```
-- file DIInterpreter.Mesa
-- last modified by
--                  Sandman, April 9, 1978  12:37 AM
--                  Barbara, July 12, 1978  2:35 PM

DIRECTORY
  DebugMiscDefs: FROM "debugmiscdefs" USING [WriteEOL],
  DIActionDefs: FROM "diactiondefs" USING [
    addressofItem, assignvalue, baseItem, CleanUp, dereferenceItem, desc1Item,
    desc2Item, evaluateExpList, getLiteral, getLongLiteral, getStringLiteral,
    incrementList, lengthItem, LookupId, loopholeItem, loopholeUnspecItem,
    memItem, minusItem, NILesp, performAddOp, performMultOp, pointertoType,
    popevalstack, poptypestack, printInterval, printOctal, pushevalstack,
    pushtypestack, qualifyItem, ResetStacks, SearchFileForId,
    SearchFileForType, SearchForType, SearchForVariantType, SearchFrameForId,
    setIntervalBit, setPredefined, startList, typeOp],
  DIDefs: FROM "didefs" USING [
    ConstOrQual, DescriptorAssigner, Operator, ParseError, ParseHandle,
    ParseObject, QueueProcessor, thereESPointer],
  DILALRDefs: FROM "dilalrdefs" USING [ActionEntry, ProductionInfo],
  DILitDefs: FROM "dilitdefs" USING [LTIndex, STIndex];

DIInterpreter: PROGRAM
  IMPORTS DebugMiscDefs, DIActionDefs, DIDefs
  EXPORTS DIDefs
  SHARES DILALRDefs =
  BEGIN

  v: DESCRIPTOR FOR ARRAY OF UNSPECIFIED; --parse stack
  l: DESCRIPTOR FOR ARRAY OF CARDINAL; --sourceline index
  h: DESCRIPTOR FOR ARRAY OF DIDefs.ConstOrQual; --alternate stack
  q: DESCRIPTOR FOR ARRAY OF DILALRDefs.ActionEntry;  --reduction rules
  proddata: DESCRIPTOR FOR ARRAY OF DILALRDefs.ProductionInfo; --production rules

  parse:  DIDefs.ParseObject ← [qp: QueueProcessing,
    da: AssignDescriptors];

  TwoParse: PUBLIC PROCEDURE RETURNS [DIDefs.ParseHandle] =
    BEGIN RETURN[@parse]; END;

  AssignDescriptors: DIDefs.DescriptorAssigner =
    BEGIN q ← qd; v ← vd; l ← ld; proddata ← pd; h ← hd; RETURN END;

 -- the interpretation rules

 QueueProcessing: PUBLIC DIDefs.QueueProcessor=
   BEGIN OPEN DIActionDefs, DIDefs;
   rule: [0..377B];
   i: CARDINAL;
   FOR i IN [0..qI)
     DO
     top ← top-q[i].rtag.plength+1;
     rule ← proddata[q[i].transition].rule;
     IF parsingInterval THEN CheckRuleForInterval[rule, top];
     SELECT rule FROM

  0  => -- goal          ::= stmtlist
    -- no action
    EXIT;

  1  => -- stmtlist      ::= stmt
    -- all finished
    BEGIN CleanUp[]; DebugMiscDefs.WriteEOL[]; END;

  2  => -- stmtlist      ::= stmtlist ; stmt
    -- clear the way for the next statement
    BEGIN DebugMiscDefs.WriteEOL[]; ResetStacks[]; END;

  3  => -- stmt          ::= exp
    -- apply proc to the value of exp
    proc[popevalstack[ ! NILesp => CONTINUE]];

  4  => -- stmt               ::= lhs ← exp
    -- take value of exp from stack, store into address of lhs
    BEGIN
    IF h[top] # var THEN SIGNAL DIDefs.ParseError[l[top]];
    assignvalue[popevalstack[], LOOPHOLE[popevalstack[],thereESPointer]];
    END;

  5  => -- exp              ::= sum
    NULL;

  20  => -- sum             ::= product
    NULL;

  21  => -- sum             ::= sum addop product
    -- combine two values on stack with addop, put result on stack
    BEGIN
    IF h[top] = var OR h[top+2] = var THEN h[top] ← var;
    pushevalstack[performAddOp[popevalstack[], popevalstack[],LOOPHOLE[v[top+1], Operator]]];
    END;

  22  => -- addop             ::= +
    -- put plus on stack
    v[top] ← Operator[plus];

  23 => -- addop            ::= -
    -- put minus on stack
    v[top] ← Operator[minus];

  24 => -- product  ::= factor
    NULL;

  25 => -- product             ::= product multop factor
    -- combine two values on stack with multop, put result on stack
    BEGIN
    IF h[top] = var OR h[top+2] = var THEN h[top] ← var;
    pushevalstack[performMultOp[popevalstack[], popevalstack[],LOOPHOLE[v[top+1], Operator]]];
    END;

  26 => -- multop            ::= *
    -- put times on stack
    v[top] ← Operator[times];

  27 => -- multop            ::= /
    -- put div on stack
    v[top] ← Operator[div];

  28 => -- multop            ::= MOD
    -- put mod on stack
    v[top] ← Operator[mod];

  30 => -- factor            ::= primary
    NULL;

  31 => -- factor            ::= - primary
    -- take value off stack, put back -value
    BEGIN
    h[top] ← h[top+1];
    pushevalstack[minusItem[popevalstack[]]];
    END;

  40 => -- primary             ::= lhs
    NULL;

  41 => -- primary             ::= ( exp )
    -- noop
    h[top] ← h[top+1];

  43 => -- primary             ::= builtincall
    NULL;

  44 => -- primary            ::=@ lhs
    -- put address of value onto stack
    BEGIN
    IF h[top+1] # var THEN SIGNAL DIDefs.ParseError[l[top]] ELSE h[top] ← var;
    pushevalstack[addressofItem[LOOPHOLE[popevalstack[],thereESPointer]]];
    END;

  50 => -- builtincall           ::= LENGTH [ lhs ]
    -- evaluate LENGTH of id on stack, put back # of elements
    BEGIN
    IF h[top+2] # var THEN SIGNAL DIDefs.ParseError[l[top]] ELSE h[top] ← var;
    pushevalstack[lengthItem[popevalstack[]]];
    END;

  51 => -- builtincall          ::= BASE [ lhs ]
    -- evaluate BASE of id on stack, put back pointer value
    BEGIN
    IF h[top+2] # var THEN SIGNAL DIDefs.ParseError[l[top]] ELSE h[top] ← var;
    pushevalstack[baseItem[popevalstack[]]];
    END;

  52 => -- builtincall          ::= DESCRIPTOR [ exp ]
    -- create a DESCRIPTOR for id on stack, put back [loc,length]
    BEGIN
    IF h[top+2] # var THEN SIGNAL DIDefs.ParseError[l[top]] ELSE h[top] ← var;
    pushevalstack[desc1Item[LOOPHOLE[popevalstack[],thereESPointer]]];
    END;

  53 => -- builtincall          ::= DESCRIPTOR [ exp , exp ]
    -- create a DESCRIPTOR for id on stack, put back [loc,length]
    BEGIN
    h[top] ← var;
    pushevalstack[desc2Item[popevalstack[], popevalstack[]]];
    END;

  54 => -- builtincall          ::= typeop [ typespec ]
    -- apply typeop to value on typestack, put result on evalstack
    BEGIN
    h[top] ← var;
    pushevalstack[typeOp[LOOPHOLE[v[top], Operator], poptypestack[]]];
    END;

  65 => -- typeop            ::= SIZE
    -- save operator - value not yet on stack
    v[top] ← Operator[size];

  71 => -- lhs              ::= id
    --  lookup id, save info with its value
    BEGIN
    h[top] ← var;
    pushevalstack[LookupId[LOOPHOLE[v[top], DILitDefs.STIndex]]];
    END;

  72 => -- lhs            ::= num
    -- put value of numeric literal on stack as UNSPECIFIED
    BEGIN
    h[top] ← num;
    pushevalstack[getLiteral[num, LOOPHOLE[v[top], DILitDefs.LTIndex]]];
    END;

  73 => -- lhs              ::= lnum
    --  put value of long numeric literal on stack as LONG INTEGER
    BEGIN
    h[top] ← lnum;
    pushevalstack[getLongLiteral[LOOPHOLE[v[top], DILitDefs.LTIndex]]];
    END;

  74 => -- lhs            ::= char
    -- put value of character literal on stack as CHARACTER
    BEGIN
    h[top] ← char;
    pushevalstack[getLiteral[char, LOOPHOLE[v[top], DILitDefs.LTIndex]]];
    END;

  75 => -- lhs            ::= sr
    -- put value of string literal on stack as SubString
    BEGIN
    h[top] ← sr;
    pushevalstack[getStringLiteral[LOOPHOLE[v[top], DILitDefs.STIndex]]];
    END;

  76 => -- lhs              ::= ( exp ) qualifier
    -- noop
    SELECT h[top+1] FROM
      lnum, num, char =>
        SELECT h[top+3] FROM
```

```
          deref, loophole, loopholetype => h[top] ← var;
          ENDCASE => SIGNAL DIDefs.ParseError[l[top]];
      sr =>
        SELECT h[top+3] FROM
          deref, loophole, loopholetype, explist => h[top] ← var;
          ENDCASE => SIGNAL DIDefs.ParseError[l[top]];
      ENDCASE;

  77 => -- lhs              ::= lhs qualifier
    -- noop
    SELECT h[top] FROM
      lnum, num, char =>
        SELECT h[top+1] FROM
          deref, loophole, loopholetype => h[top] ← var;
          ENDCASE => SIGNAL DIDefs.ParseError[l[top]];
      sr =>
        SELECT h[top+1] FROM
          deref, loophole, loopholetype, explist => h[top] ← var;
          ENDCASE => SIGNAL DIDefs.ParseError[l[top]];
      ENDCASE;

  80 => -- lhs              ::= MEMORY [ interval ]
    -- find values in MEMORY for interval - only valid at top level
    BEGIN
    h[top] ← num;
    printOctal[popevalstack[], popevalstack[]];
    END;

  81 => -- lhs              ::= MEMORY [ exp ]
    -- find value in MEMORY for exp
    BEGIN
    h[top] ← var;
    pushevalstack[memItem[popevalstack[]]];
    END;

  83 => -- lhs              ::= id $ id
    -- go to file named by first id to lookup second id
    BEGIN
    h[top] ← var;
    pushevalstack[SearchFileForId[v[top], v[top+2]]];
    END;

  84 => -- lhs              ::= num $ id
    -- go to file named by global frame num to lookup second id
    BEGIN
    h[top] ← var;
    pushevalstack[SearchFrameForId[LOOPHOLE[v[top], DILitDefs.LTIndex], v[top+2]]];
    END;

  90 => -- qualifier          ::= . id
    -- put field of record on stack
    BEGIN
    h[top] ← dot;
    pushevalstack[qualifyItem[LOOPHOLE[popevalstack[],thereESPointer],
      LOOPHOLE[v[top+1], DILitDefs.STIndex], locals]];
    END;

  91 => -- qualifier          ::= ↑
    -- dereference value on stack
    BEGIN
    h[top] ← deref;
    pushevalstack[dereferenceItem[LOOPHOLE[popevalstack[],thereESPointer]]];
    END;

  92 => -- qualifier          ::= %
    -- loophole value to be an UNSPECIFIED
    BEGIN
    h[top] ← loophole;
    pushevalstack[loopholeUnspecItem[popevalstack[]]];
    END;

  93 => -- qualifier          ::= % typespec
    -- change type of value on stack to be typespec
    BEGIN
    h[top] ← loopholetype;
    pushevalstack[loopholeItem[popevalstack[],poptypestack[]]];
    END;

  94 => -- qualifier          ::= [ explist ]
    -- qualify value on stack (procedure or array or string) - note listsize
    BEGIN
    h[top] ← explist;
    pushevalstack[evaluateExpList[]];
    END;

  95 => -- qualifier         ::= [ interval ]
    -- apply interval op to value - valid only at top level
    BEGIN
    h[top] ← interval;
    printInterval[popevalstack[], popevalstack[], popevalstack[]];
    END;

  105 => -- typespec          ::= typeid
    NULL;

  107 => -- typespec          ::= typeconstruct
    NULL;

  110 => -- typeid            ::= INTEGER
    -- save type of INTEGER;
    pushtypestack[setPredefined[integer]];

  111 => -- typeid            ::= CARDINAL
    -- save type of CARDINAL;
    pushtypestack[setPredefined[cardinal]];

  112 => -- typeid            ::= CHARACTER
    -- save type of CHARACTER;
    pushtypestack[setPredefined[character]];

  113 => -- typeid           ::= BOOLEAN
    -- save type of BOOLEAN;
    pushtypestack[setPredefined[boolean]];

  114 => -- typeid            ::= STRING
    -- save type of STRING;
    pushtypestack[setPredefined[string]];

  115 => -- typeid            ::= UNSPECIFIED
    -- save type of UNSPECIFIED;
    pushtypestack[setPredefined[unspecified]];

  116 => -- typeid            ::= id $ id
    -- go to file to find type
    pushtypestack[SearchFileForType[LOOPHOLE[v[top], DILitDefs.STIndex], LOOPHOLE[v[top+2], DILitDefs.S
**TIndex]]];

  119 => -- typeid           ::= id
    -- look for type
    pushtypestack[SearchForType[LOOPHOLE[v[top], DILitDefs.STIndex]]];

  120 => -- typeid           ::= id typeid
    -- add the variant to the type
    pushtypestack[SearchForVariantType[LOOPHOLE[v[top], DILitDefs.STIndex], poptypestack[]]];

  125 => -- typeconstruct        ::= @ typespec
    -- construct a POINTER TO TYPE for typespec
    pushtypestack[pointertoType[poptypestack[]]];

  140 => -- explist             ::= exp
    -- start list
    pushevalstack[startList[1]];

  141 => -- explist             ::= explist , exp
    -- increment list size
    incrementList[];

  142 => -- explist             ::=
    -- empty expression list
    pushevalstack[startList[0]];

  143 => -- interval           ::= exp .. exp
  -- noop - have start address and finish address on stack already
  BEGIN
  parsingInterval ← TRUE;
  intervalState ← interval;
  intervalRule ← 0;
  END;

144 => -- interval          ::= exp ! exp
  -- note interval type
  BEGIN
  parsingInterval ← TRUE;
  intervalState ← interval;
  intervalRule ← 0;
  pushevalstack[setIntervalBit[popevalstack[]]];
  END;

ENDCASE => SIGNAL ParseError[l[top]]; -- error or unimplemented
    ENDLOOP;
  RETURN
  END;


-- Interval Checking

parsingInterval: BOOLEAN ← FALSE;

NextIntervalRule: PACKED ARRAY [0..5] OF [0..377B] = [
  40,  --  primary ::= lhs
  30,  --  factor ::= primary
  24,  --  product ::= factor
  20,  --  sum ::= product
  5,   --  exp ::= sum
  3];  --  stmt ::= exp

intervalState: {primary, interval, qualifier};

intervalRule: CARDINAL;

CheckRuleForInterval: PROCEDURE [rule: [0..377B], top: CARDINAL] =
  BEGIN
  IF rule < 3 THEN BEGIN parsingInterval ← FALSE; RETURN END;
  BEGIN
    SELECT intervalState FROM
      primary =>
        SELECT rule FROM
          NextIntervalRule[intervalRule] => intervalRule ← intervalRule + 1;
          ENDCASE => GO TO Error;
      interval =>
        SELECT rule FROM
          80 => intervalState ← primary;
          95 => intervalState ← qualifier;
          ENDCASE => GO TO Error;
      qualifier =>
        SELECT rule FROM
          77, 76 => intervalState ← primary;
          ENDCASE => GO TO Error;
      ENDCASE => ERROR;
  EXITS
    Error =>
      BEGIN
      parsingInterval ← FALSE;
      SIGNAL DIDefs.ParseError[l[top]];
      END;
  END;
  RETURN
  END;

END...
```