Design and implement a recursive descent parser for PLS. The parser should use your existing lexical analyzer to get the next token. I suggest that you attack this assignment in three phases:
1) construct an "acceptor" for mathematical type expressions first, then
2) construct an "acceptor" for relational expressions (this acceptor will call (1)), and then
3) extend the main program to include the other constructs for PLS.

Thus, in the recursive descent parser, "math-expression" and "rel-expression" should be regarded as nonterminal symbols with predefined procedures, mathexpr and relexpr, that recognize them.

The final calling sequence will be:

```
+---------+  parse  +---------+
|         |           |         |
| rel-expr|           V
|         | +-----------+     |
|          | math-expr |     V
|          +-----------+     |
|               |            V
|               |       +----------+
|               ------->| lexical  |
|---------------------->| analyzer |
+----------+
```

Document your parser the appropriate recursive descent grammar. Use a format similar to that given in class (or in the text). You may use "{( ... )}" to denote repetition, and "[ ... ]" to denote optional parts. Your test files, rd*.inp, will be in the anonymous ftp director CS4304 on arthur.cs.vt.edu.

You will implement procedure tracing and error recovery action messages as discussed in class. You will also print out appropriate error messages whenever an error occurs.
program expreval();
ch : char;
val : real;

procedure getnxtchar; /* rudimentary lexical analyzer */
begin
  ch := ' ';  
  while (not eof) and (ch = ' ') do 
    read(ch);
  if eof then ch := ' ';
end;

function id: boolean; /* actually ids can only be */
begin                  /* single digit integers   */
  if (ch >= '0') and (ch <= '9') then begin 
    val := atoi(ch);
    id := true;
  end;
end;

function addop: boolean;
begin
  if (ch = '+') or (ch = '-') then /* is char an */
    addop := true                 /* addop?    */
  else
    addop := false;
end;

function multop: boolean;
begin
  if (ch = '*') or (ch = '/') then /* is char a */
    multop := true               /* multop?   */
  else
    multop := false;
end;

function mathexpr:real;
begin
  sum := mterm;
  while addop do begin
    if ch = '+' then begin 
      getnxtchar;
      sum := sum + mterm
    end
    else begin 
      getnxtchar;
      sum := sum - mterm;
    end;
  end;
  mathexpr := sum;
end;
function mterm /* real */;
  product : real;
begin
  product := mfactor;
  while multop do begin
    if ch = '*' then begin
      getnxtchar;
      product := product * mfactor
    end
    else begin
      getnxtchar;
      product := product / mfactor;
    end;
  end;
  mterm := product;
end;

function mfactor /* real */;
begin
  if id then begin /* id */
    getnxtchar;
    mfactor := val
  end
  else if addop then /* unary + */
    if ch = '+' then begin
      getnxtchar;
      mfactor := mfactor
    end
    else begin /* unary - */
      getnxtchar;
      mfactor := - mfactor
    end
  else if ch = '(' then begin /* parenthesized */
    getnxtchar;        /* expression    */
    mfactor := mathexpr;
    if ch = ')' then
      getnxtchar
    else
      writeln(' *** ERROR *** NO ')
  end
  else
    writeln(' *** ERROR *** INVALID EXPRESSION');
end;

/* begin main procedure */
begin;
  getnxtchar;
  writeln(' Total is : ',mathexpr);
end.