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[];
      s[m] ← currentState ← tj.transition;
      EXITS
        SyntaxError =>
          BEGIN  k ← top;  m ← 0;
          WHILE m < qptr AND ~q[m].rtag.reduce
            DO
            k ← k - q[m].rtag.plength + 1;  m ← m+1;
            ENDLOOP;
          IF m > 0
            THEN
              BEGIN
              s[k] ← ntentry[s[k-1], prodData[q[m-1].transition].lhs].transition;
              ProcessQueue[m, top];  qptr ← 0;
              END;
          top ← k;
          complete ← SyntaxError[(nErrors←nErrors+1)>ErrorLimit];
          errors ← TRUE;
          i ← valid ← top;  qptr ← 0;
          currentState ← s[i];
          [inputSymbol, inputValue, inputLoc] ← input[].symbol;
          IF ~complete THEN EXIT
          END;
      END;
    ENDLOOP;

    ProcessQueue[qptr, top];
    ParseErase[];
    RETURN
    END;

  ntentry: PROCEDURE [state: State, lhs: Symbol] RETURNS [ActionEntry] =
    BEGIN
    j: CARDINAL;
    IF state <= laststate 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;


  SyntaxStackOverflow: ERROR = CODE;

  ExpandStack: PROCEDURE =
    BEGIN
    ERROR SyntaxStackOverflow;
    END;

  ExpandQueue: PROCEDURE =
    BEGIN
    ERROR SyntaxStackOverflow;
    END;
```

```
-- error recovery

  NoMoreTreeSpace: SIGNAL = CODE;

--parameters of error recovery

  MinScanLimit: INTEGER = 2;
  MaxScanLimit: INTEGER = MinScanLimit+InsertLimit;
  InsertLimit: INTEGER = 3;
  DiscardLimit: INTEGER = 10;
  TreeSize: INTEGER = 256;

--monitor control

  track: BOOLEAN = FALSE;

  DisplayNode: PROCEDURE [n: NodeIndex] =
    BEGIN  OPEN IODefs;
    WriteString["::new node::"];
    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];  RETURN
    END;

--recovery primary data structures

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

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

  tree: DESCRIPTOR FOR ARRAY OF StackNode;

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

  newText: ARRAY [0..InsertLimit) OF SymbolRecord;
  lookAhead: ARRAY [0..MaxScanLimit] OF SymbolRecord;
  discardSymbol: ARRAY [0..DiscardLimit) OF SymbolRecord;

  scanLimit, discardCount: CARDINAL;
  endFile: BOOLEAN;

--stack node indices
  nextNode, rTop: NodeIndex;


  ParseStep: PROCEDURE [input: Symbol, node: NodeIndex] RETURNS [NodeIndex, State] =
    BEGIN
    currentNode: NodeIndex ← node;
    currentState: State ← tree[node].state;
    j, j0: CARDINAL;
    mState: State;
    lhs: Symbol;
    tj: ActionEntry;
    count: CARDINAL ← 0;
    newSymbol: BOOLEAN ← FALSE;
    WHILE ~newSymbol
      DO
      IF currentState = FinalState THEN
        RETURN [NullIndex, FinalState];
      j0 ← tState[currentState];
      FOR j IN [j0..j0+asstl[currentState].tlen)
        DO
        SELECT tSymbol[j] FROM
          input, DefaultMarker => EXIT;
          ENDCASE;
        REPEAT
```

```
        FINISHED => RETURN [NullIndex, InitialState];
      ENDLOOP;
    tj ← tAction[j];
    IF ~tj.rtag.reduce
      THEN  --next state or shift reduce
        BEGIN
        IF count = 0
          THEN count ← 1
          -- shift after a reduce, insert nonterminal
          ELSE currentNode ← allocate[currentNode, node, 0, mState];
        newSymbol ← TRUE;
        END;
    WHILE tj.rtag # ActionTag[FALSE,0]
      DO  -- perform reductions
      WHILE count < tj.rtag.plength
        DO
        currentNode ← tree[currentNode].father;
        count ← count+1;
        ENDLOOP;
      currentState ← tree[currentNode].state;
      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;
      count ← 1;
      ENDLOOP;
    currentState ← mState ← tj.transition;
    IF input = DefaultMarker THEN EXIT;
    ENDLOOP;
  RETURN [currentNode, tj.transition]
  END;

RightScan: PROCEDURE [node: NodeIndex] RETURNS [BOOLEAN] =
  BEGIN
  savedNextNode: NodeIndex = nextNode;
  i: CARDINAL;
  state: State;
  FOR i IN [0 .. scanLimit]
    DO
    [node, state] ← ParseStep[lookAhead[i].class, node];
    IF node = NullIndex THEN
      BEGIN  nextNode ← savedNextNode;
      RETURN [state=FinalState
        AND (i = scanLimit OR lookAhead[i+1].class = endmarker)]
      END;
    node ← allocate[node, 0, lookAhead[i].class, state];
    ENDLOOP;
  nextNode ← savedNextNode;  RETURN [TRUE]
  END;

discard: PROCEDURE [advance: BOOLEAN] =
  BEGIN
  j: CARDINAL;
  discardSymbol[discardCount] ← lookAhead[0];
  FOR j IN [0 .. scanLimit) DO lookAhead[j] ← lookAhead[j+1] ENDLOOP;
  endFile ← lookAhead[0].class = endmarker;
  IF ~advance
    THEN  scanLimit ← scanLimit-1
    ELSE
      BEGIN
      lookAhead[scanLimit] ← input[];
      IF track THEN
        BEGIN  OPEN IODefs;
        WriteString["::discarding symbol -- "];
```

```
          TypeSym[discardSymbol[discardCount].class];  WriteChar[CR];
          END;
        END;
    discardCount ← discardCount+1;
    RETURN
    END;

  undiscard: PROCEDURE =
    BEGIN
    j: CARDINAL;
    scanLimit ← scanLimit+1;
    FOR j DECREASING IN (0..scanLimit]
      DO lookAhead[j] ← lookAhead[j-1] ENDLOOP;
    discardCount ← discardCount-1;
    lookAhead[0] ← discardSymbol[discardCount];
    IF track THEN
      BEGIN  OPEN IODefs;
      WriteString["::recovering symbol -- "];
      TypeSym[discardSymbol[discardCount].class];  WriteChar[CR];
      END;
    RETURN
    END;

  allocate: PROCEDURE [parent, pred: NodeIndex, terminal: Symbol, stateno: State] RETURNS [index: NodeI
**ndex] =
    BEGIN
    IF (index ← nextNode) >= TreeSize THEN SIGNAL NoMoreTreeSpace;
    tree[index] ← StackNode[parent, pred, stateno, terminal, NullIndex];
    nextNode ← nextNode+1;  RETURN
    END;

  levelStart, levelEnd: ARRAY [0..InsertLimit] OF NodeIndex;

  GenerateTree: PROCEDURE [level: CARDINAL] RETURNS [BOOLEAN, NodeIndex] =
    BEGIN
    i, n, n1, n2, newnode, stacktop, newtop, savenextNode: NodeIndex;
    htIndex: NodeIndex;
    j, jlimit: CARDINAL;
    state, newstate, s1, s2: State;
    IF track THEN
      BEGIN  OPEN IODefs;
      WriteString["::generating level -- "];
      WriteDecimal[level];  WriteChar[CR];
      END;
    FOR i IN [levelStart[level-1] .. levelEnd[level-1])
      DO
      IF tree[i].symbol # 0 OR level = 1
        THEN
          BEGIN
          stacktop ← i;  state ← tree[i].state;
          j ← tState[state];  jlimit ← j + asstl[state].tlen;
          WHILE j < jlimit
            DO
              BEGIN  savenextNode ← nextNode;
              [newtop, newstate] ← ParseStep[tSymbol[j], stacktop];
              IF newtop = NullIndex THEN
                IF newstate = FinalState AND endFile
                  THEN RETURN [TRUE, i]
                  ELSE GO TO next;      -- input invalid in this context
              -- check if this new state has already been seen
              htIndex ← newstate MOD HashSize;
              FOR n ← hashTable[htIndex], tree[n].link UNTIL n = NullIndex
                DO
                s1 ← newstate;  s2 ← tree[n].state;
                n1 ← newtop;  n2 ← tree[n].father;
                  DO
                  IF s1 # s2 THEN EXIT;
                  IF n1 = n2 THEN GO TO duplicate;
                  s1 ← tree[n1].state;  s2 ← tree[n2].state;
                  n1 ← tree[n1].father;  n2 ← tree[n2].father;
                  ENDLOOP;
                ENDLOOP;
              newnode ← allocate[newtop, i, tSymbol[j], newstate];
              tree[newnode].link ← hashTable[htIndex];
              hashTable[htIndex] ← newnode;
              IF track THEN DisplayNode[newnode];
```

```
            IF tSymbol[j] = DefaultMarker
              THEN
                BEGIN
                tree[newnode].symbol ← 0;
                stacktop ← newnode;  state ← newstate;
                j ← tState[state];  jlimit ← j + asstl[state].tlen;
                END
              ELSE  -- check if input acceptable in new state
                IF RightScan[newnode]
                  THEN RETURN [TRUE, newnode]
                  ELSE GO TO next;
            EXITS
              next => j ← j+1;
              duplicate =>
                BEGIN  nextNode ← savenextNode;  j ← j+1;
                END;
            END;
          ENDLOOP;
        END;
    ENDLOOP;
  RETURN [FALSE, NullIndex]
  END;

CheckTree: PROCEDURE [level: CARDINAL] RETURNS [BOOLEAN, NodeIndex] =
  BEGIN
  i: NodeIndex;
  IF track THEN
    BEGIN  OPEN IODefs;
    WriteString["::checking level -- "];
    WriteDecimal[level];  WriteChar[CR];
    END;
  FOR i IN [levelStart[level] .. levelEnd[level])
    DO
    IF RightScan[i] THEN RETURN [TRUE, i];
    ENDLOOP;
  RETURN [FALSE, NullIndex]
  END;


scanCount, insertCount: CARDINAL;

recoverinput: PROCEDURE RETURNS [sym: SymbolRecord] =
  BEGIN
  IF insertCount < InsertLimit
    THEN
      BEGIN  sym ← newText[insertCount];
      insertCount ← insertCount+1;
      END
    ELSE
      BEGIN  sym ← lookAhead[scanCount];
      IF (scanCount ← scanCount+1) > scanLimit THEN input ← Atom;
      END;
  RETURN
  END;

accept: PROCEDURE [node: NodeIndex] =
  BEGIN
  j: CARDINAL;
  p: NodeIndex;
  s: Symbol;
  insertCount ← InsertLimit;
  FOR p ← node, tree[p].last WHILE p > rTop
    DO
    IF (s ← tree[p].symbol) # 0 THEN
      BEGIN
      insertCount ← insertCount-1;
      newText[insertCount] ← SymbolRecord[s, TokenValue[s], inputLoc];
      END;
    ENDLOOP;
  IF discardCount > 0
    THEN
      BEGIN  OPEN IODefs;
      WriteString["Text deleted is: "];
      FOR j IN [0 .. discardCount)
        DO
        TypeSym[discardSymbol[j].class];
```

```
        ENDLOOP;
      WriteChar[CR];
      END;
  IF insertCount < InsertLimit
    THEN
      BEGIN  OPEN IODefs;
      WriteString["Text inserted is: "];
      FOR j IN [insertCount .. InsertLimit)
        DO
        TypeSym[newText[j].class];
        ENDLOOP;
      WriteChar[CR];
      END;
  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;


SyntaxError: PROCEDURE [abort: BOOLEAN] RETURNS [success: BOOLEAN] =
  BEGIN
  i, level: CARDINAL;
  inserts, discards: CARDINAL;
  n: NodeIndex;
  ErrorContext[FALSE];
  IF abort THEN
    BEGIN  OPEN IODefs;
    WriteString["... Parse abandoned."];  WriteChar[CR];
    RETURN [FALSE]
    END;
  -- setup for recovery
    tree ← DESCRIPTOR[SystemDefs.AllocateSegment[TreeSize*SIZE[StackNode]], TreeSize];
    hashTable ← DESCRIPTOR[SystemDefs.AllocateSegment[HashSize*SIZE[NodeIndex]], HashSize];
    FOR i IN [0 .. HashSize) DO hashTable[i] ← NullIndex ENDLOOP;
    rTop ← NullIndex;  nextNode ← 1;
    lookAhead[0] ← SymbolRecord[inputSymbol, inputValue, inputLoc];
    endFile ← inputSymbol = endmarker;
    scanLimit ← MinScanLimit;
    FOR i IN (0 .. scanLimit] DO lookAhead[i] ← input[]  ENDLOOP;
    FOR i IN [0 .. top]
      DO
      rTop ← allocate[rTop, rTop, 0, s[i]];
      IF track THEN DisplayNode[rTop];
      ENDLOOP;
    hashTable[tree[rTop].state MOD HashSize] ← rTop;
    discardCount ← 0;
    levelStart[0] ← rTop;  levelEnd[0] ← nextNode ← rTop+1;
    FOR level IN [1..InsertLimit]
      DO
      -- try simple insertion (inserts=level)
      levelStart[level] ← nextNode;
      [success, n] ← GenerateTree[level !NoMoreTreeSpace => CONTINUE];
      levelEnd[level] ← nextNode;
      IF success THEN GO TO found;
      -- try discards followed by 0 or more insertions
      FOR discards IN [1 .. level]
        DO
        discard[discards=level];
        FOR inserts IN [(IF discards=level THEN 0 ELSE level) .. level]
          DO
          [success, n] ← CheckTree[inserts !NoMoreTreeSpace => CONTINUE];
          IF success THEN GO TO found;
```

```
          ENDLOOP;
        ENDLOOP;
      -- undo discards at this level
      THROUGH [1..level] DO undiscard[] ENDLOOP;
      REPEAT
        found => NULL;
        FINISHED =>
          BEGIN
          FOR i IN [1..InsertLimit] DO discard[i#1] ENDLOOP;
          success ← FALSE;
          UNTIL success OR discardCount >= DiscardLimit
            DO
            discard[TRUE];
            FOR inserts IN [0..InsertLimit]
              DO
              [success, n] ← CheckTree[inserts !NoMoreTreeSpace => CONTINUE];
              IF success THEN EXIT;
              ENDLOOP;
            ENDLOOP;
          END;
      ENDLOOP;
  -- clean up state
    IF success
      THEN
        BEGIN  accept[n];  scanCount ← 0;  input ← recoverinput;
        END
      ELSE
        BEGIN  OPEN IODefs;
        WriteString["No recovery found."];  WriteChar[CR];
        END;
    SystemDefs.FreeSegment[BASE[hashTable]];
    SystemDefs.FreeSegment[BASE[tree]];
  RETURN
  END;

END...
```