Calc Gotchas

```
L = list(); push(L, 9); push(L, 8); push(L, 7); push(L, 6);
mat M[] = {1, 2, 3, 4, 5};
A = assoc(); A[0]=0; A[1]=1; A[2]=2; A[3]=3; A[4]=4;
```

The statements a = M[1];, b = L[[1]];, c = L[1];, and d = A[2]; all work as expected. Neither M[1] = M[1] + 1; nor L[[1]] = L[[1]] + 1; nor A[1] = A[1] + 1; will work. But M[0] = M[0] + 1; does work as expected.

Suppose you are using L as a stack and wish to swap the top two items, such that pop(L) would return 7 rather than 6.

The following statements do not work

```
swap(L[[0]], L[[1]]); swap(L[0], L[1]);
```

Instead use

```
insert(L, 1, pop(L));
```

or better yet use

```
if(size(L)>1) insert(L, 1, pop(L));
```

Generally, to accomplish L[offset] = L[offset] operand value;, you could use

```
if(size(L)>offset) insert(L, offset, delete(L, offset) operand value);
```

To use M as an array, as in the above example, you must use something like

```
mat M[] = {M[0], M[1] + 1, M[2]};
```

Command sequence

This is a sequence of any the following command formats, where each command is terminated by a semicolon or newline. Long command lines can be extended by using a back-slash followed by a newline character. When this is done, the prompt shows a double angle bracket to indicate that the line is still in progress. Certain cases will automatically prompt for more input in a similar manner, even without the back-slash. The most common case for this is when a function is being defined, but is not yet completed.

Each command sequence terminates only on an end of file. In addition, commands can consist of expression sequences, which are described in the next section.

```
define a function
-----------------
define function(params) { body }
define function(params) = expression

  This first form defines a full function which can consist
  of declarations followed by many statements which implement
  the function.

  The second form defines a simple function which calculates
  the specified expression value from the specified parameters.
  The expression cannot be a statement.  However, the comma
  and question mark operators can be useful.      Examples of
  simple functions are:

        define sumcubes(a, b) = a^3 + b^3
        define pimod(a) = a % pi()
        define printnum(a, n, p)
        {
            if (p == 0) {
              print a: "^": n, "=", a^n;
            } else {
              print a: "^": n, "mod", p, "=", pmod(a,n,p);
            }
        }

show information
----------------
show item

  This command displays some information where 'item' is
  one of the following:

        blocks            unfreed named blocks
        builtin           built in functions
        config            config parameters and values
        constants         cache of numeric constants
        custom            custom functions if calc -C was used
        errors            new error-values created
        files       open files, file position and sizes
        function          user-defined functions
        globaltypes       global variables
```

```
      objfunctions      possible object functions
      objtypes          defined objects
      opcodes func      internal opcodes for function `func'
      sizes       size in octets of calc value types
      realglobals       numeric global variables
      statics           unscoped static variables
      numbers           calc number cache
      redcdata          REDC data defined
      strings           calc string cache
      literals          calc literal cache
```

Only the first 4 characters of item are examined, so:

```
      show globals
      show global
      show glob
```

do the same thing.

define - command keyword to start a function definition

```
SYNTAX
    define fname([param_1 [= default_1], ...]) = [expr]
    define fname([param_1 [= default_1], ...]) { [statement_1 ... ] }

TYPES
    fname         identifier, not a builtin function name
    param_1, ...  identifiers, no two the same
    default_1, ...      expressions
    expr          expression
    statement_1, ...    statements

DESCRIPTION
    The intention of a function definition is that the identifier fname
    becomes the name of a function which may be called by an expression
    of the form  fname(arg_1, arg_2, ...), where arg_1, arg_2, ... are
    expressions (including possibly blanks, which are treated as
    null values).  Evaluation of the function begins with evaluation
    of arg_1, arg_2, ...; then, in increasing order of i, if arg_i is
    null-valued and  "= default_i" has been included in the definition,
    default_i is evaluated and its value becomes the value of arg_i.
    The instructions in expr or the listed statements are then executed
    with each occurrence of param_i replaced by the value obtained
    for arg_i.

    In a call, arg_i may be preceded by a backquote (`)  to indicate that
    evaluation of arg_i is not to include a final evaluation of an lvalue.
    For example, suppose a function f and a global variable A have been
    defined by:

      ; define f(x) = (x = 3);
      ; global mat A[3];

    If g() is  a function that evaluates to 2:

      ; f(A[g()]);

    assigns the value of A[2] to the parameter x and then assigns the
    value 3 to x:

      ; f(`A[g()]);

    has essentially the effect of assigning A[2] as an lvalue to x and
    then assigning the value 3 to A[2].  (Very old versions of calc
    achieved the same result by using '&' as in  f(&A[g()]).)

    The number of arguments arg_1, arg_2, ... in a call need not equal the
    number of parameters.  If there are fewer arguments than parameters,
    the "missing" values are assigned the null value.

    In the definition of a function, the builtin function param(n)
    provides a way of referring to the parameters.  If n (which may
    result from evaluating an expreession) is zero, it returns the number
    of arguments in a call to the function, and if 1 <= n <= param(0),
    param(n) refers to the parameter with index n.
```

If no error occurs and no quit statement or abort statement is encountered during evaluation of the expression or the statements, the function call returns a value. In the expression form, this is simply the value of the expression.

In the statement form, if a return statement is encountered, the "return" keyword is to be either immediately followed by an expression or by a statement terminator (semicolon or rightbrace); in the former case, the expression is evaluated, evaluation of the function ceases, and the value obtained for the expression is returned as the "value of the function"; in the no-expression case, evaluation ceases immediately and the null-value is returned.

In the expression form of definition, the end of the expression expr is to be indicated by either a semicolon or a newline not within a part enclosed by parentheses; the definition may extend over several physical lines by ending each line with a '\' character or by enclosing the expression in parentheses. In interactive mode, that a definition has not been completed is indicated by the continuation prompt. A ctrl-C interrupt at this stage will abort the definition.

If the expr is omitted from an expression definition, as in:

```
  ; define h() = ;
```

any call to the function will evaluate the arguments and return the null value.

In the statement form, the definition ends when a matching right brace completes the "block" started by the initial left brace. Newlines within the block are treated as white space; statements within the block end with a ';' or a '}' matching an earlier '{'.

If a function with name fname had been defined earlier, the old definition has no effect on the new definition, but if the definition is completed successfully, the new definition replaces the old one; otherwise the old definition is retained. The number of parameters and their names in the new definiton may be quite different from those in the old definition.

An attempt at a definition may fail because of scanerrors as the definition is compiled. Common causes of these are: bad syntax, using identifiers as names of variables not yet defined. It is not a fault to have in the definition a call to a function that has not yet been defined; it is sufficient that the function has been defined when a call is made to the function.

After fname has been defined, the definition may be removed by the command:

```
  ; undefine fname
```

The definitions of all user-defined functions may be removed by:

```
  ; undefine *
```

If bit 0 of config("resource_debug") is set and the define command is at interactive level, a message saying that fname has been defined or redefined is displayed. The same message is displayed if bit 1 of config("resource_debug") is set and the define command is read from a file.

The identifiers used for the parameters in a function definition do not form part of the completed definition. For example,

```
; define f(a,b) = a + b;
; define g(alpha, beta) = alpha + beta;
```

result in identical code for the functions f, g.

If config("trace") & 8 is nonzero, the opcodes of a newly defined function are displayed on completion of its definition, parameters being specified by names used in the definition. For example:

```
; config("trace", 8),
; define f(a,b) = a + b
0: PARAMADDR a
2: PARAMADDR b
4: ADD
5: RETURN
f(a,b) defined
```

The opcodes may also be displayed later using the show opcodes command; parameters will be specified by indices instead of by names. For example:

```
; show opco f
0: PARAMADDR 0
2: PARAMADDR 1
4: ADD
5: RETURN
```

When a function is defined by the statement mode, the opcodes normally include DEBUG opcodes which specify statement boundaries at which SIGINT interruptions are likely to be least risky. Inclusion of the DEBUG opcodes is disabled if config("trace") & 2 is nonzero. For details, see help interrupt.

While config("trace") & 1 is nonzero, the opcodes are displayed as they are being evaluated. The current function is identified by its name, or "*" in the case of a command-line and "**" in the case of an eval(str) evaluation.

When a function is called, argument values may be of any type for which the operations and any functions used within the body of the definition can be executed. For example, whatever the intention at the time they were defined, the functions f1(), f2() defined above may be called with integer, fractional, or complex-number values, or with both arguments strings, or under some compatibility conditions, matrices or objects.

EXAMPLE

```
; define f(a,b) = 2*a + b;
; define g(alpha, beta)
;; {
;;          local a, pi2;
;;
;;          pi2 = 2 * pi();
;;          a = sin(alpha % pi2);
;;          if (a > 0.0) {
```

```
    ;;              return a*beta;
    ;;          }
    ;;          if (beta > 0.0) {
    ;;              a *= cos(-beta % pi2);
    ;;          }
    ;;          return a;
    ;; }
```

LIMITS
    The number of arguments in a function-call cannot exceed 1024.

LIBRARY
    none

SEE ALSO
    param, variable, undefine, show

param - value of argument in a user-function call

```
SYNOPSIS
    param([n])

TYPES
    n       nonnegative integer

    return  any

DESCRIPTION
    The function param(n) can be used only within the body of the
    definition of a function.

    If that function is f()

      [[ which may have been defined with named arguments as in f(x,y,z))

    and either the number of arguments or the value of an argument
    in an anticipated call to f() is to be used, the number of
    arguments in that call will then be returned by:

      param(0)

    and the value of the n-th argument by:

      param(n)

    Note that unlike the argv() builtin, param(1) is the 1st parameter
    and param(param(0) is the last.

EXAMPLE
    ; define f() {
    ;;      local n, v = 0;
    ;;      for (n = 1; n <= param(0); n++)
    ;;            v += param(n)^2;
    ;;      return v;
    ;; }

    ; print f(), f(1), f(1,2), f(1,2,3)
    0 1 5 14

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    argv, command
```

Statements

```
Statements are very much like C statements.  Most statements act
identically to those in C, but there are minor differences and
some additions.  The following is a list of the statement types,
with explanation of the non-C statements.

Statements are generally terminated with semicolons or { ... }.

C-like statements
-----------------
{ statement }
{ statement; ... statement }


C-like flow control
-------------------
if (expr) statement
if (expr) statement else statement
for (optionalexpr ; optionalexpr ; optionalexpr) statement
while (expr) statement
do statement while (expr)

      These all work like in normal C.

      IMPORTANT NOTE: When statement is of the form { ... },
      the leading { must be on the same line as the if, for,
      while or do keyword.

      This works as expected:

        if (expr) {
            ...
        }

      However this WILL NOT WORK AS EXPECTED:

        if (expr)
        {
            ...
        }

      because calc will parse the if being terminated by
      an empty statement followed by a

        if (expr) ;
        {
            ...
        }

      In the same way, use these forms:

        for (optionalexpr ; optionalexpr ; optionalexpr) {
              ...
        }

        while (expr) {
```

```
                ...
        }

        do {
                ...
        while (expr);

      where the initial { is on the SAME LINE as the if, while,
      for or do.

      See 'help expression' for details on expressions.
      See 'help builtin' for details on calc builtin functions.
      See 'help unexpanded' for things C programmers do not expect.
      See also 'help todo' and 'help bugs'.


C-like flow breaks
------------------
continue
break
goto label
      These all work like in normal C.

      See 'help expression' for details on expressions.
      See 'help builtin' for details on calc builtin functions.


return
------
return
return expr
return ( expr )
      This returns a value from a function.  Functions always
      have a return value, even if this statement is not used.
      If no return statement is executed, or if no expression
      is specified in the return statement, then the return
      value from the function is the null type.


switch
------
switch (expr) { caseclauses }
      Switch statements work similarly to C, except for the
      following.    A switch can be done on any type of value,
      and the case statements can be of any type of values.
      The case statements can also be expressions calculated
      at runtime.    The calculator compares the switch value
      with each case statement in the order specified, and
      selects the first case which matches.  The default case
      is the exception, and only matches once all other cases
      have been tested.


matrix
------
mat variable [dimension] [dimension] ...
mat variable [dimension, dimension, ...]
mat variable [] = { value, ... }
      This creates a matrix variable with the specified dimensions.
```

Matrices can have from 1 to 4 dimensions. When specifying multiple dimensions, you can use either the standard C syntax, or else you can use commas for separating the dimensions. For example, the following two statements are equivalent, and so will create the same two dimensional matrix:

```
mat foo[3][6];
mat foo[3,6];
```

By default, each dimension is indexed starting at zero, as in normal C, and contains the specified number of elements. However, this can be changed if a colon is used to separate two values. If this is done, then the two values become the lower and upper bounds for indexing. This is convenient, for example, to create matrices whose first row and column begin at 1. Examples of matrix definitions are:

```
mat x[3]      one dimension, bounds are 0-2
mat foo[4][5] two dimensions, bounds are 0-3 and 0-4
mat a[-7:7] one dimension, bounds are (-7)-7
mat s[1:9,1:9]      two dimensions, bounds are 1-9 and 1-9
```

Note that the MAT statement is not a declaration, but is executed at runtime. Within a function, the specified variable must already be defined, and is just converted to a matrix of the specified size, and all elements are set to the value of zero. For convenience, at the top level command level, the MAT command automatically defines a global variable of the specified name if necessary.

Since the MAT statement is executed, the bounds on the matrix can be full expressions, and so matrices can be dynamically allocated. For example:

```
size = 20;
mat data[size*2];
```

allocates a matrix which can be indexed from 0 to 39.

Initial values for the elements of a matrix can be specified by following the bounds information with an equals sign and then a list of values enclosed in a pair of braces. Even if the matrix has more than one dimension, the elements must be specified as a linear list. If too few values are specified, the remaining values are set to zero. If too many values are specified, a runtime error will result. Examples of some initializations are:

```
mat table1[5] = {77, 44, 22};
mat table2[2,2] = {1, 2, 3, 4};
```

When an initialization is done, the bounds of the matrix can optionally be left out of the square brackets, and the correct bounds (zero based) will be set. This can only be done for one-dimensional matrices. An example of this is:

```
mat fred[] = {99, 98, 97};
```

```
       The MAT statement can also be used in declarations to set
       variables as being matrices from the beginning.  For example:

             local mat temp[5];
             static mat strtable[] = {"hi", "there", "folks");


object
------
obj type { elementnames } optionalvariables
obj type variable
       These create a new object type, or create one or more
       variables of the specified type.  For this calculator,
       an object is just a structure which is implicitly acted
       on by user defined routines.  The user defined routines
       implement common operations for the object, such as plus
       and minus, multiply and divide, comparison and printing.
       The calculator will automatically call these routines in
       order to perform many operations.

       To create an object type, the data elements used in
       implementing the object are specified within a pair
       of braces, separated with commas.  For example, to
       define an object will will represent points in 3-space,
       whose elements are the three coordinate values, the
       following could be used:

             obj point {x, y, z};

       This defines an object type called point, whose elements
       have the names x, y, and z.      The elements are accessed
       similarly to structure element accesses, by using a period.
       For example, given a variable 'v' which is a point object,
       the three coordinates of the point can be referenced by:

             v.x
             v.y
             v.z

       A particular object type can only be defined once, and
       is global throughout all functions.    However, different
       object types can be used at the same time.

       In order to create variables of an object type, they
       can either be named after the right brace of the object
       creation statement, or else can be defined later with
       another obj statement.  To create two points using the
       second (and most common) method, the following is used:

             obj point p1, p2;

       This statement is executed, and is not a declaration.
       Thus within a function, the variables p1 and p2 must have
       been previously defined, and are just changed to be the
       new object type.  For convenience, at the top level command
       level, object variables are automatically defined as being
       global when necessary.

       Initial values for an object can be specified by following
```

```
       the variable name by an equals sign and a list of values
       enclosed in a pair of braces.  For example:

             obj point pt = {5, 6};

       The OBJ statement can also be used in declarations to set
       variables as being objects from the beginning.  If multiple
       variables are specified, then each one is defined as the
       specified object type.  Examples of declarations are:

             local obj point temp1;
             static obj point temp2 = {4, 3};
             global obj point p1, p2, p3;


print expressions
-----------------
print expr
print expr, ... expr
print expr: ... expr
       For interactive expression evaluation, the values of all
       typed-in expressions are automatically displayed to the
       user.  However, within a function or loop, the printing of
       results must be done explicitly.  This can be done using
       the 'printf' or 'fprintf' functions, as in standard C, or
       else by using the built-in 'print' statement.  The advantage
       of the print statement is that a format string is not needed.
       Instead, the given values are simply printed with zero or one
       spaces between each value.

       Print accepts a list of expressions, separated either by
       commas or colons.  Each expression is evaluated in order
       and printed, with no other output, except for the following
       special cases.  The comma which separates expressions prints
       a single space, and a newline is printed after the last
       expression unless the statement ends with a colon.      As
       examples:

             print 3, 4;               prints "3 4" and newline.
             print 5:;                 prints "5" with no newline.
             print 'a' : 'b' , 'c';    prints "ab c" and newline.
             print;              prints a newline.

       For numeric values, the format of the number depends on the
       current "mode" configuration parameter.  The initial mode
       is to print real numbers, but it can be changed to other
       modes such as exponential, decimal fractions, or hex.

       If a matrix or list is printed, then the elements contained
       within the matrix or list will also be printed, up to the
       maximum number specified by the "maxprint" configuration
       parameter.    If an element is also a matrix or a list, then
       their values are not recursively printed.  Objects are printed
       using their user-defined routine.  Printing a file value
       prints the name of the file that was opened.


Also see the help topic:
```

```
      help command  top level commands
      help expression     calc expression syntax
      help builtin  calc builtin functions
      help usage          how to invoke the calc command and calc -options

You may obtain help on individual builtin functions.  For example:

      help asinh
      help round

See:
      help builtin

for a list of builtin functions.

Some calc operators have their own help pages:

      help ->
      help *
      help .
      help %
      help //
      help #

See also:

      help help
```

Expression sequences

This is a sequence of statements, of which expression statements are the commonest case. Statements are separated with semicolons, and the newline character generally ends the sequence. If any statement is an expression by itself, or is associated with an 'if' statement which is true, then two special things can happen. If the sequence is executed at the top level of the calculator, then the value of '.' is set to the value of the last expression. Also, if an expression is a non-assignment, then the value of the expression is automatically printed if its value is not NULL. Some operations such as pre-increment and plus-equals are also treated as assignments.

Examples of this are the following:

```
expression              sets '.' to         prints
----------              -----------         ------
3+4                       7              7
2*4; 8+1; fact(3)         6           8, 9, and 6
x=3^2               9              -
if (3 < 2) 5; else 6      6              6
x++                       old x          -
print fact(4)       -              24
null()              null()               -
```

Variables can be defined at the beginning of an expression sequence. This is most useful for local variables, as in the following example, which sums the square roots of the first few numbers:

```
local s, i; s = 0; for (i = 0; i < 10; i++) s += sqrt(i); s
```

If a return statement is executed in an expression sequence, then the result of the expression sequence is the returned value. In this case, '.' is set to the value, but nothing is printed.

Operators

The operators are similar to C, but there are some differences in the associativity and precedence rules for some operators. In addition, there are several operators not in C, and some C operators are missing. A more detailed discussion of situations that may be unexpected for the C programmer may be found in the 'unexpected' help file.

Below is a list giving the operators arranged in order of precedence, from the least tightly binding to the most tightly binding. Except where otherwise indicated, operators at the same level of precedence associate from left to right.

Unlike C, calc has a definite order for evaluation of terms (addends in a sum, factors in a product, arguments for a function or a matrix, etc.). This order is always from left to right. but skipping of terms may occur for ||, && and ? : . For example, an expression of the form:

```
A * B + C * D
```

is evaluated in the following order:

```
A
B
A * B
C
D
C * D
A * B + C * D
```

This order of evaluation is significant if evaluation of a term changes a variable on which a later term depends. For example:

```
x++ * x++ + x++ * x++
```

returns the value of:

```
x * (x + 1) + (x + 2) * (x + 3)
```

and increments x as if by x += 4. Similarly, for functions f, g, the expression:

```
f(x++, x++) + g(x++)
```

evaluates to:

```
f(x, x + 1) + g(x + 2)
```

and increments x three times.

In A || B, B is read only if A tests as false; in A && B, B is read only if A tests as true. Thus if x is nonzero, x++ || x++ returns x and increments x once; if x is zero, it returns x + 1 and increments x twice.

```
, Comma operator.
      a, b returns the value of b.
      For situations in which a comma is used for another purpose
      (function arguments, array indexing, and the print statement),
      parenthesis must be used around the comma operator expression.
      E.g., if A is a matrix, A[(a, b), c] evaluates a, b, and c, and
      returns the value of A[b, c].

+=      -=  *=      /=  %=      //=  &=      |=  <<=  >>=  ^=  **=
   Operator-with-assignments.
      These associate from left to right, e.g. a += b *= c has the
      effect of a = (a + b) * c, where only a is required to be an
      lvalue.  For the effect of b *= c; a += b; when both a and b
      are lvalues, use a += (b *= c).

= Assignment.
      As in C, this, when repeated, this associates from right to left,
      e.g. a = b = c has the effect of a = (b = c).  Here both a and b
      are to be lvalues.

? : Conditional value.
      a ? b : c  returns b if a tests as true (i.e. nonzero if
      a is a number), c otherwise.  Thus it is equivalent to:
      if (a) return b; else return c;.
      All that is required of the arguments in this function
      is that the "is-it-true?" test is meaningful for a.
      As in C, this operator associates from right to left,
      i.e. a ? b : c ? d : e is evaluated as a ? b : (c ? d : e).

||      Logical OR.
      Unlike C, the result for a || b is one of the operands
      a, b rather than one of the numbers 0 and 1.
      a || b is equivalent to a ? a : b, i.e. if a tests as
      true, a is returned, otherwise b.  The effect in a
      test like "if (a || b) ... " is the same as in C.

&&      Logical AND.
      Unlike C, the result for a && b is one of the operands
      a, b rather than one of the numbers 0 and 1.
      a && b is equivalent to a ? b : a, i.e. if a tests as
      true, b is returned, otherwise a.  The effect in a
      test like "if (a && b) ... " is the same as in C.

==      !=  <=      >=  <  >
      Relations.

+  -
      Binary plus and minus and unary plus and minus when applied to
      a first or only term.

*  /  //  %
      Multiply, divide, and modulo.
      Please Note: The '/' operator is a fractional divide,
      whereas the '//' is an integral divide.  Thus think of '/'
      as division of real numbers, and think of '//' as division
      of integers (e.g., 8 / 3 is 8/3 whereas 8 // 3 is 2).
      The '%' is integral or fractional modulus (e.g., 11%4 is 3,
      and 10%pi() is ~.575222).
```

```
| Bitwise OR.
      In a | b, both a and b are to be real integers;
      the signs of a and b are ignored, i.e.
      a | b = abs(a) | abs(b) and the result will
      be a non-negative integer.

& Bitwise AND.
      In a & b, both a and b are to be real integers;
      the signs of a and b are ignored as for a | b.

^  **  <<  >>
      Powers and shifts.
      The '^' and '**' are both exponentiation, e.g. 2^3
      returns 8, 2^-3 returns .125.  Note that in a^b, if
      'a' == 0 and 'b' is real, then is must be >= 0 as well.
      Also 0^0 and 0**0 return the value 1.

      For the shift operators both arguments are to be
      integers, or if the first is complex, it is to have
      integral real and imaginary parts.    Changing the
      sign of the second argument reverses the shift, e.g.
      a >> -b = a << b.  The result has the same sign as
      the first argument except that a nonzero value is
      reduced to zero by a sufficiently long shift to the
      right.  These operators associate right to left,
      e.g.  a << b ^ c = a << (b ^ c).

+  -  !
      Plus (+) and minus (-) have their usual meanings as unary
      prefix operators at this level of precedence when applied to
      other than a first or only term.

      As a prefix operator, '!' is the logical NOT: !a returns 0 if
      a tests as nonzero, and 1 if a tests as zero, i.e. it is
      equivalent to a ? 0 : 1.  Be careful about
      using this as the first character of a top level command,
      since it is also used for executing shell commands.

      As a postfix operator ! gives the factorial function, i.e.
      a! = fact(a).

++      --
      Pre or post incrementing or decrementing.
      These are applicable only to variables.

[ ]      [[ ]]      .  ( )
      Indexing, double-bracket indexing, element references,
      and function calls.  Indexing can only be applied to matrices,
      element references can only be applied to objects, but
      double-bracket indexing can be applied to matrices, objects,
      or lists.

variables  constants  .  ( )
      These are variable names and constants, the special '.' symbol,
      or a parenthesized expression.  Variable names begin with a
      letter, but then can contain letters, digits, or underscores.
      Constants are numbers in various formats, or strings inside
      either single or double quote marks.
```

```
The most significant difference from the order of precedence in
C is that | and & have higher precedence than ==, +, -, *, / and %.
For example, in C a == b | c * d is interpreted as:

      (a == b) | (c * d)

and calc it is:

      a == ((b | c) * d)


Most of the operators will accept any real or complex numbers
as arguments.  The exceptions are:

/  //  %
      Second argument must be nonzero.

^
      The exponent must be an integer.  When raising zero
      to a power, the exponent must be non-negative.

|  &
      Both both arguments must be integers.

<<      >>
      The shift amount must be an integer.  The value being
      shifted must be an integer or a complex number with
      integral real and imaginary parts.


See the 'unexpected' help file for a list of unexpected
surprises in calc syntax/usage.  Persons familiar with C should
read the 'unexpected' help file to avoid confusion.
```

Variable declarations

```
Variables can be declared as either being global, local, or static.
Global variables are visible to all functions and on the command
line, and are permanent.  Local variables are visible only within
a single function or command sequence.  When the function or command
sequence returns, the local variables are deleted.      Static variables
are permanent like global variables, but are only visible within the
same input file or function where they are defined.

To declare one or more variables, the 'local', 'global', or 'static'
keywords are used, followed by the desired list of variable names,
separated by commas.  The definition is terminated with a semicolon.
Examples of declarations are:

      local   x, y, z;
      global  fred;
      local   foo, bar;
      static  var1, var2, var3;

Variables may have initializations applied to them.      This is done
by following the variable name by an equals sign and an expression.
Global and local variables are initialized each time that control
reaches them (e.g., at the entry to a function which contains them).
Static variables are initialized once only, at the time that control
first reaches them (but in future releases the time of initialization
may change).  Unlike in C, expressions for static variables may
contain function calls and refer to variables.  Examples of such
initializations are:

      local   a1 = 7, a2 = 3;
      static  b = a1 + sin(a2);

Within function declarations, all variables must be defined.
But on the top level command line, assignments automatically define
global variables as needed.      For example, on the top level command
line, the following defines the global variable x if it had not
already been defined:

      x = 7

The static keyword may be used at the top level command level to
define a variable which is only accessible interactively, or within
functions defined interactively.

Variables have no fixed type, thus there is no need or way to
specify the types of variables as they are defined.      Instead, the
types of variables change as they are assigned to or are specified
in special statements such as 'mat' and 'obj'.  When a variable is
first defined using 'local', 'global', or 'static', it has the
value of zero.

If a procedure defines a local or static variable name which matches
a global variable name, or has a parameter name which matches a
global variable name, then the local variable or parameter takes
precedence within that procedure, and the global variable is not
directly accessible.
```

The MAT and OBJ keywords may be used within a declaration statement in order to initially define variables as that type. Initialization of these variables are also allowed. Examples of such declarations are:

```
static mat table[3] = {5, 6, 7};
local obj point p1, p2;
```

When working with user-defined functions, the syntax for passing an lvalue by reference rather than by value is to precede an expression for the lvalue by a backquote. For example, if the function invert is defined by:

```
define invert(x) {x = inverse(x)}
```

then invert(`A) achieves the effect of A = inverse(A). In other words, passing and argument of `variable (with a back-quote) will cause and changes to the function argument to be applied to the calling variable. Calling invert(A) (without the ` backquote) assigns inverse(A) to the temporary function parameter x and leaves A unchanged.

In an argument, a backquote before other than an lvalue is ignored. Consider, for example:

```
; define logplus(x,y,z) {return log(++x + ++y + ++z);}

; eh = 55;
; mi = 25;
; answer = logplus(eh, `mi, `17);

; print eh, mi, answer;
55 26 2
```

The value of eh is was not changed because eh was used as an argument without a back-quote (`). However, mi was incremented because it was passed as `mi (with a back-quote). Passing 17 (not an lvalue) as `17 has not effect on the value 17.

The back-quote should only be used before arguments to a function. In all other contexts, a backquote causes a compile error.

Another method is to pass the address of the lvalue explicitly and use the indirection operator * (star) to refer to the lvalue in the function body. Consider the following function:

```
; define ten(a) { *a = 10; }

; n = 17;
; ten(n);
; print n;
17

; ten(`n);
; print n;
17

; ten(&n);
```

```
; print n;
10
```

Passing an argument with a & (ampersand) allows the tenmore() function to modify the calling variable:

```
; wa = tenmore(&vx);
; print vx, wa;
65 65
```

Great care should be taken when using a pointer to a local variable or element of a matrix, list or object, since the lvalue pointed to is deleted when evaluation of the function is completed or the lvalue whose value is the matrix, list or object is assigned another value.

As both of the above methods (using & arguments (ampersand) *value (star) function values or by using ` arguments (back quote) alone) copy the address rather than the value of the argument to the function parameter, they allow for faster calls of functions when the memory required for the value is huge (such as for a large matrix).

As the built-in functions and object functions always accept their arguments as addresses, there is no gain in using the backquote when calling these functions.

Unexpected

```
While calc is C-like, users of C will find some unexpected
surprises in calc syntax and usage.    Persons familiar with C should
review this file.

Persons familiar with shell scripting may want to review this file
as well, particularly notes dealing with command line evaluation
and execution.


The Comma
=========

The comma is also used for continuation of obj and mat creation
expressions and for separation of expressions to be used for
arguments or values in function calls or initialization lists.  The
precedence order of these different uses is:  continuation,
separator, comma operator.      For example, assuming the variables a,
b, c, d, e, and object type xx have been defined, the arguments
passed to f in:

      f(a, b, c, obj xx d, e)

are a, b, c, and e, with e having the value of a newly created xx
object.  In:

      f((a, b), c, (obj xx d), e)

the arguments of f are b, c, d, e, with only d being a newly
created xx object.

In combination with other operators, the continuation use of the
comma has the same precedence as [] and ., the separator use the
same as the comma operator.      For example, assuming xx.mul() has
been defined:

      f(a = b, obj xx c, d = {1,2} * obj xx e = {3,4})

passes two arguments: a (with value b) and the product d * e of two
initialized xx objects.


^ is not xor
** is exponentiation
====================

In C, ^ is the xor operator. The expression:

      a ^ b

yields "a to the b power", NOT "a xor b".

Unlike in C, calc evaluates the expression:

      a ** b
```

```
also yields "a to the b power".

Here "a" and "b" can be a real value or a complex value:

  2^3                 3i^4
  2.5 ^ 3.5           0.5i ^ 0.25
  2.5 ^ 2.718i              3.13145i ^ 0.30103i

In addition, "a" can be matrix.  In this case "b" must be an integer:

  mat a[2,2] = {1,2,3,4};
  a^3

Note that 'a' == 0 and 'b' is real, then is must be >= 0 as well.
Also 0^0 and 0**0 return the value 1.

Be careful about the precedence of operators.  Note that:

  -1 ^ 0.5 == -1

whereas:

  (-1) ^ 0.5 == 1i

because the above expression in parsed as:

  -(1 ^ 0.5) == -1

whereas:

  (-1) ^ 0.5 == 1i


op= operators associate left to right
=====================================

Operator-with-assignments:

      +=      -=  *=      /=  %=      //=  &=      |=  <<=  >>=  ^=  **=

associate from left to right instead of right to left as in C.
For example:

      a += b *= c

has the effect of:

      a = (a + b) * c

where only 'a' is required to be an lvalue.  For the effect of:

      b *= c; a += b

when both 'a' and 'b' are lvalues, use:

      a += (b *= c)


|| yields values other than 0 or 1
```

```
==================================

In C:

      a || b

will produce 0 or 1 depending on the logical evaluation
of the expression.  In calc, this expression will produce
either 'a' or 'b' and is equivalent to the expression:

      a ? a : b

In other words, if 'a' is true, then 'a' is returned, otherwise
'b' is returned.


&& yields values other than 0 or 1
==================================

In C:

      a && b

will produce 0 or 1 depending on the logical evaluation
of the expression.  In calc, this expression will produce
either 'a' or 'b' and is equivalent to the expression:

      a ? b : a

In other words, if 'a' is true, then 'b' is returned, otherwise
'a' is returned.


/ is fractional divide, // is integral divide
=============================================

In C:

      x/y

performs integer division when 'x' and 'y' are integer types.
In calc, this expression yields a rational number.

Calc uses:

      x//y

to perform division with integer truncation and is the equivalent to:

      int(x/y)


| and & have higher precedence than ==, +, -, *, / and %
========================================================

Is C:

      a == b | c * d
```

is interpreted as:

```
      (a == b) | (c * d)
```

and calc it is interpreted as:

```
      a == ((b | c) * d)
```

```
calc always evaluates terms from left to right
==============================================
```

Calc has a definite order for evaluation of terms (addends in a sum, factors in a product, arguments for a function or a matrix, etc.). This order is always from left to right. but skipping of terms may occur for ||, && and ? : .

Consider, for example:

```
      A * B + C * D
```

In calc above expression is evaluated in the following order:

```
      A
      B
      A * B
      C
      D
      C * D
      A * B + C * D
```

This order of evaluation is significant if evaluation of a term changes a variable on which a later term depends. For example:

```
      x++ * x++ + x++ * x++
```

in calc returns the value:

```
      x * (x + 1) + (x + 2) * (x + 3)
```

and increments x as if by x += 4. Similarly, for functions f, g, the expression:

```
      f(x++, x++) + g(x++)
```

evaluates to:

```
      f(x, x + 1) + g(x + 2)
```

and increments x three times.

```
&A[0] and A are different things in calc
========================================
```

In calc, value of &A[0] is the address of the first element, whereas A is the entire array.

```
*X may be used to to return the value of X
==========================================

If the current value of a variable X is an octet, number or string,
*X may be used to to return the value of X; in effect X is an
address and *X is the value at X.


freeing a variable has the effect of assigning the null value to it
===================================================================

The freeglobals(), freestatics(), freeredc() and free() free
builtins to not "undefine" the variables, but have the effect of
assigning the null value to them, and so frees the memory used for
elements of a list, matrix or object.

Along the same lines:

      undefine *

undefines all current user-defined functions.  After executing
all the above freeing functions (and if necessary free(.) to free
the current "old value"), the only remaining numbers as displayed by

      show numbers

should be those associated with epsilon(), and if it has been
called, qpi().


#! is also a comment
====================

In addition to the C style /* comment lines */, lines that begin with
#! are treated as comments.

A single # is an calc operator, not a comment.  However two or more
##'s in a row is a comment.  See "help pound" for more information.

  #!/usr/local/src/cmd/calc/calc -q -f

  /* a correct comment */
  ## another correct comment
  ### two or more together is also a comment
  /*
   * another correct comment
   */
  print "2+2 =", 2+2;     ## yet another comment

This next example is WRONG:

  #!/usr/local/src/cmd/calc/calc -q -f

  # This is not a calc calc comment because it has only a single #
  # You must to start comments with ## or /*
  print "This example has invalid comments"

See "help cscript" and "help usage" for more information.
```

```
The { must be on the same line as an if, for, while or do
=========================================================

When statement is of the form { ... }, the leading { MUST BE ON
THE SAME LINE as the if, for, while or do keyword.

This works as expected:

  if (expr) {
      ...
  }

However this WILL NOT WORK AS EXPECTED:

  if (expr)
  {
      ...
  }

because calc will parse the if being terminated by
an empty statement followed by a

  if (expr) ;
  {
      ...
  }

In the same way, use these forms:

  for (optionalexpr ; optionalexpr ; optionalexpr) {
        ...
  }

  while (expr) {
        ...
  }

  do {
        ...
  while (expr);

where the initial { is on the SAME LINE as the if, while,
for or do keyword.

NOTE: See "help statement", "help todo", and "help bugs".


Shell evaluation of command line arguments
==========================================

In most interactive shells:

  calc 2 * 3

will frequently produce a "Missing operator" error because the '*' is
evaluated as a "shell glob".  To avoid this you must quote or escape
argument with characters that your interactive shell interprets.
```

```
For example, bash / ksh / sh shell users should use:

  calc '2 * 3'

or:

  calc 2 \* 3

or some other form of shell meta-character escaping.


Calc reads standard input after processing command line args
============================================================

The shell command:

  seq 5 | while read i; do calc "($i+3)^2"; done

  FYI: The command "seq 5" will write 1 through 5 on separate
       lines on standard output, while read i sets $i to
       the value of each line that is read from stdin.

will produce:

    16
  2
  3
  4
  5

The reason why the last 4 lines of output are 2 through 5 is
that after calc evaluates the first line and prints (1+3)^2
(i.e., 16), calc continues to read stdin and slurps up all
of the remaining data on the pipe.

To avoid this problem, use:

  seq 5 | while read i; do calc "($i+3)^2" </dev/null; done

which produces the expected results:

  16
  25
  36
  49
  64
```

& - address operator

```
SYNOPSIS
    &X

TYPES
    X       expression specifying an octet, lvalue, string or number

    return  pointer

DESCRIPTION
    &X returns the address at which information for determining the current
    value of X is stored.  After an assignment as in p = &X, the
    value of X is accessible by *p so long as the connection between p
    and the value is not broken by relocation of the information or by the
    value ceasing to exist.  Use of an address after the connection
    is broken is unwise since the calculator may use that address for other
    purposes; the consequences of attempting to write data to, or
    otherwise accessing, such a vacated address may be catastrophic.

    An octet is normally expressed by B[i] where B is a block and
    0 <= i < sizeof(B).  &B[i] then returns the address at which this
    octet is located until the block is freed or relocated.  Freeing
    of an unnamed block B occurs when a new value is assigned to B or
    when B ceases to exist; a named block B is freed by blkfree(B).
    A block is relocated when an operation like copying to B requires
    a change of sizeof(B).

    An lvalue may be expressed by an identifier for a variable, or by
    such an identifier followed by one or more qualifiers compatible with
    the type of values associated with the variable and earlier qualifiers.
    If an identifier A specifies a global or static variable, the address
    &A is permanently associated with that variable.  For a local variable
    or function parameter A, the association of the variable with &A
    is limited to each occasion when the function is called.  If X specifies a
    component or element of a matrix or object, connection of
    &X with that component or element depends only on the continued existence
    of the matrix or object.  For example, after

      ; mat A[3]

    the addresses &A[0], &A[1], &A[2] locate the three elements
    of the matrix specified by A until another value is assigned to A, etc.
    Note one difference from C in that &A[0] is not the same as A.

    An element of a list has a fixed address while the list exists and
    the element is not removed by pop(), remove(), or delete(); the index
    of the element changes if an element is pushed onto the list, or if
    earlier elements are popped or deleted.

    Elements of an association have fixed addresses so long as the association
    exists.  If A[a,b,...] has not been defined for the association A,
    &A[a,b,...] returns the constant address of a particular null value.

    Some other special values have fixed addresses; e.g. the old value (.).

    Some arithmetic operations are defined for addresses but these should
```

be used only for octets or components of a matrix or object where the results refer to octets in the same block or existing components of the same matrix or object. For example, immediately after

```
; mat A[10]
; p = &A[5]
```

it is permitted to use expressions like p + 4, p - 5, p++ .

Strings defined literally have fixed addresses, e.g., after

```
; p = &"abc"
; A = "abc"
```

the address &*A of the value of A will be equal to p.

Except in cases like strcat(A, "") when *A identified with a literal string as above, definitions of string values using strcat() or substr() will copy the relevant strings to newly allocated addresses which will be useable only while the variables retain these defined values.
For example, after

```
; B = C = strcat("a", "bc");
```

&*B and &*C will be different. If p is defined by p = &*B, p should not be used after a new value is assigned to B, or B ceases to exist, etc.

When compilation of a function encounters for the first time a particular literal number or the result of simple arithmetic operations (like +, -, *, or /) on literal numbers, that number is assigned to a particular address which is then used for later similar occurrences of that number so long as the number remains associated with at least one function or lvalue. For example, after

```
; x = 27;
; y = 3 * 9;
; define f(a) = 27 + a;
```

the three occurrences of 27 have the same address which may be displayed by any of &27, &*x, &*y and &f(0). If x and y are assigned other values and f is redefined or undefined and the 27 has not been stored elsewhere (e.g. as the "old value" or in another function definition or as an element in an association), the address assigned at the first occurrence of 27 will be freed and calc may later use it for another number.

When a function returns a number value, that number value is usually placed at a newly allocated address, even if an equal number is stored elsewhere. For example calls to f(a), as defined above, with the same non-zero value for a will be assigned to different addresses as can be seen from printing &*A, &*B, &*C after

```
; A = f(2); B = f(2); C = f(2);
```

(the case of f(0) is exceptional since 27 + 0 simply copies the 27 rather than creating a new number value). Here it is clearly more efficient to use

```
    ; A = B = C = f(2);
```

which, not only performs the addition in f() only once, but stores the number values for A, B and C at the same address.

Whether a value V is a pointer and if so, its type, is indicated by the value returned by isptr(V): 1, 2, 3, 4 for octet-, value-, string- and number-pointer respectively, and 0 otherwise.

The output when addresses are printed consists of a description (o_ptr, v_ptr, s_ptr, n_ptr) followed by : and the address printed in %p format.

Iteration of & is not permitted; &&X causes a "non-variable operand" scan error.

EXAMPLE

Addresses for particular systems may differ from those displayed here.

```
; mat A[3]
; B = blk()

; print &A, &A[0], &A[1]
v-ptr: 1400470d0 v-ptr: 140044b70 v-ptr: 140044b80

; print &B, &B[0], &B[1]
v-ptr: 140047130 o-ptr: 140044d00 o-ptr: 140044d01

; a = A[0] = 27
; print &*a, &*A[0]. &27
n_ptr: 14003a850 n_ptr: 14003a850 n_ptr: 14003a850

; a = A[0] = "abc"
; print &*a, &*A[0], &"abc"
s_ptr: 14004cae0 s_ptr: 14004cae0 s_ptr: 14004cae0
```

LIMITS

none

LINK LIBRARY

none

SEE ALSO

dereference, isptr

* - dereference or indirection operator

```
SYNOPSIS
    * X

TYPES
    X       address or lvalue

    return  any

DESCRIPTION
    When used as a binary operator, '*' performs multiplication.  When
    used as a operator, '*' returns the value at a given address.

    If X is an address, *X returns the value at that address.  This value
    will be an octet, lvalue, string, or number, depending on the
    type of address.  Thus, for any addressable A, *&A is the same as A.

    If X is an lvalue, *X returns the current value at the address
    considered to be specified by X.  This value may be an lvalue or
    octet, in which cases, for most operations except when X is the
    destination of an assignment, *X will contribute the same as X to
    the result of the operation. For example, if A and B are lvalues
    whose current values are numbers, A + B, *A + B, A + *B and *A + *B
    will all return the same result.  However if C is an lvalue and A is
    the result of the assignment A = &C, then A = B will assign the value
    of B to A, *A = B will assign the value of B to C without affecting
    the value of A.

    If X is an lvalue whose current value is a structure (matrix, object,
    list, or association), the value returned by *X is a copy of the
    structure rather than the structure identified by X.  For example,
    suppose B has been created by

      ; mat B[3] = {1,2,3}

    then

      ; A = *B = {4,5,6}

    will assign the values 4,5,6 to the elements of a copy of B, which
    will then become the value of A, so that the values of A and B will
    be different.  On the other hand,

      ; A = B = {4,5,6}

    will result in A and B having the same value.

    If X is an octet, *X returns the value of that octet as a number.

    The * operator may be iterated with suitable sequences of pointer-valued
    lvalues.  For example, after

      ; global a, b, c;
      ; b = &a;
      ; c = &b;
```

```
    **c returns the lvalue a;  ***c returns the value of a.

EXAMPLE
    ; mat A[3] = {1,2,3}
    ; p = &A[0]
    ; print *p, *(p + 1), *(p + 2)
    1 2 3

    ; *(p + 1) = 4
    ; print A[1]
    4

    ; A[0] = &a
    ; a = 7
    ; print **p
    7

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    address, isptr
```

Builtin types

```
The calculator has the following built-in types.

null value
      This is the undefined value type.  The function 'null'
      returns this value.  Functions which do not explicitly
      return a value return this type.  If a function is called
      with fewer parameters than it is defined for, then the
      missing parameters have the null type.  The null value is
      false if used in an IF test.

rational numbers
      This is the basic data type of the calculator.
      These are fractions whose numerators and denominators
      can be arbitrarily large.  The fractions are always
      in lowest terms.  Integers have a denominator of 1.
      The numerator of the number contains the sign, so that
      the denominator is always positive.    When a number is
      entered in floating point or exponential notation, it is
      immediately converted to the appropriate fractional value.
      Printing a value as a floating point or exponential value
      involves a conversion from the fractional representation.

      Numbers are stored in binary format, so that in general,
      bit tests and shifts are quicker than multiplies and divides.
      Similarly, entering or displaying of numbers in binary,
      octal, or hex formats is quicker than in decimal.  The
      sign of a number does not affect the bit representation
      of a number.

complex numbers
      Complex numbers are composed of real and imaginary parts,
      which are both fractions as defined above.  An integer which
      is followed by an 'i' character is a pure imaginary number.
      Complex numbers such as "2+3i" when typed in, are processed
      as the sum of a real and pure imaginary number, resulting
      in the desired complex number.  Therefore, parenthesis are
      sometimes necessary to avoid confusion, as in the two values:

            1+2i ^2       (which is -3)
            (1+2i) ^2     (which is -3+4i)

      Similar care is required when entering fractional complex
      numbers.  Note the differences below:

            3/4i          (which is -(3/4)i)
            3i/4          (which is (3/4)i)

      The imaginary unit itself is input using "1i".

strings
      Strings are a sequence of zero or more characters.
      They are input using either of the single or double
      quote characters.  The quote mark which starts the
      string also ends it.  Various special characters can
      also be inserted using back-slash.    Example strings:
```

```
        "hello\n"
        "that's all"
        'lots of """"'
        'a'
        ""

    There is no distinction between single character and
    multi-character strings.  The 'str' and 'ord' functions
    will convert between a single character string and its
    numeric value.  The 'str' and 'eval' functions will
    convert between longer strings and the corresponding
    numeric value (if legal).  The 'strcat', 'strlen', and
    'substr' functions are also useful.

matrices
    These are one to four dimensional matrices, whose minimum
    and maximum bounds can be specified at runtime.  Unlike C,
    the minimum bounds of a matrix do not have to start at 0.
    The elements of a matrix can be of any type.  There are
    several built-in functions for matrices.  Matrices are
    created using the 'mat' statement.

associations
    These are one to four dimensional matrices which can be
    indexed by arbitrary values, instead of just integers.
    These are also known as associative arrays.  The elements of
    an association can be of any type.    Very few operations are
    permitted on an association except for indexing.  Associations
    are created using the 'assoc' function.

lists
    These are a sequence of values, which are linked together
    so that elements can be easily be inserted or removed
    anywhere in the list.  The values can be of any type.
    Lists are created using the 'list' function.
```

Using objects

Objects are user-defined types which are associated with user-defined functions to manipulate them. Object types are defined similarly to structures in C, and consist of one or more elements. The advantage of an object is that the user-defined routines are automatically called by the calculator for various operations, such as addition, multiplication, and printing. Thus they can be manipulated by the user as if they were just another kind of number.

An example object type is "surd", which represents numbers of the form

```
a + b*sqrt(D),
```

where D is a fixed integer, and 'a' and 'b' are arbitrary rational numbers. Addition, subtraction, multiplication, and division can be performed on such numbers, and the result can be put unambiguously into the same form. (Complex numbers are an example of surds, where D is -1.)

The "obj" statement defines either an object type or an actual variable of that type. When defining the object type, the names of its elements are specified inside of a pair of braces. To define the surd object type, the following could be used:

```
obj surd {a, b};
```

Here a and b are the element names for the two components of the surd object. An object type can be defined more than once as long as the number of elements and their names are the same.

When an object is created, the elements are all defined with zero values. A user-defined routine should be provided which will place useful values in the elements. For example, for an object of type 'surd', a function called 'surd' can be defined to set the two components as follows:

```
define surd(a, b)
{
      local x;

      obj surd x;
      x.a = a;
      x.b = b;
      return x;
}
```

When an operation is attempted for an object, user functions with particular names are automatically called to perform the operation. These names are created by concatenating the object type name and the operation name together with an underscore. For example, when multiplying two objects of type surd, the function "surd_mul" is called.

The user function is called with the necessary arguments for that operation. For example, for "surd_mul", there are two arguments, which are the two numbers. The order of the arguments is always the order of the binary operands. If only one of the operands to a binary operator is an object, then the user function for that object type is still called. If the two operands are of different object types, then the user function that is called is the one for the first operand.

The above rules mean that for full generality, user functions should detect that one of their arguments is not of its own object type by using the 'istype' function, and then handle these cases specially. In this way, users can mix normal numbers with object types. (Functions which only have one operand don't have to worry about this.) The following example of "surd_mul" demonstrates how to handle regular numbers when used together with surds:

```
define surd_mul(a, b)
{
      local x;

      obj surd x;
      if (!istype(a, x)) {
            /* a not of type surd */
            x.a = b.a * a;
            x.b = b.b * a;
      } else if (!istype(b, x)) {
            /* b not of type surd */
            x.a = a.a * b;
            x.b = a.b * b;
      } else {
            /* both are surds */
            x.a = a.a * b.a + D * a.b * b.b;
            x.b = a.a * b.b + a.b * b.a;
      }
      if (x.b == 0)
            return x.a; /* normal number */
      return x;           /* return surd */
}
```

In order to print the value of an object nicely, a user defined routine can be provided. For small amounts of output, the print routine should not print a newline. Also, it is most convenient if the printed object looks like the call to the creation routine. For output to be correctly collected within nested output calls, output should only go to stdout. This means use the 'print' statement, the 'printf' function, or the 'fprintf' function with 'files(1)' as the output file. For example, for the "surd" object:

```
define surd_print(a)
{
      print "surd(" : a.a : "," : a.b : ")" : ;
}
```

It is not necessary to provide routines for all possible operations for an object, if those operations can be defaulted or do not make sense for the object. The calculator will attempt meaningful defaults for many operations if they are not defined. For example, if 'surd_square' is not defined to square a number, then 'surd_mul' will be called to perform the squaring. When a default is not possible, then an error will be generated.

```
Please note: Arguments to object functions are always passed by
reference (as if an '&' was specified for each variable in the call).
Therefore, the function should not modify the parameters, but should
copy them into local variables before modifying them.  This is done
in order to make object calls quicker in general.

The double-bracket operator can be used to reference the elements
of any object in a generic manner.    When this is done, index 0
corresponds to the first element name, index 1 to the second name,
and so on.    The 'size' function will return the number of elements
in an object.

The following is a list of the operations possible for objects.
The 'xx' in each function name is replaced with the actual object
type name.    This table is displayed by the 'show objfuncs' command.

      Name    Args  Comments

      xx_print      1     print value, default prints elements
      xx_one  1     multiplicative identity, default is 1
      xx_test 1     logical test (false,true => 0,1),
                    default tests elements
      xx_add  2
      xx_sub  2     subtraction, default adds negative
      xx_neg  1     negative
      xx_mul  2
      xx_div  2     non-integral division, default multiplies
                    by inverse
      xx_inv  1     multiplicative inverse
      xx_abs  2     absolute value within given error
      xx_norm 1     square of absolute value
      xx_conj 1     conjugate
      xx_pow  2     integer power, default does multiply,
                    square, inverse
      xx_sgn  1     sign of value (-1, 0, 1)
      xx_cmp  2     equality (equal,non-equal => 0,1),
                    default tests elements
      xx_rel  2     inequality (less,equal,greater => -1,0,1)
      xx_quo  2     integer quotient
      xx_mod  2     remainder of division
      xx_int  1     integer part
      xx_frac 1     fractional part
      xx_inc  1     increment, default adds 1
      xx_dec  1     decrement, default subtracts 1
      xx_square     1     default multiplies by itself
      xx_scale      2     multiply by power of 2
      xx_shift      2     shift left by n bits (right if negative)
      xx_round      2     round to given number of decimal places
      xx_bround     2     round to given number of binary places
      xx_root 3     root of value within given error
      xx_sqrt 2     square root within given error
      xx_or   2     boolean or
      xx_and  2     boolean and
      xx_not  1     boolean not
      xx_fact 1     factorial


Also see the standard resource files:
```

```
dms.cal
mod.cal
poly.cal
quat.cal
surd.cal
```

Builtin functions

```
There is a large number of built-in functions.  Many of the
functions work on several types of arguments, whereas some only
work for the correct types (e.g., numbers or strings).      In the
following description, this is indicated by whether or not the
description refers to values or numbers.  This display is generated
by the 'show builtin' command.


Name  Args  Description

abs       1-2   absolute value within accuracy b
acos      1-2   arccosine of a within accuracy b
acosh     1-2   inverse hyperbolic cosine of a within accuracy b
acot      1-2   arccotangent of a within accuracy b
acoth     1-2   inverse hyperbolic cotangent of a within accuracy b
acsc      1-2   arccosecant of a within accuracy b
acsch     1-2   inverse csch of a within accuracy b
agd       1-2   inverse gudermannian function
append    1+    append values to end of list
appr      1-3   approximate a by multiple of b using rounding c
arg       1-2   argument (the angle) of complex number
asec      1-2   arcsecant of a within accuracy b
asech     1-2   inverse hyperbolic secant of a within accuracy b
asin      1-2   arcsine of a within accuracy b
asinh     1-2   inverse hyperbolic sine of a within accuracy b
assoc     0     create new association array
atan      1-2   arctangent of a within accuracy b
atan2     2-3   angle to point (b,a) within accuracy c
atanh     1-2   inverse hyperbolic tangent of a within accuracy b
avg       0+    arithmetic mean of values
base      0-1   set default output base
base2     0-1   set default secondary output base
bernoulli 1     Bernoulli number for index a
bit       2     whether bit b in value a is set
blk       0-3   block with or without name, octet number, chunksize
blkcpy    2-5   copy value to/from a block: blkcpy(d,s,len,di,si)
blkfree   1     free all storage from a named block
blocks    0-1   named block with specified index, or null value
bround    1-3   round value a to b number of binary places
btrunc    1-2   truncate a to b number of binary places
calclevel 0     current calculation level
calcpath  0     current CALCPATH search path value
catalan   1     catalan number for index a
ceil      1     smallest integer greater than or equal to number
cfappr    1-3   approximate a within accuracy b using
                  continued fractions
cfsim     1-2   simplify number using continued fractions
char      1     character corresponding to integer value
cmp       2     compare values returning -1, 0, or 1
comb      2     combinatorial number a!/b!(a-b)!
config    1-2   set or read configuration value
conj      1     complex conjugate of value
copy      2-5   copy value to/from a block: copy(s,d,len,si,di)
cos       1-2   cosine of value a within accuracy b
cosh      1-2   hyperbolic cosine of a within accuracy b
```

```
cot        1-2   cotangent of a within accuracy b
coth       1-2   hyperbolic cotangent of a within accuracy b
count      2     count listr/matrix elements satisfying some condition
cp         2     cross product of two vectors
csc        1-2   cosecant of a within accuracy b
csch       1-2   hyperbolic cosecant of a within accuracy b
ctime      0     date and time as string
custom     0+    custom builtin function interface
delete     2     delete element from list a at position b
den        1     denominator of fraction
det        1     determinant of matrix
digit      2-3   digit at specified decimal place of number
digits     1-2   number of digits in base b representation of a
display    0-1   number of decimal digits for displaying numbers
dp         2     dot product of two vectors
epsilon    0-1   set or read allowed error for real calculations
errcount   0-1   set or read error count
errmax     0-1   set or read maximum for error count
errno      0-1   set or read calc_errno
error      0-1   generate error value
estr       1     exact text string representation of value
euler      1     Euler number
eval       1     evaluate expression from string to value
exp        1-2   exponential of value a within accuracy b
factor     1-3   lowest prime factor < b of a, return c if error
fcnt       2     count of times one number divides another
fib        1     Fibonacci number F(n)
forall     2     do function for all elements of list or matrix
frem       2     number with all occurrences of factor removed
fact       1     factorial
floor      1     greatest integer less than or equal to number
free       0+    free listed or all global variables
freebernoulli 0     free stored Bernoulli numbers
freeeuler 0      free stored Euler numbers
freeglobals 0      free all global and visible static variables
freeredc   0     free redc data cache
freestatics 0      free all unscoped static variables
frac       1     fractional part of value
gcd        1+    greatest common divisor
gcdrem     2     a divided repeatedly by gcd with b
gd         1-2   gudermannian function
hash       1+    return non-negative hash value for one or
                   more values
head       2     return list of specified number at head of a list
highbit    1     high bit number in base 2 representation
hmean      0+    harmonic mean of values
hnrmod     4     v mod h*2^n+r, h>0, n>0, r = -1, 0 or 1
hypot      2-3   hypotenuse of right triangle within accuracy c
ilog       2     integral log of a to integral base b
ilog10     1     integral log of a number base 10
ilog2      1     integral log of a number base 2
im         1     imaginary part of complex number
indices    2     indices of a specified assoc or mat value
inputlevel 0      current input depth
insert     2+    insert values c ... into list a at position b
int        1     integer part of value
inverse    1     multiplicative inverse of value
iroot      2     integer b'th root of a
isassoc    1     whether a value is an association
```

```
isblk      1     whether a value is a block
isconfig   1     whether a value is a config state
isdefined  1     whether a string names a function
iserror    1     where a value is an error
iseven     1     whether a value is an even integer
isfile     1     whether a value is a file
ishash     1     whether a value is a hash state
isident    1     returns 1 if identity matrix
isint      1     whether a value is an integer
islist     1     whether a value is a list
ismat      1     whether a value is a matrix
ismult     2     whether a is a multiple of b
isnull     1     whether a value is the null value
isnum      1     whether a value is a number
isobj      1     whether a value is an object
isobjtype  1     whether a string names an object type
isodd      1     whether a value is an odd integer
isoctet    1     whether a value is an octet
isprime    1-2   whether a is a small prime, return b if error
isptr      1     whether a value is a pointer
isqrt      1     integer part of square root
israndom   1     whether a value is a additive 55 state
israndom   1     whether a value is a Blum state
isreal     1     whether a value is a real number
isrel      2     whether two numbers are relatively prime
isstr      1     whether a value is a string
issimple   1     whether value is a simple type
issq       1     whether or not number is a square
istype     2     whether the type of a is same as the type of b
jacobi     2     -1 => a is not quadratic residue mod b
             1 => b is composite, or a is quad residue of b
join       1+    join one or more lists into one list
lcm        1+    least common multiple
lcmfact    1     lcm of all integers up till number
lfactor    2     lowest prime factor of a in first b primes
links      1     links to number or string value
list       0+    create list of specified values
ln         1-2   natural logarithm of value a within accuracy b
log        1-2   base 10 logarithm of value a within accuracy b
lowbit     1     low bit number in base 2 representation
ltol       1-2   leg-to-leg of unit right triangle (sqrt(1 - a^2))
makelist   1     create a list with a null elements
matdim     1     number of dimensions of matrix
matfill    2-3   fill matrix with value b (value c on diagonal)
matmax     2     maximum index of matrix a dim b
matmin     2     minimum index of matrix a dim b
matsum     1     sum the numeric values in a matrix
mattrace   1     return the trace of a square matrix
mattrans   1     transpose of matrix
max        0+    maximum value
memsize    1     number of octets used by the value, including overhead
meq        3     whether a and b are equal modulo c
min        0+    minimum value
minv       2     inverse of a modulo b
mmin       2     a mod b value with smallest abs value
mne        3     whether a and b are not equal modulo c
mod        2-3   residue of a modulo b, rounding type c
modify     2     modify elements of a list or matrix
name       1     name assigned to block or file
```

```
near       2-3   sign of (abs(a-b) - c)
newerror   0-1   create new error type with message a
nextcand   1-5   smallest value == d mod e > a, ptest(a,b,c) true
nextprime 1-2   return next small prime, return b if err
norm       1     norm of a value (square of absolute value)
null       0+    null value
num        1     numerator of fraction
ord        1     integer corresponding to character value
param      1     value of parameter n (or parameter count if n
                   is zero)
perm       2     permutation number a!/(a-b)!
prevcand   1-5   largest value == d mod e < a, ptest(a,b,c) true
prevprime 1-2   return previous small prime, return b if err
pfact      1     product of primes up till number
pi         0-1   value of pi accurate to within epsilon
pix        1-2   number of primes <= a < 2^32, return b if error
places     1-2   places after "decimal" point (-1 if infinite)
pmod       3     mod of a power (a ^ b (mod c))
polar      2-3   complex value of polar coordinate (a * exp(b*1i))
poly       1+    evaluates a polynomial given its coefficients
                   or coefficient-list
pop        1     pop value from front of list
popcnt     1-2   number of bits in a that match b (or 1)
power      2-3   value a raised to the power b within accuracy c
protect    1-3   read or set protection level for variable
ptest      1-3   probabilistic primality test
printf     1+    print formatted output to stdout
prompt     1     prompt for input line using value a
push       1+    push values onto front of list
quo        2-3   integer quotient of a by b, rounding type c
quomod     4-5   set c and d to quotient and remainder of a
                   divided by b
rand       0-2   additive 55 random number [0,2^64), [0,a), or [a,b)
randbit    0-1   additive 55 random number [0,2^a)
random     0-2   Blum-Blum-Shub random number [0,2^64), [0,a), or [a,b)
randombit 0-1   Blum-Blum-Sub random number [0,2^a)
randperm   1     random permutation of a list or matrix
rcin       2     convert normal number a to REDC number mod b
rcmul      3     multiply REDC numbers a and b mod c
rcout      2     convert REDC number a mod b to normal number
rcpow      3     raise REDC number a to power b mod c
rcsq       2     square REDC number a mod b
re         1     real part of complex number
remove     1     remove value from end of list
reverse    1     reverse a copy of a matrix or list
root       2-3   value a taken to the b'th root within accuracy c
round      1-3   round value a to b number of decimal places
rsearch    2-4   reverse search matrix or list for value b
                   starting at index c
runtime    0     user and kernel mode cpu time in seconds
saveval    1     set flag for saving values
scale      2     scale value up or down by a power of two
scan       1+    scan standard input for assignment to one
                   or more variables
scanf      2+    formatted scan of standard input for assignment
                   to variables
search     2-4   search matrix or list for value b starting
                   at index c
sec        1-2   sec of a within accuracy b
```

```
sech       1-2   hyperbolic secant of a within accuracy b
seed       0     return a 64 bit seed for a psuedo-random generator
segment    2-3   specified segment of specified list
select     2     form sublist of selected elements from list
setbit     2-3   set specified bit in string
sgn        1     sign of value (-1, 0, 1)
sha1       0+    Secure Hash Algorithm (SHS-1 FIPS Pub 180-1)
sin        1-2   sine of value a within accuracy b
sinh       1-2   hyperbolic sine of a within accuracy b
size       1     total number of elements in value
sizeof     1     number of octets used to hold the value
sleep      0-1   suspend operation for a seconds
sort       1     sort a copy of a matrix or list
sqrt       1-3   square root of value a within accuracy b
srand      0-1   seed the rand() function
srandom    0-4   seed the random() function
ssq        1+    sum of squares of values
stoponerror 0-1   assign value to stoponerror flag
str        1     simple value converted to string
strcat     1+    concatenate strings together
strcmp     2     compare two strings
strcpy     2     copy string to string
strerror   0-1   string describing error type
strlen     1     length of string
strncmp    3     compare strings a, b to c characters
strncpy    3     copy up to c characters from string to string
strpos     2     index of first occurrence of b in a
strprintf  1+    return formatted output as a string
strscan    2+    scan a string for assignments to one or more variables
strscanf   2+    formatted scan of string for assignments to variables
substr     3     substring of a from position b for c chars
sum        0+    sum of list or object sums and/or other terms
swap       2     swap values of variables a and b (can be dangerous)
systime    0     kernel mode cpu time in seconds
tail       2     retain list of specified number at tail of list
tan        1-2   tangent of a within accuracy b
tanh       1-2   hyperbolic tangent of a within accuracy b
test       1     test that value is nonzero
time       0     number of seconds since 00:00:00 1 Jan 1970 UTC
trunc      1-2   truncate a to b number of decimal places
ungetc     2     unget char read from file
usertime   0     user mode cpu time in seconds
version    0     calc version string
xor        1+    logical xor
```

```
The config function sets or reads the value of a configuration
parameter.  The first argument is a string which names the parameter
to be set or read.  If only one argument is given, then the current
value of the named parameter is returned.  If two arguments are given,
then the named parameter is set to the value of the second argument,
and the old value of the parameter is returned.  Therefore you can
change a parameter and restore its old value later.  The possible
parameters are explained in the next section.

The scale function multiplies or divides a number by a power of 2.
This is used for fractional calculations, unlike the << and >>
operators, which are only defined for integers.  For example,
scale(6, -3) is 3/4.
```

The quomod function is used to obtain both the quotient and remainder of a division in one operation. The first two arguments a and b are the numbers to be divided. The last two arguments c and d are two variables which will be assigned the quotient and remainder. For nonnegative arguments, the results are equivalent to computing a//b and a%b. If a is negative and the remainder is nonzero, then the quotient will be one less than a//b. This makes the following three properties always hold: The quotient c is always an integer. The remainder d is always 0 <= d < b. The equation a = b * c + d always holds. This function returns 0 if there is no remainder, and 1 if there is a remainder. For examples, quomod(10, 3, x, y) sets x to 3, y to 1, and returns the value 1, and quomod(-4, 3.14159, x, y) sets x to -2, y to 2.28318, and returns the value 1.

The eval function accepts a string argument and evaluates the expression represented by the string and returns its value. The expression can include function calls and variable references. For example, eval("fact(3) + 7") returns 13. When combined with the prompt function, this allows the calculator to read values from the user. For example, x=eval(prompt("Number: ")) sets x to the value input by the user.

The digit and bit functions return individual digits of a number, either in base 10 or in base 2, where the lowest digit of a number is at digit position 0. For example, digit(5678, 3) is 5, and bit(0b1000100, 2) is 1. Negative digit positions indicate places to the right of the decimal or binary point, so that for example, digit(3.456, -1) is 4.

The ptest builtin is a primality testing function. The 1st argument is the suspected prime to be tested. The absolute value of the 2nd argument is an iteration count.

If ptest is called with only 2 args, the 3rd argument is assumed to be 0. If ptest is called with only 1 arg, the 2nd argument is assumed to be 1. Thus, the following calls are equivalent:

```
ptest(a)
ptest(a,1)
ptest(a,1,0)
```

Normally ptest performs a some checks to determine if the value is divisable by some trivial prime. If the 2nd argument is < 0, then the trivial check is omitted.

For example, ptest(a,10) performs the same work as:

```
ptest(a,-3) (7 tests without trivial check)
ptest(a,-7,3)     (3 more tests without the trivial check)
```

The ptest function returns 0 if the number is definitely not prime, and 1 is the number is probably prime. The chance of a number which is probably prime being actually composite is less than 1/4 raised to the power of the iteration count. For example, for a random number p, ptest(p, 10) incorrectly returns 1 less than once in every million numbers, and you will probably never find a number where ptest(p, 20) gives

```
the wrong answer.

The first 3 args of nextcand and prevcand functions are the same
arguments as ptest.  But unlike ptest, nextcand and prevcand return
the next and previous values for which ptest is true.

For example, nextcand(2^1000) returns 2^1000+297 because
2^1000+297 is the smallest value x > 2^1000 for which
ptest(x,1) is true.  And for example, prevcand(2^31-1,10,5)
returns 2147483629 (2^31-19) because 2^31-19 is the largest
value y < 2^31-1 for which ptest(y,10,5) is true.

The nextcand and prevcand functions also have a 5 argument form:

      nextcand(num, count, skip, modval, modulus)
      prevcand(num, count, skip, modval, modulus)

return the smallest (or largest) value ans > num (or < num) that
is also == modval % modulus for which ptest(ans,count,skip) is true.

The builtins nextprime(x) and prevprime(x) return the
next and previous primes with respect to x respectively.
As of this release, x must be < 2^32.  With one argument, they
will return an error if x is out of range.  With two arguments,
they will not generate an error but instead will return y.

The builtin function pix(x) returns the number of primes <= x.
As of this release, x must be < 2^32.  With one argument, pix(x)
will return an error if x is out of range.  With two arguments,
pix(x,y) will not generate an error but instead will return y.

The builtin function factor may be used to search for the
smallest factor of a given number.  The call factor(x,y)
will attempt to find the smallest factor of x < min(x,y).
As of this release, y must be < 2^32.  If y is omitted, y
is assumed to be 2^32-1.

If x < 0, factor(x,y) will return -1.  If no factor <
min(x,y) is found, factor(x,y) will return 1.  In all other
cases, factor(x,y) will return the smallest prime factor
of x.  Note except for the case when abs(x) == 1, factor(x,y)
will not return x.

If factor is called with y that is too large, or if x or y
is not an integer, calc will report an error.  If a 3rd argument
is given, factor will return that value instead.  For example,
factor(1/2,b,c) will return c instead of issuing an error.

The builtin lfactor(x,y) searches a number of primes instead
of below a limit.  As of this release, y must be <= 203280221
(y <= pix(2^32-1)).  In all other cases, lfactor is operates
in the same way as factor.

If lfactor is called with y that is too large, or if x or y
is not an integer, calc will report an error.  If a 3rd argument
is given, lfactor will return that value instead.  For example,
lfactor(1/2,b,c) will return c instead of issuing an error.

The lfactor function is slower than factor.  If possible factor
```

should be used instead of lfactor.

The builtin isprime(x) will attempt to determine if x is prime. As of this release, x must be < 2^32. With one argument, isprime(x) will return an error if x is out of range. With two arguments, isprime(x,y) will not generate an error but instead will return y.

The functions rcin, rcmul, rcout, rcpow, and rcsq are used to perform modular arithmetic calculations for large odd numbers faster than the usual methods. To do this, you first use the rcin function to convert all input values into numbers which are in a format called REDC format. Then you use rcmul, rcsq, and rcpow to multiply such numbers together to produce results also in REDC format. Finally, you use rcout to convert a number in REDC format back to a normal number. The addition, subtraction, negation, and equality comparison between REDC numbers are done using the normal modular methods. For example, to calculate the value 13 * 17 + 1 (mod 11), you could use:

```
p = 11;
t1 = rcin(13, p);
t2 = rcin(17, p);
t3 = rcin(1, p);
t4 = rcmul(t1, t2, p);
t5 = (t4 + t3) % p;
answer = rcout(t5, p);
```

The swap function exchanges the values of two variables without performing copies. For example, after:

```
x = 17;
y = 19;
swap(x, y);
```

then x is 19 and y is 17. This function should not be used to swap a value which is contained within another one. If this is done, then some memory will be lost. For example, the following should not be done:

```
mat x[5];
swap(x, x[0]);
```

The hash function returns a relatively small non-negative integer for one or more input values. The hash values should not be used across runs of the calculator, since the algorithms used to generate the hash value may change with different versions of the calculator.

The base function allows one to specify how numbers should be printed. The base function provides a numeric shorthand to the config("mode") interface. With no args, base() will return the current mode. With 1 arg, base(val) will set the mode according to the arg and return the previous mode.

The following convention is used to declare modes:

```
 base  config
value  string

   2  "binary"    binary fractions
```

```
        8  "octal"           octal fractions
       10  "real"            decimal floating point
       16  "hex"       hexadecimal fractions
      -10  "int"       decimal integer
      1/3  "frac"            decimal fractions
     1e20  "exp"       decimal exponential

For convenience, any non-integer value is assumed to mean "frac",
and any integer >= 2^64 is assumed to mean "exp".
```

abs - absolute value

```
SYNOPSIS
    abs(x [,eps])

TYPES
    If x is an object of type xx, the function xx_abs has to have
      been defined; this will determine the types for x, eps and
      the returned value.

    For non-object x and eps:

    x       number (real or complex)
    eps           ignored if x is real, nonzero real for complex x,
            defaults to epsilon().

    return  non-negative real

DESCRIPTION
    If x is real, returns the absolute value of x, i.e. x if x >= 0,
    -x if x < 0.

    For complex x with zero real part, returns the absolute value of im(x).

    For other complex x, returns the multiple of eps nearest to the absolute
    value of x, or in the case of two equally near nearest values, the
    the nearest even multiple of eps.  In particular, with eps = 10^-n,
    the result will be the absolute value correct to n decimal places.

EXAMPLE
    ; print abs(3.4), abs(-3.4)
    3.4 3.4

    ; print abs(3+4i, 1e-5), abs(4+5i, 1e-5), abs(4+5i, 1e-10)
    5 6.40312 6.4031242374

LIMITS
    none

LINK LIBRARY
    NUMBER *qqabs(NUMBER *x)

SEE ALSO
    cmp, epsilon, hypot, norm, near, obj
```

acos - inverse trigonometric cosine

```
SYNOPSIS
    acos(x [,eps])

TYPES
    x       real, -1 <= x <= 1
    eps             nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the acos of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    v = acos(x) is the number in [0, pi] for which cos(v) = x.

EXAMPLE
    ; print acos(.5, 1e-5), acos(.5, 1e-10), acos(.5, 1e-15), acos(.5, 1e-20)
    1.0472 1.0471975512 1.047197551196598 1.04719755119659774615

LIMITS
    none

LINK LIBRARY
    NUMBER *qacos(NUMBER *x, NUMBER *eps)

SEE ALSO
    asin, atan, asec, acsc, acot, epsilon
```

acosh - inverse hyperbolic cosine

```
SYNOPSIS
    acosh(x [,eps])

TYPES
    x       real, x >= 1
    eps            nonzero real, defaults to epsilon()

    return  nonnegative real

DESCRIPTION
    Returns the acosh of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    acosh(x) is the nonnegative real number v for which cosh(v) = x.
    It is given by

             acosh(x) = ln(x + sqrt(x^2 - 1))

EXAMPLE
    ; print acosh(2, 1e-5), acosh(2, 1e-10), acosh(2, 1e-15), acosh(2, 1e-20)
    1.31696 1.3169578969 1.316957896924817 1.31695789692481670862

LIMITS
    none

LINK LIBRARY
    NUMBER *qacosh(NUMBER *x, NUMBER *eps)

SEE ALSO
    asinh, atanh, asech, acsch, acoth, epsilon
```

acot - inverse trigonometric cotangent

```
SYNOPSIS
    acot(x [,eps])

TYPES
    x       real
    eps            nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the acot of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    v = acot(x) is the number in (0, pi) for which cot(v) = x.

EXAMPLE
    ; print acot(2, 1e-5),  acot(2, 1e-10), acot(2, 1e-15), acot(2, 1e-20)
    .46365 .463647609 .463647609000806 .46364760900080611621

LIMITS
    none

LINK LIBRARY
    NUMBER *qacot(NUMBER *x, NUMBER *eps)

SEE ALSO
    asin, acos, atan, asec, acsc, epsilon
```

acoth - inverse hyperbolic cotangent

```
SYNOPSIS
    acoth(x [,eps])

TYPES
    x        real, with abs(x) > 1
    eps            nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the acoth of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    acoth(x) is the real number v for which coth(v) = x.
    It is given by
            acoth(x) = ln((x + 1)/(x - 1))/2

EXAMPLE
    ; print acoth(2, 1e-5), acoth(2, 1e-10), acoth(2, 1e-15), acoth(2, 1e-20)
    .54931 .5493061443 .549306144334055 .5493061443340548457

LIMITS
    none

LINK LIBRARY
    NUMBER *qacoth(NUMBER *x, NUMBER *eps)

SEE ALSO
    asinh, acosh, atanh, asech, acsch, epsilon
```

acsc - inverse trigonometric cosecant

```
SYNOPSIS
    acsc(x [,eps])

TYPES
    x        real, with absolute value >= 1
    eps            nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the acsc of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    v = acsc(x) is the number in [-pi/2, pi/2] for which csc(v) = x.

EXAMPLE
    ; print acsc(2, 1e-5), acsc(2, 1e-10), acsc(2, 1e-15), acsc(2, 1e-20)
    .5236 .5235987756 .523598775598299 .52359877559829887308

LIMITS
    none

LINK LIBRARY
    NUMBER *qacsc(NUMBER *x, NUMBER *eps)

SEE ALSO
    asin, acos, atan, asec, acot, epsilon
```

acsch - inverse hyperbolic cosecant

```
SYNOPSIS
    acsch(x [,eps])

TYPES
    x       nonzero real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the acsch of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    acsch(x) is the real number v for which csch(v) = x.  It is given by

            acsch(x) = ln((1 + sqrt(1 + x^2))/x)


EXAMPLE
    ; print acsch(2, 1e-5), acsch(2, 1e-10), acsch(2, 1e-15), acsch(2, 1e-20)
    .48121 .4812118251 .481211825059603 .4812118250596034475

LIMITS
    none

LINK LIBRARY
    NUMBER *qacsch(NUMBER *x, NUMBER *eps)

SEE ALSO
    asinh, acosh, atanh, asech, acoth, epsilon
```

agd - inverse gudermannian function

```
SYNOPSIS
    agd(z [,eps])

TYPES
    z       number (real or complex)
    eps           nonzero real, defaults to epsilon()

    return  number or infinite error value

DESCRIPTION
    Calculate the inverse gudermannian of z to a nultiple of eps with
    errors in real and imaginary parts less in absolute value than .75 * eps,
    or an error value if z is very close to one of the one of the branch
    points of agd(z)..

    agd(z) is usually defined initially for real z with abs(z) < pi/2 by
    one of the formulae

             agd(z) = ln(sec(z) + tan(z))

                  = 2 * atanh(tan(z/2))

                  = asinh(tan(z)),

    or as the integral from 0 to z of (1/cos(t))dt.  For complex z, the
    principal branch, approximated by gd(z, eps), has cuts along the real
    axis outside -pi/2 < z < pi/2.

    If z = x + i * y and abs(x) < pi/2, agd(z) is given by

      agd(z) = atanh(sin(x)/cosh(y)) + i * atan(sinh(y)/cos(x)>


EXAMPLE
    ; print agd(1, 1e-5), agd(1, 1e-10), agd(1, 1e-15)
    1.22619 1.2261911709 1.226191170883517

    ; print agd(2, 1e-5), agd(2, 1e-10)
    1.52345-3.14159i 1.5234524436-3.1415926536i

    ; print agd(5, 1e-5), agd(5, 1e-10), agd(5, 1e-15)
    -1.93237 -1.9323667197 -1.932366719745925

    ; print agd(1+2i, 1e-5), agd(1+2i, 1e-10)
    .22751+1.42291i .2275106584+1.4229114625i

LIMITS
    none

LINK LIBRARY
    COMPLEX *c_agd(COMPLEX *x, NUMBER *eps)

SEE ALSO
    gd, exp, ln, sin, sinh, etc.
```

append - append one or more values to end of list

```
SYNOPSIS
    append(x, y_0, y_1, ...)

TYPES
    x       lvalue whose value is a list
    y_0, ...      any

    return  null value

DESCRIPTION
    If after evaluation of y_0, y_1, ..., x is a list with contents
    (x_0, x_1, ...), then after append(x, y_0, y_1, ...), x has
    contents (x_0, x_1, ..., y_0, y_1, ...).

    If after evaluation of y_0, y_1, ..., x has size n,
    append(x, y_0, y_1, ...) is equivalent to insert(x, n, y_0, y_1, ...).

EXAMPLE
    ; x = list(2,3,4)
    ; append(x, 5, 6)
    ; print x

    list (5 elements, 5 nonzero):
      [[0]] = 2
      [[1]] = 3
      [[2]] = 4
      [[3]] = 5
      [[4]] = 6

    ; append(x, pop(x), pop(x))
    ; print x

    list (5 elements, 5 nonzero):
      [[0]] = 4
      [[1]] = 5
      [[2]] = 6
      [[3]] = 2
      [[4]] = 3

    ; append(x, (remove(x), 7))
    ; print x

    list (5 elements, 5 nonzero):
      [[0]] = 4
      [[1]] = 5
      [[2]] = 6
      [[3]] = 2
      [[4]] = 7

LIMITS
    append() can have at most 100 arguments

LINK LIBRARY
    none
```

```
SEE ALSO
     delete, insert, islist, pop, push, remove, rsearch, search,
     select, size
```

appr - approximate numbers by multiples of a specified number

```
SYNOPSIS
    appr(x [,y [,z]])

TYPES
    x       real, complex, matrix, list
    y       real
    z       integer

    return  same type as x except that complex x may return a real number

DESCRIPTION
    Return the approximate value of x as specified by a specific error
    (epsilon) and config ("appr") value.

    The default value for y is epsilon().  The default value for z is
    the current value of the "appr" configuration parameter.

    If y is zero or x is a multiple of y, appr(x,y,z) returns x.  I.e.,
    there is no "approximation" - the result represents x exactly.

    In the following it is assumed y is nonzero and x is not a multiple of y.
    For real x:

      appr(x,y,z) is either the nearest multiple of y greater
      than x or the nearest multiple of y less than x.  Thus, if
      we write a = appr(x,y,z) and r = x - a, then a/y is an integer
      and abs(r) < abs(y).  If r > 0, we say x has been "rounded down"
      to a; if r < 0, the rounding is "up".  For particular x and y,
      whether the rounding is down or up is determined by z.

      Only the 5 lowest bits of z are used, so we may assume z has been
      replaced by its value modulo 32.  The type of rounding depends on
      z as follows:

      z = 0 round down or up according as y is positive or negative,
            sgn(r) = sgn(y)

      z = 1 round up or down according as y is positive or negative,
            sgn(r) = -sgn(y)

      z = 2 round towards zero, sgn(r) = sgn(x)

      z = 3 round away from zero, sgn(r) = -sgn(x)

      z = 4 round down, r > 0

      z = 5 round up, r < 0

      z = 6 round towards or from zero according as y is positive or
            negative, sgn(r) = sgn(x/y)

      z = 7 round from or towards zero according as y is positive or
            negative, sgn(r) = -sgn(x/y)

      z = 8 a/y is even
```

```
  z = 9 a/y is odd

  z = 10      a/y is even or odd according as x/y is positive or negative

  z = 11      a/y is odd or even according as x/y is positive or negative

  z = 12      a/y is even or odd according as y is positive or negative

  z = 13      a/y is odd or even according as y is positive or negative

  z = 14      a/y is even or odd according as x is positive or negative

  z = 15      a/y is odd or even according as x is positive or negative

  z = 16 to 31      abs(r) <= abs(y)/2; if there is a unique multiple
        of y that is nearest x, appr(x,y,z) is that multiple of y
        and then abs(r) < abs(y)/2.  If x is midway between
        successive multiples of y, then abs(r) = abs(y)/2 and
        the value of a is as given by appr(x, y, z-16).

Matrix or List x:

  appr(x,y,z) returns the matrix or list indexed in the same way as x,
  in which each element t has been replaced by appr(t,y,z).

Complex x:

  Returns      appr(re(x), y, z) + appr(im(x), y, z) * 1i
```

PROPERTIES

```
  If appr(x,y,z) != x, then abs(x - appr(x,y,z)) < abs(y).

  If appr(x,y,z) != x and 16 <= z <= 31, abs(x - appr(x,y,z)) <= abs(y)/2.

  For z = 0, 1, 4, 5, 16, 17, 20 or 21, and any integer n,
        appr(x + n*y, y, z) = appr(x, y, z) + n * y.

  If y is nonzero, appr(x,y,8)/y = an odd integer n only if x = n * y.
```

EXAMPLES

```
; print appr(-5.44,0.1,0), appr(5.44,0.1,0), appr(5.7,1,0), appr(-5.7,1,0)
-5.5 5.4 5 -6

; print appr(-5.44,-.1,0), appr(5.44,-.1,0), appr(5.7,-1,0), appr(-5.7,-1,0)
-5.4 5.5 6 -5

; print appr(-5.44,0.1,3), appr(5.44,0.1,3), appr(5.7,1,3), appr(-5.7,1,3)
-5.5 5.5 6 -6

; print appr(-5.44,0.1,4), appr(5.44,0.1,4), appr(5.7,1,4), appr(-5.7,1,4)
-5.5 5.4 5 -6

; print appr(-5.44,0.1,6), appr(5.44,0.1,6), appr(5.7,1,6), appr(-5.7,1,6)
-5.4 5.4 6 -5

; print appr(-5.44,-.1,6), appr(5.44,-.1,6), appr(5.7,-1,6), appr(-5.7,-1,6)
-5.5 5.5 6 -6
```

```
    ; print appr(-5.44,0.1,9), appr(5.44,0.1,9), appr(5.7,1,9), appr(-5.7,1,9)
    -5.5 5.5 5 -5

    ; print appr(-.44,0.1,11), appr(.44,0.1,11), appr(5.7,1,11), appr(-5.7,1,11)
    -.4 .5 5 -6

    ; print appr(-.44,-.1,11),appr(.44,-.1,11),appr(5.7,-1,11),appr(-5.7,-1,11)
    -.5 .4 6 -5

    ; print appr(-.44,0.1,12), appr(.44,0.1,12), appr(5.7,1,12), appr(-5.7,1,12)
    -.4 .5 5 -6

    ; print appr(-.44,-.1,12),appr(.44,-.1,12),appr(5.7,-1,12),appr(-5.7,-1,12)
    -.5 .4 6 -5

    ; print appr(-.44,0.1,15), appr(.44,0.1,15), appr(5.7,1,15), appr(-5.7,1,15)
    -.4 .5 5 -6

    ; print appr(-.44,-.1,15),appr(.44,-.1,15),appr(5.7,-1,15),appr(-5.7,-1,15)
    -.4 .5 5 -6

    ; x = sqrt(7-3i, 1e-20)
    ; print appr(x,1e-5,0), appr(x,1e-5,1), appr(x,1e-5,2), appr(x,1e-6,3)
    2.70331-.55488i 2.70332-.55487i 2.70331-.55487i 2.70332-.55488i

LIMITS
    none

LINK LIBRARY
    NUMBER *qmappr(NUMBER *q, NUMBER *e, long R);
    LIST *listappr(LIST *oldlp, VALUE *v2, VALUE *v3);
    MATRIX *matappr(MATRIX *m, VALUE *v2, VALUE *v3);

SEE ALSO
    round, bround, cfappr, cfsim
```

arg - argument (the angle or phase) of a complex number

```
SYNOPSIS
    arg(x [,eps])

TYPES
    x       number
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the argument of x to the nearest or next to nearest multiple of
    eps; the error will be less in absolute value than 0.75 * abs(eps),
    but usually less than 0.5 * abs(eps).

EXAMPLE
    ; print arg(2), arg(2+3i, 1e-5), arg(2+3i, 1e-10), arg(2+3i, 1e-20)
    0 .98279 .9827937232 .98279372324732906799

    ; pi = pi(1e-10); deg = pi/180; eps = deg/10000
    ; print arg(2+3i, eps)/deg, arg(-1 +1i, eps)/deg, arg(-1 - 1i,eps)/deg
    56.3099 135 -135

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    conj, im, polar, re
```

asec - inverse trigonometric secant

```
SYNOPSIS
    asec(x [,eps])

TYPES
    x       real, with absolute value >= 1
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the asec of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    v = asec(x) is the number in [0, pi] for which sec(v) = x.

EXAMPLE
    ; print asec(2, 1e-5), asec(2, 1e-10), asec(2, 1e-15), asec(2, 1e-20)
    1.0472 1.0471975512 1.047197551196598 1.04719755119659774615

LIMITS
    none

LINK LIBRARY
    NUMBER *qasec(NUMBER *x, NUMBER *eps)

SEE ALSO
    asin, acos, atan, acsc, acot, epsilon
```

asech - inverse hyperbolic secant

```
SYNOPSIS
    asech(x [,eps])

TYPES
    x       real, 0 < x <= 1
    eps             nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the asech of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    asech(x) is the real number v for which sech(v) = x.  It is given by

            asech(x) = ln((1 + sqrt(1 - x^2))/x)

EXAMPLE
    ; print asech(.5,1e-5), asech(.5,1e-10), asech(.5,1e-15), asech(.5,1e-20)
    1.31696 1.3169578969 1.316957896924817 1.31695789692481670862

LIMITS
    none

LINK LIBRARY
    NUMBER *qasech(NUMBER *x, NUMBER *eps)

SEE ALSO
    asinh, acosh, atanh, acsch, acoth, epsilon
```

asin - inverse trigonometric sine

```
SYNOPSIS
    asin(x [,eps])

TYPES
    x       real, -1 <= x <= 1
    eps             nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the asin of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    v = asin(x) is the number in [-pi/2, pi/2] for which sin(v) = x.

EXAMPLE
    ; print asin(.5, 1e-5), asin(.5, 1e-10), asin(.5, 1e-15), asin(.5, 1e-20)
    .5236 .5235987756 .523598775598299 .52359877559829887308

LIMITS
    none

LINK LIBRARY
    NUMBER *qasin(NUMBER *q, NUMBER *epsilon)

SEE ALSO
    acos, atan, asec, acsc, acot, epsilon
```

asinh - inverse hyperbolic sine

```
SYNOPSIS
    asinh(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the asinh of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    asinh(x) is the real number v for which sinh(v) = x.  It is given by

            asinh(x) = ln(x + sqrt(1 + x^2))

EXAMPLE
    ; print asinh(2, 1e-5), asinh(2, 1e-10), asinh(2, 1e-15), asinh(2, 1e-20)
    1.44363 1.4436354752 1.443635475178881 1.44363547517881034249

LIMITS
    none

LINK LIBRARY
    NUMBER *qasinh(NUMBER *x, NUMBER *eps)

SEE ALSO
    acosh, atanh, asech, acsch, acoth, epsilon
```

=

```
SYNOPSIS
    a = b
    a = {e_1, e_2, ...[ { ... } ] }

TYPES
    a              lvalue, current value a structure in { } case

    b              expression

    e_0, e_1, ... expressions, blanks, or initializer lists


    return         lvalue (a)

DESCRIPTION
    Here an lvalue is either a simple variable specified by an identifier,
    or an element of an existing structure specified by one or more
    qualifiers following an identifier.

    An initializer list is a comma-separated list enclosed in braces as in

            {e_0, e_1, ... }

    where each e_i is an expression, blank or initializer list.

    a = b evaluates b, assigns its value to a, and returns a.

    a = {e_0, e_1, ... } where the e_i are expressions or blanks,
    requires the current value of a to be a matrix, list
    or object with at least as many elements as listed e_i.  Each non-blank
    e_i is evaluated and its value is assigned to a[[i]]; elements a[[i]]
    corresponding to blank e_i are unchanged.

    If, in a = {e_0, e_1, ...}, e_i is an initializer list, as in
    {e_i_0, e_1_1, ...}, the corresponding a[[i]] is to be a matrix, list
    or object with at least as many elements as listed e_i_j.  Depending on
    whether e_i_j is an expression, blank, or initializer list, one, no, or
    possibly more than one assignment, is made to a[[i]][[j]] or, if
    relevant and possible, its elements.

    In simple assignments, = associates from right to left so that, for
    example,

            a = b = c

    has the effect of a = (b = c) and results in assigning the value of c
    to both a and b.  The expression (a = b) = c is acceptable, but has the
    effect of a = b; a = c; in which the first assignment is superseded by
    the second.

    In initializations, = { ...} associates from left to right so that,
    for example,

            a = {e_0, ... } = {v_0, ...}
```

```
    first assigns e_0, ... to the elements of a, and then assigns v_0, ...
    to the result.

    If there are side effects in the evaluations involved in executing a = b,
    it should be noted that the order of evaluations is: first the address
    for a, then the value of b, and finally the assignment.  For example if
    A is a matrix and i = 0, then the assignment in A[i++] = A[i] is
    that of A[0] = A[1].

    If, in execution of a = b, a is changed by the evaluation of b, the
    value of b may be stored in an unintended or inaccessible location.  For
    example,
            mat A[2]= {1,2};
            A[0] = (A = 3);

    results in the value 3 being stored not only as the new value for A
    but also at the now unnamed location earlier used for A[0].


EXAMPLE
    ; b = 3+1
    ; a = b
    ; print a, b
    4 4

    ; obj point {x,y}
    ; mat A[3] = {1, list(2,3), obj point = {4,5}}

    ; A[1][[0]] = 6; A[2].x = 7
    ; print A[1]

    list (2 elements, 2 nonzero):
      [[0]] = 6
      [[1]] = 3

    ; print A[2]
    obj point {7, 5}

    ; A = {A[2], , {9,10}}
    ; print A[0]
    obj point {7, 5}

    ; print A[2]
    obj point {9, 10}

    ; A = {, {2}}
    print A[1]

    list (2 elements, 2 nonzero):
      [[0]] = 2
      [[1]] = 3

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
```

```
swap, quomod
```

assoc - create a new association array

```
SYNOPSIS
    assoc()

TYPES
    return  association

DESCRIPTION
    This function returns an empty association array.

    After A = assoc(), elements can be added to the association by
    assignments of the forms

      A[a_1] = v_1
      A[a_1, a_2] = v_2
      A[a_1, a_2, a_3] = v_3
      A[a_1, a_2, a_3, a_4] = v_4

    There are no restrictions on the values of the "indices" a_i or
    the "values" v_i.

    After the above assignments, so long as no new values have been
    assigned to A[a_i], etc., the expressions A[a_1], A[a_1, a_2], etc.
    will return the values v_1, v_2, ...

    Until A[a_1], A[a_1, a_2], ... are defined as described above, these
    expressions return the null value.

    Thus associations act like matrices except that different elements
    may have different numbers (between 1 and 4 inclusive) of indices,
    and these indices need not be integers in specified ranges.

    Assignment of a null value to an element of an association does not
    delete the element, but a later reference to that element will return
    the null value as if the element is undefined.

    The elements of an association are stored in a hash table for
    quick access.  The index values are hashed to select the correct
    hash chain for a small sequential search for the element.  The hash
    table will be resized as necessary as the number of entries in
    the association becomes larger.

    The size function returns the number of elements in an association.
    This size will include elements with null values.

    Double bracket indexing can be used for associations to walk through
    the elements of the association.  The order that the elements are
    returned in as the index increases is essentially random.  Any
    change made to the association can reorder the elements, this making
    a sequential scan through the elements difficult.

    The search and rsearch functions can search for an element in an
    association which has the specified value.  They return the index
    of the found element, or a NULL value if the value was not found.

    Associations can be copied by an assignment, and can be compared
```

```
    for equality.  But no other operations on associations have meaning,
    and are illegal.

EXAMPLE
    ; A = assoc(); print A
     assoc (0 elements):

    ; A["zero"] = 0; A["one"] = 1; A["two"] = 2; A["three"] = 3;
    ; A["smallest", "prime"] = 2;
    ; print A
    assoc (5 elements);
    ["two"] = 2
    ["three"] = 3
    ["one"] = 1
    ["zero"] = 0
    ["smallest","prime"] = 2

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, rsearch, search, size
```

atan - inverse trigonometric tangent

```
SYNOPSIS
    atan(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the atan of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    v = atan(x) is the number in (-pi/2, pi/2) for which tan(v) = x.

EXAMPLE
    ; print atan(2, 1e-5), atan(2, 1e-10), atan(2, 1e-15), atan(2, 1e-20)
    1.10715 1.1071487178 1.107148717794091 1.10714871779409050302

LIMITS
    none

LINK LIBRARY
    NUMBER *qatan(NUMBER *x, NUMBER *eps)

SEE ALSO
    asin, acos, asec, acsc, acot, epsilon
```

atan2 - angle to point

```
SYNOPSIS
    atan2(y, x, [,eps])

TYPES
    y       real
    x       real
    eps             nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    If x and y are not both zero, atan2(y, x, eps) returns, as a multiple of
    eps with error less than abs(eps), the angle t such that
    -pi < t <= pi and x = r * cos(t), y = r * sin(t), where
    r > 0.  Usually the error does not exceed abs(eps)/2.

    Note that by convention, y is the first argument; if x > 0,
    atan2(y, x) = atan(y/x).

    To conform to the 4.3BSD ANSI/IEEE 754-1985 math lib, atan2(0,0)
    returns 0.

EXAMPLE
    ; print atan2(0,0), atan2(1,sqrt(3)), atan2(17,53,1e-100)
    0 ~.52359877559829887307 ~.31038740713235146535

LIMITS
    none

LINK LIBRARY
    NUMBER *qatan2(NUMBER *y, *x, *acc)

SEE ALSO
    acos, asin, atan, cos, epsilon, sin, tan
```

atanh - inverse hyperbolic tangent

```
SYNOPSIS
    atanh(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns the atanh of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    atanh(x) is the real number v for which tanh(v) = x.  It is given by

            atanh(x) = ln((1 + x)/(1 - x))/2

EXAMPLE
    ; print atanh(.5,1e-5), atanh(.5,1e-10), atanh(.5,1e-15), atanh(.5,1e-20)
    .54931 .5493061443 .549306144334055 .5493061443340548457

LIMITS
    none

LINK LIBRARY
    NUMBER *qatanh(NUMBER *x, NUMBER *eps)

SEE ALSO
    asinh, acosh, asech, acsch, acoth, epsilon
```

avg - average (arithmetic) mean of values

```
SYNOPSIS
    avg(x_1, x_2, ...)

TYPES
    x_1, ...      arithmetic or list

    return  as determined by types of items averaged

DESCRIPTION
    If there are n non-list arguments x_1, x_2, ..., x_n,
    for which the required additions and division by n are defined,
    avg(x_1, x_2, ..., x_n) returns the value of:

      (x_1 + x_2 + ... + x_n)/n.

    If the x_i are real, the result will be a real number; if the
    x_i are real or complex numbers, the result will be a real or complex
    number.  If the x_i are summable matrices the result will be a matrix
    of the same size (e.g. if the x_i are all 3 x 4 matrices with real
    entries, the result will be a 3 x 4 matrix with real entries).

    If an argument x_i is list-valued, e.g. list(y_1, y_2, ...), this
    is treated as contributing y_1, y_2, ... to the list of items to
    be averaged.

EXAMPLE
    ; print avg(1,2,3,4,5), avg(list(1,2,3,4,5)), avg(1,2,list(3,4,5))
    3 3 3

    ; mat x[2,2] = {1,2,3,4}
    ; mat y[2,2] = {1,2,4,8}
    ; avg(x,y)

    mat [2,2] (4 elements, 4 nonzero):
      [0,0] = 1
      [0,1] = 2
      [1,0] = 3.5
      [1,1] = 6

LIMITS
   The number of arguments is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    hmean
```

base - set default output base

```
SYNOPSIS
    base([mode])

TYPES
    mode    real

    return  real

DESCRIPTION
    The base function allows one to specify how numbers should be
    printed.  The base function provides a numeric shorthand to the
    config("mode") interface.  With no args, base() will return the
    current mode.  With 1 arg, base(val) will set the mode according to
    the arg and return the previous mode.

    The following convention is used to declare modes:

           base   equivalent
                  config("mode")'s

             2    "binary"    base 2 fractions
                  "bin"

             8    "octal"           base 8 fractions
                  "oct"

            10    "real"            base 10 floating point
                  "float"
                  "default"

           -10    "integer"   base 10 integers
                  "int"

            16    "hexadecimal"     base 16 fractions
                  "hex"

           1/3    "fraction"  base 10 fractions
                  "frac"

          1e20    "scientific"      base 10 scientific notation
                  "sci"
                  "exp"

    For convenience, any non-integer value is assumed to mean base 10
    fractions and any integer >= 2^64 is assumed to mean base 10
    scientific notation.

    These base() calls have the same meaning as config("mode", "fraction"):

      base(1/3)   base(0.1415)      base(16/37)

    These base() calls have the same meaning as config("mode", "scientific"):

      base(1e20)  base(2^64)  base(2^8191-1)
```

```
    However the base() function will only return one of the base values
    listed in the table above.

EXAMPLE
    ; base()
          10

    ; base(8)
          012

    ; print 10
    012

LIMITS
    none

LINK LIBRARY
    int math_setmode(int newmode)

    NOTE: newmode must be one of MODE_DEFAULT, MODE_FRAC, MODE_INT,
        MODE_REAL, MODE_EXP, MODE_HEX, MODE_OCTAL, MODE_BINARY

SEE ALSO
    base2, config, str
```

base2 - set 2nd output base

```
SYNOPSIS
    base2([mode])

TYPES
    mode    real

    return  real

DESCRIPTION
    By default, calc will output values according to the default base
    as controlled by the base() builtin function.

    The base2() builtin function, if given a non-zero argument, enables
    double base output mode.  In double base output mode, calc values
    are displayed twice, once according to base() and again according
    to base2().  In double base output mode, the second time a value is
    displayed, it is displayed within comments:

      21701 /* 0x54c5 */

    The arguments for base2() are identical to base() with the addition
    of the 0 value:

           base2  equivalent
                  config("mode2")'s

             2    "binary"    base 2 fractions
                  "bin"

             8    "octal"           base 8 fractions
                  "oct"

            10    "real"            base 10 floating point
                  "float"
                  "default"

           -10    "integer"   base 10 integers
                  "int"

            16    "hexadecimal"     base 16 fractions
                  "hex"

           1/3    "fraction"  base 10 fractions
                  "frac"

          1e20    "scientific"      base 10 scientific notation
                  "sci"
                  "exp"

             0    "off"       disable double base output

    For convenience, any non-integer non-zero value is assumed to mean
    base 10 fractions and any integer >= 2^64 is assumed to mean base 10
    scientific notation.
```

```
    These base2() calls have the same meaning as config("mode2", "fraction"):

      base2(1/3)  base2(0.1415)     base2(16/37)

    These base2() calls have the same meaning as config("mode2", "scientific"):

      base2(1e20) base2(2^64) base2(2^8191-1)

    However the base2() function will only return one of the base values
    listed in the table above.

EXAMPLE
    ; base2()
          0
    ; base2(8)
          0 /* 0 */
    ; print 10
    10 /* 012 */
    ; base2(16),
    ; 131072
      131072 /* 0x20000 */
    ; 2345
      2345 /* 0x929 */

LIMITS
    none

LINK LIBRARY
    int math_setmode2(int newmode)

    NOTE: newmode must be one of MODE_DEFAULT, MODE_FRAC, MODE_INT,
        MODE_REAL, MODE_EXP, MODE_HEX, MODE_OCTAL, MODE_BINARY,
        MODE2_OFF

SEE ALSO
    base, config, str
```

bernoulli - Bernoulli number

```
SYNOPSIS
    bernoulli(n)

TYPES
    n       integer, n < 2^31 if even

    return  rational

DESCRIPTION
    Returns the Bernoulli number with index n, i.e. the coefficient B_n in
    the expansion

            t/(exp(t) - 1) = Sum B_n * t^n/n!

    bernoulli(n) is zero both for n < 0 and for n odd and > 2.
    When bernoulli(n) is computed for positive even n, the values for
    n and smaller positive even indices are stored in a table so that
    a later call to bernoulli(k) with 0 <= k < n will be executed quickly.

    Considerable runtime and memory are required for calculating
    bernoulli(n) for large even n.  For n = 1000, the numerator has
    1779 digits, the denominator 9 digits.

    The memory used to store calculated bernoulli numbers is freed by
    freebernoulli().

EXAMPLE
    ; config("mode", "frac"),;
    ; for (n = 0; n <= 6; n++) print bernoulli(n),; print;
    1 -1/2 1/6 0 -1/30 0 1/42

LIMITS
    n < 2^31-1

LIBRARY
    NUMBER *qbernoulli(long n)

SEE ALSO
    euler, catalan, comb, fact, perm
```

bit - whether a given binary bit is set in a value

```
SYNOPSIS
    bit(x, y)

TYPES
    x       real
    y       int

    return  int

DESCRIPTION
    Determine if the binary bit y is set in x.  If:

            x
      int(---) mod 2 == 1
           2^y

    return 1, otherwise return 0.

EXAMPLE
    ; print bit(9,0), bit(9,1), bit(9,2), bit(9,3)
    1 0 0 1

    ; print bit(9,4), bit(0,0), bit(9,-1)
    0 0 0

    ; print bit(1.25, -2), bit(1.25, -1), bit(1.25, 0)
    1 0 1

    ; p = pi()
    ; print bit(p, 1), bit(p, -2), bit(p, -3)
    1 0 1

LIMITS
    -2^31 < y < 2^31

LINK LIBRARY
    BOOL qbit(NUMBER *x, long y)

SEE ALSO
    highbit, lowbit, digit
```

blk - generate or modify block values

```
SYNOPSIS
    blk([len, chunk]);
    blk(val [, len, chunk]);

TYPES
    len           null or integer
    chunk   null or integer
    val           non-null string, block, or named block

    return  block or named block

DESCRIPTION
    With only integer arguments, blk(len, chunk) attempts to
    allocate a block of memory consisting of len octets (unsigned 8-bit
    bytes).  Allocation is always done in multiples of chunk
    octets, so the actual allocation size of len rounded up
    to the next multiple of chunk.

    The default value for len is 0.  The default value for chunk is 256.

    If the allocation is successful, blk(len, chunk) returns a value B, say,
    for which the octets in the block may be referenced by B[0], B[1],
    ... , B[len-1], these all initially having zero value.

    The octets B[i] for i >= len always have zero value.  If B[i] with
    some i >= len is referenced, size(B) is increased to i + 1.   For example:

                  B[i] = x

    has an effect like that of two operations on a file stream fs:

                  fseek(fs, pos);
                  fputc(fs, x).

    Similarly:

                  x = B[i]

    is like:

                  fseek(fs, pos);
                  x = fgetc(fs).

    The value of chunk is stored as the "chunksize" for B.

    The size(B) builtin returns the current len for the block; sizeof(B)
    returns its maxsize; memsize(B) returns maxsize + overhead for any block
    value.  Also size(B) is analogous to the length of a file stream in that
    if size(B) < sizeof(B):

                  B[size(B)] = x

    will append one octet to B and increment size(B).

    The builtin test(B) returns 1 or 0 according as at least one octet
```

is nonzero or all octets are zero. If B1 and B2 are blocks, they are considered equal (B1 == B2) if they have the same length and the same data, i.e. B1[i] == B2[i] for 0 <= i < len. Chunksizes and maxsizes are ignored.

The output for print B occupies two lines, the first line giving the chunksize, number of octets allocated (len rounded up to the next chunk) and len, and the second line up to 30 octets of data. If the datalen is zero, the second line is blank. If the datalen exceeds 30, this indicated by a trailing "...".

If a block value B created by B = blk(len, chunk) is assigned to another variable by C = B, a new block of the same structure as B is created to become the value of C, and the octets in B are copied to this new block. A block with possibly different length or chunksize is created by C = blk(B, newlen, newchunk), only the first min(len, newlen) octets being copied from B; later octets are assigned zero value. If omitted, newlen and newchunk default to the current datalen and chunk-size for B. The current datalen, chunksize and number of allocated octets for B may be changed by:

```
B = blk(B, newlen, newchunk).
```

No data is lost if newlen is greater than or equal to the old size(B).

The memory block allocated by blk(len, chunk) is freed at or before termination of the statement in which this occurred, the memory allocated in B = blk(len, chunk) is freed when B is assigned another value.

With a string str as its first argument, blk(str [, len, chunk]) when called for the first time creates a block with str as its name. Here there no restriction on the characters used in str; thus the string may include white space or characters normally used for punctuation or operators. Any subsequent call to blk(str, ...) with the same str will refer to the same named block.

A named block is assigned length and chunksize and consequent maximum size in the same way as unnamed blocks. A major difference is that in assignments, a named block is not copied. Thus, if a block A has been created by:

```
              A = blk("foo")
any subsequent:
              B = A
or:
              B = blk("foo")
```

will give a second variable B referring to the same block as A. Either A[i] = x or B[i] = x may then be used to assign a value to an octet in the block. Its length or chunksize may be changed by instructions like:

```
blk(A, len, chunk);

A = blk(A, len, chunk);

null(blk(A, len, chunk)).
```

These have the same effect on A; when working interactively, the last two avoid printing of the new value for A.

Named blocks are assigned index numbers 0, 1, 2, ..., in the order of their creation. The block with index id is returned by blocks(id). With no argument, blocks() returns the number of current unfreed named blocks.

The memory allocated to a named block is freed by the blkfree() function with argument the named block, its name, or its id number. The block remains in existence but with a null data pointer, its length and size being reduced to zero. A new block of memory may be allocated to it, with possibly new length and chunksize by:

```
blk(val [, len, chunk])
```

where val is either the named block or its name.

The printing output for a named block is in three lines, the first line displaying its id number and name, the other two as for an unnamed block, except that "NULL" is printed if the memory has been freed.

The identifying numbers and names of the current named blocks are displayed by:

```
show blocks
```

If A and B are named blocks, A == B will be true only if they refer to the same block of memory. Thus, blocks with the same data and datalen will be considered unequal if they have different names.

If A is a named block, str(A) returns the name of the block.

Values may be assigned to the early octets of a named or unnamed block by use of = { } initialization as for matrices.

EXAMPLE

```
; B = blk(15,10)

; B[7] = 0xff
; B
  chunksize = 10, maxsize = 20, datalen = 15
  00000000000000ff00000000000000

; B[18] = 127
; B
  chunksize = 10, maxsize = 20, datalen = 18
  00000000000000ff0000000000000000007f

; B[20] = 2
Index out of bounds for block

; print size(B), sizeof(B)
18 20

; B = blk(B, 100, 20)
; B
```

```
        chunksize = 20, maxsize = 120, datalen = 100
        00000000000000ff00000000000000000007f000000000000000000000000...

    ; C = blk(B, 10} = {1,2,3}
    ; C
        chunksize = 20, maxsize = 20, datalen = 10
        01020300000000ff0000

    ; A1 = blk("alpha")
    ; A1
        block 0: alpha
        chunksize = 256, maxsize = 256, datalen = 0

    ; A1[7] = 0xff
    ; A2 = A1
    ; A2[17] = 127
    ; A1
        block 0: alpha
        chunksize = 256, maxsize = 256, datalen = 18
        00000000000000ff00000000000000000007f

    ; A1 = blk(A1, 1000)
    ; A1
        block 0: alpha
        chunksize = 256, maxsize = 1024, datalen = 1000
        00000000000000ff00000000000000000007f000000000000000000000000...

    ; A1 = blk(A1, , 16)
    ; A1
        block 0: alpha
        chunksize = 16, maxsize = 1008, datalen = 1000
        00000000000000ff00000000000000000007f000000000000000000000000...
```

LIMITS

```
    0 <= len < 2^31

    1 <= chunk < 2^31
```

LINK LIBRARY

```
    BLOCK *blkalloc(int len, int chunk)
    void blk_free(BLOCK *blk)
    BLOCK *blkrealloc(BLOCK *blk, int newlen, int newchunk)
    void blktrunc(BLOCK *blk)
    BLOCK *blk_copy(BLOCK *blk)
    int blk_cmp(BLOCK *a, BLOCK *b)
    void blk_print(BLOCK *blk)
    void nblock_print(NBLOCK *nblk)
    NBLOCK *reallocnblock(int id, int len, int chunk)
    NBLOCK *createnblock(char *name, int len, int chunk)
    int findnblockid(char * name)
    int removenblock(int id)
    int countnblocks(void)
    void shownblocks(void)
    NBLOCK *findnblock(int id)
    BLOCK *copyrealloc(BLOCK *blk, int newlen, int newchunk)
```

SEE ALSO

```
    blocks, blkfree
```

blkcpy, copy - copy items from a structure to a structure

```
SYNOPSIS
    blkcpy(dst, src [, num [, dsi [, ssi]]]
    copy(src, dest [, [ssi [, num [, dsi]]]])

TYPES
    src            block, file, string, matrix, or list
    dest    block, file, matrix or list - compatible with src

    ssi            nonnegative integer, defaults to zero
    num            nonnegative integer, defaults to maximum possible
    dsi            nonnegative integer, defaults to datalen for a block, filepos
                   for a file, zero for other structures

    return  null if successful, error value otherwise

DESCRIPTION
    A call to:

      blkcpy(dst, src, num, dsi, ssi)

    attempts to copy 'num' consecutive items (octets or values) starting
    from the source item 'src' with index 'ssi'.  By default, 'num'
    is the maximum possible and 'ssi' is 0.

    A call to:

      copy(src, dst, ssi, num, dsi)

    does the same thing, but with a different arg order.

    A copy fails if ssi or num is too large for the number of items in
    the source, if sdi is too large for the number of positions
    available in the destination, or, in cases involving a file stream,
    if the file is not open in the required mode.  The source and
    destination need not be of the same type, e.g. when a block is
    copied to a matrix the octets are converted to numbers.

    The following pairs of source-type, destination-type are permitted:

      block to
              int
              block
              matrix
              file

      matrix to
              block
              matrix
              list

      string to
              block
              file

      list to
```

```
            list
            matrix

       file to
            block

       int to
            block
```

In the above table, int refers to integer values. However if a rational value is supplied, only the numerator is copied.

Each copied octet or value replaces the octet or value in the corresponding place in the destination structure. When copying values to values, the new values are stored in a buffer, the old values are removed, and the new values copied from the buffer to the destination. This permits movement of data within one matrix or list, and copying of an element of structure to the structure.

Except for copying to files or blocks, the destination is already to have sufficient memory allocated for the copying. For example, to copy a matrix M of size 100 to a newly created list, one may use:

```
      ; L = makelist(100);
      ; copy(M, L);
   or:
      ; L = makelist(100);
      ; blkcpy(L, M);
```

For copying from a block B (named or unnamed), the total number of octets available for copying is taken to the the datalen for that block, so that num can be at most size(B) - ssi.

For copying to a block B (named or unnamed), reallocation will be required if dsi + num > sizeof(B).   (This will not be permitted if protect(B) has bit 4 set.)

For copying from a file stream fs, num can be at most size(fs) - ssi.

For copying from a string str, the string is taken to include the terminating '\0', so the total number of octets available is strlen(str) + 1 and num can be at most strlen(str) + 1 - ssi. If num <= strlen(str) - ssi, the '\0' is not copied.

For copying from or to a matrix M, the total number of values in M is size(M), so in the source case, num <= size(M) - ssi, and in the destination case, num <= size(M) - dsi. The indices ssi and dsi refer to the double-bracket method of indexing, i.e. the matrix is as if its elements were indexed 0, 1, ..., size(M) - 1.

EXAMPLE

```
    ; A = blk() = {1,2,3,4}
    ; B = blk()
    ; blkcpy(B,A)
    ; B
      chunksize = 256, maxsize = 256, datalen = 4
      01020304
    ; blkcpy(B,A)
```

```
; B
  chunksize = 256, maxsize = 256, datalen = 8
  0102030401020304
; blkcpy(B, A, 2, 10)
; B
  chunksize = 256, maxsize = 256, datalen = 12
  010203040102030400000102
; blkcpy(B,32767)
; B
  chunksize = 256, maxsize = 256, datalen = 16
  010203040102030400000102ff7f0000
; mat M[2,2]
; blkcpy(M, A)
; M
  mat [2,2] (4 elements, 4 nonzero):
    [0,0] = 1
    [0,1] = 2
    [1,0] = 3
    [1,1] = 4
; blkcpy(M, A, 2, 2)
; M
  mat [2,2] (4 elements, 4 nonzero):
    [0,0] = 1
    [0,1] = 2
    [1,0] = 1
    [1,1] = 2

; A = blk() = {1,2,3,4}
; B = blk()
; copy(A,B)
; B
  chunksize = 256, maxsize = 256, datalen = 4
  01020304
; copy(A,B)
; B
  chunksize = 256, maxsize = 256, datalen = 8
  0102030401020304
; copy(A,B,1,2)
; B
  chunksize = 256, maxsize = 256, datalen = 10
  01020304010203040203
; mat M[2,2]
; copy(A,M)
; M
  mat [2,2] (4 elements, 4 nonzero):
    [0,0] = 1
    [0,1] = 2
    [1,0] = 3
    [1,1] = 4

; copy(A,M,2)
; M
  mat [2,2] (4 elements, 4 nonzero):
    [0,0] = 3
    [0,1] = 4
    [1,0] = 3
    [1,1] = 4

; copy(A,M,0,2,2)
```

```
    ; M
      mat [2,2] (4 elements, 4 nonzero):
        [0,0] = 3
        [0,1] = 4
        [1,0] = 1
        [1,1] = 2
```

```
LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    blk, mat, file, list, str
```

blkfree - free memory allocated to named block

```
SYNOPSIS
    blkfree(val)

TYPES
    val           named block, string, or integer

    return  null value

DESCRIPTION
    If val is a named block, or the name of a named block, or the
    identifying index for a named block, blkfree(val) frees the
    memory block allocated to this named block.  The block remains
    in existence with the same name, identifying index, and chunksize,
    but its size and maxsize becomes zero and the pointer for the start
    of its data block null.

    A new block of memory may be allocated to a freed block B by
    blk(B [, len, chunk]), len defaulting to zero and chunk to the
    chunksize when the block was freed.

EXAMPLE

    ; B1 = blk("foo")
    ; B2 = blk("Second block")
    show blocks
     id       name
    ----  -----
     0       foo
     1       Second block

    ; blkfree(B1)
    ; show blocks
     id       name
    ----  -----
     1       Second block

    ; B1
      block 0: foo
      chunksize = 256, maxsize = 0, datalen = 0
      NULL

    ; blk(B1); B[7] = 5
    ; B1
      block 0: foo
      chunksize = 256, maxsize = 256, datalen = 8
      0000000000000005

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    blk, blocks
```

blocks - return a named block or number of unfreed named blocks

```
SYNOPSIS
    blocks([id])

TYPES
    id            non-negative integer

    return  named block or null value

DESCRIPTION
    With no argument blocks() returns the number of blocks that have
    been created but not freed by the blkfree function.

    With argument id less than the number of named blocks that have been
    created, blocks(id) returns the named block with identifying index id.
    These indices 0, 1, 2, ... are assigned to named blocks in the order
    of their creation.

EXAMPLE

    ; A = blk("alpha")
    ; B = blk("beta") = {1,2,3}
    ; blocks()
      2
    ; blocks(1)
      block 1: beta
      chunksize = 256, maxsize = 256, datalen = 3
      010203
    ; blocks(2)
      Error 10211
    ; strerror()
      "Non-allocated index number for blocks"

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    blk, blkfree
```

bround - round numbers to a specified number of binary digits

```
SYNOPSIS
    bround(x [,plcs [, rnd]])

TYPES
    If x is a matrix or a list, bround(x[[i]], ...) is to return
    a value for each element x[[i]] of x; the value returned will be
    a matrix or list with the same structure as x.

    Otherwise, if x is an object of type tt, or if x is not an object or
    number but y is an object of type tt, and the function tt_bround has
    to be defined; the types for x, plcs, rnd, and the returned value,
    if any, are as required for specified in tt_bround.      For the object
    case, plcs and rnd default to the null value.

    For other cases:

    x       number (real or complex)
    plcs    integer, defaults to zero
    rnd           integer, defaults to config("round")

    return  number

DESCRIPTION
    For real x, bround(x, plcs, rnd) returns x rounded to either
    plcs significant binary digits (if rnd & 32 is nonzero) or to plcs
    binary places (if rnd & 32 is zero).  In the significant-figure
    case the rounding is to plcs - ilog10(x) - 1 binary places.
    If the number of binary places is n and eps = 10^-n, the
    result is the same as for appr(x, eps, rnd).  This will be
    exactly x if x is a multiple of eps; otherwise rounding occurs
    to one of the nearest multiples of eps on either side of x.    Which
    of these multiples is returned is determined by z = rnd & 31, i.e.
    the five low order bits of rnd, as follows:

          z = 0 or 4:         round down, i.e. towards minus infinity
          z = 1 or 5:         round up, i.e. towards plus infinity
          z = 2 or 6:         round towards zero
          z = 3 or 7:         round away from zero
          z = 8 or 12:  round to the nearest even multiple of eps
          z = 9 or 13:  round to the nearest odd multiple of eps
          z = 10 or 14: round to nearest even or odd multiple of eps
                            according as x > or < 0
          z = 11 or 15: round to nearest odd or even multiple of eps
                            according as x > or < 0
          z = 16 to 31: round to the nearest multiple of eps when
                            this is uniquely determined.  Otherwise
                            rounding is as if z is replaced by z - 16

    For complex x:

      The real and imaginary parts are rounded as for real x; if the
      imaginary part rounds to zero, the result is real.

    For matrix or list x:
```

```
      The returned values has element bround(x[[i]], plcs, rnd) in
      the same position as x[[i]] in x.

    For object x or plcs:

      When bround(x, plcs, rnd) is called, x is passed by address so may be
      changed by assignments; plcs and rnd are copied to temporary
      variables, so their values are not changed by the call.
```

EXAMPLES

```
    ; a = 7/32, b = -7/32

    ; print a, b
    .21875 -.21875

    ; print round(a,3,0), round(a,3,1), round(a,3,2), print round(a,3,3)
    .218, .219, .218, .219

    ; print round(b,3,0), round(b,3,1), round(b,3,2), print round(b,3,3)
    -.219, -.218, -.218, -.219

    ; print round(a,3,16), round(a,3,17), round(a,3,18), print round(a,3,19)
    .2188 .2188 .2188 .2188

    ; print round(a,4,16), round(a,4,17), round(a,4,18), print round(a,4,19)
    .2187 .2188 .2187 .2188

    ; print round(a,2,8), round(a,3,8), round(a,4,8), round(a,5,8)
    .22 .218 .2188 .21875

    ; print round(a,2,24), round(a,3,24), round(a,4,24), round(a,5,24)
    .22 .219 .2188 .21875

    ; c = 21875
    ; print round(c,-2,0), round(c,-2,1), round(c,-3,0), round(c,-3,16)
    21800 21900 21000 22000

    ; print round(c,2,32), round(c,2,33), round(c,2,56), round(c,4,56)
    21000 22000 22000 21880

    ; A = list(1/8, 2/8, 3/8, 4/8, 5/8, 6/8, 7/8)
    ; print round(A,2,24)

    list(7 elements, 7 nonzero):
      [[0]] = .12
      [[1]] = .25
      [[3]] = .38
      [[4]] = .5
      [[5]] = .62
      [[6]] = .75
      [[7]] = .88
```

LIMITS

```
    For non-object case:
      0 <= abs(plcs) < 2^31
      0 <= abs(rnd) < 2^31
```

LINK LIBRARY

```
    void broundvalue(VALUE *x, VALUE *plcs, VALUE *rnd, VALUE *result)
```

```
    MATRIX *matbround(MATRIX *m, VALUE *plcs, VALUE *rnd);
    LIST *listbround(LIST *m, VALUE *plcs, VALUE *rnd);
    NUMBER *qbround(NUMBER *m, long plcs, long rnd);
```

SEE ALSO

round, trunc, btrunc, int, appr

btrunc - truncate a value to a number of binary places

```
SYNOPSIS
    btrunc(x [,plcs])

TYPES
    x       real
    plcs    integer, defaults to zero

    return  real

DESCRIPTION
    Truncate x to plcs binary places, rounding if necessary towards zero,
    i.e. btrunc(x, plcs) is a multiple of 2^-plcs and the remainder
    x - btrunc(x, plcs) is either zero or has the same sign as x and
    absolute value less than 2^-plcs.  Here plcs may be positive, zero or
    negative.

    Except that it is defined only for real x, btrunc(x, plcs) is equivalent
    to bround(x, plcs, 2).  btrunc(x,0) and btrunc(x) are equivalent to
    int(x).

EXAMPLE
    ; print btrunc(pi()), btrunc(pi(), 10)
    3 3.140625

    ; print btrunc(3.3), btrunc(3.7), btrunc(3.3, 2), btrunc(3.7, 2)
    3 3 3.25 3.5

    ; print btrunc(-3.3), btrunc(-3.7), btrunc(-3.3, 2), btrunc(-3.7, 2)
    -3 -3 -3.25 -3.5

    ; print btrunc(55.123, -4), btrunc(-55.123, -4)
    48 -48

LIMITS
    abs(j) < 2^31

LINK LIBRARY
    NUMBER *qbtrunc(NUMBER *x, *j)

SEE ALSO
    bround, int, round, trunc
```

calclevel - current calculation level

```
SYNOPSIS
    calclevel()

TYPES
    return  nonnegative integer

DESCRIPTION
    This function returns the calculation level at which it is called.
    When a command is being read from a terminal or from a file,
    calc is at calculation level zero.    The level is increased
    by 1 each time calculation starts of a user-defined function
    or of eval(S) for some expression S which evaluates to a string.  It
    decreases to zero if an error occurs or a quit or abort statement
    is executed.  Otherwise, it decreases by 1 when the calculation
    is completed.  Except when an error occurs or abort is executed,
    the input level is not affected by changes in the calculation level.

    Zero calculation level is also called top calculation level; greater
    values of calclevel() indicate calculation is occurring at greater
    depths.

EXAMPLE
    n/a

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    eval, read, quit, abort, inputlevel
```

catalan - Catalan number

```
SYNOPSIS
    catalan(n)

TYPES
    n        integer

    return   integer

DESCRIPTION
    If n >= 0, this returns the Catalan number for index n:

      catalan(n) = comb(2*n,n)/(n + 1)

    Zero is returned for negative n.

    The Catalan numbers occur in solutions of several elementary
    combinatorial problems, e.g. for n >= 1, catalan(n) is the number of
    ways of using parentheses to express a product of n + 1 letters in
    terms of binary products; it is the number of ways of dissecting a
    convex polygon with n + 2 sides into triangles by nonintersecting
    diagonals; it is the number of integer-component-incrementing paths
    from (x,y) = (0,0) to (x,y) = (n,n) for which always y <= x.

EXAMPLE
    ; print catalan(2), catalan(3), catalan(4), catalan(20)
    2 5 14 6564120420

LIMITS
    none

LINK LIBRARY
    NUMBER *qcatalan(NUMBER *n)

SEE ALSO
    comb, fact, perm
```

ceil - ceiling

```
SYNOPSIS
    ceil(x)

TYPES
    x       real, complex, list, matrix

    return  real or complex, list, matrix

DESCRIPTION
    For real x, ceil(x) is the least integer not less than x.

    For complex, ceil(x) returns the real or complex number v for
    which re(v) = ceil(re(x)), im(v) = ceil(im(x)).

    For list or matrix x, ceil(x) returns the list or matrix of the
    same structure as x for which each element t of x has been replaced
    by ceil(t).

EXAMPLE
    ; print ceil(27), ceil(1.23), ceil(-4.56), ceil(7.8 - 9.1i)
    27 2 -4 8-9i

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    floor, int
```

cfappr - approximate a real number using continued fractions

```
SYNOPSIS
    cfappr(x [,eps [,rnd]]) or cfappr(x, n [,rnd])

TYPES
    x       real
    eps           real with abs(eps) < 1, defaults to epsilon()
    n       real with n >= 1
    rnd           integer, defaults to config("cfappr")

    return  real

DESCRIPTION
    If x is an integer or eps is zero, either form returns x.

    If abs(eps) < 1, cfappr(x, eps) returns the smallest-denominator
    number in one of the three intervals, [x, x + abs(eps)],
    [x - abs(eps], x], [x - abs(eps)/2, x + abs(eps)/2].

    If n >= 1 and den(x) > n, cfappr(x, n) returns the nearest above,
    nearest below, or nearest, approximation to x with denominator less
    than or equal to n.  If den(x) <= n, cfappr(x,n) returns x.

    In either case when the result v is not x, how v relates to x is
    determined by bits 0, 1, 2 and 4 of the argument rnd in the same way as
    these bits are used in the functions round() and appr().  In the
    following y is either eps or n.

            rnd   sign of remainder x - v

            0           sgn(y)
            1           -sgn(y
            2           sgn(x), "rounding to zero"
            3           -sgn(x), "rounding from zero"
            4           +, "rounding down"
            5           -, "rounding up"
            6           sgn(x/y)
            7           -sgn(x/y)

    If bit 4 of rnd is set, the other bits are irrelevant for the eps case;
    thus for 16 <= rnd < 24, cfappr(x, eps, rnd) is the smallest-denominator
    number differing from x by at most abs(eps)/2.

    If bit 4 of rnd is set and den(x) > 2, the other bits are irrelevant for
    the bounded denominator case; in the case of two equally near nearest
    approximations with denominator less than n, cfappr(x, n, rnd)
    returns the number with smaller denominator.   If den(x) = 2,  bits
    0, 1 and 2 of rnd are used as described above.

    If -1 < eps < 1, cfappr(x, eps, 0) may be described as the smallest
    denominator number in the closed interval with end-points x and x - eps.
    It follows that if abs(a - b) < 1, cfappr(a, a - b, 0) gives the smallest
    denominator number in the interval with end-points a and b; the same
    result is returned by cfappr(b, b - a, 0) or cfappr(a, b - a, 1).

    If abs(eps) < 1 and v = cfappr(x, eps, rnd), then
```

```
    cfappr(x, sgn(eps) * den(v), rnd) = v.

    If 1 <= n < den(x), u = cfappr(x, n, 0) and v = cfappr(x, n, 1), then
    u < x < v, den(u) <= n, den(v) <= n, den(u) + den(v) > n, and
    v - u = 1/(den(u) * den(v)).

    If x is not zero, the nearest approximation with numerator not
    exceeding n is 1/cfappr(1/x, n, 16).
```

EXAMPLE

```
    ; c = config("mode", "frac")
    ; x = 43/30; u = cfappr(x, 10, 0); v = cfappr(x, 10, 1);
    ; print u, v, x - u, v - x, v - u, cfappr(x, 10, 16)
    10/7 13/9 1/210 1/90 1/63 10/7

    ; pi = pi(1e-10)
    ; print cfappr(pi, 100, 16), cfappr(pi, .01, 16), cfappr(pi, 1e-6, 16)
    311/99 22/7 355/113

    ; x = 17/12; u = cfappr(x,4,0); v = cfappr(x,4,1);
    ; print u, v, x - u, v - x, cfappr(x,4,16)
    4/3 3/2 1/12 1/12 3/2
```

LIMITS

    none

LINK LIBRARY

```
    NUMBER *qcfappr(NUMBER *q, NUMBER *epsilon, long R)
```

SEE ALSO

    appr, cfsim

cfsim - simplify a value using continued fractions

```
SYNOPSIS
    cfsim(x [,rnd])

TYPES
    x       real
    rnd           integer, defaults to config("cfsim")

    return  real

DESCRIPTION
    If x is not an integer, cfsim(x, rnd) returns either the nearest
    above x, or the nearest below x, number with denominator less than
    den(x).  If x is an integer, cfsim(x, rnd) returns x + 1, x - 1, or 0.
    Which of the possible results is returned is controlled
    by bits 0, 1, 3 and 4 of the parameter rnd.

    For 0 <= rnd < 4, the sign of the remainder x - cfsim(x, rnd) is
    as follows:

            rnd         sign of x - cfsim(x, rnd)

            0           +, as if rounding down
            1           -. as if rounding up
            2           sgn(x), as if rounding to zero
            3           -sgn(x), as if rounding from zero

    This corresponds to the use of rnd for functions like round(x, n, rnd).

    If bit 3 or 4 of rnd is set, the lower order bits are ignored; bit 3
    is ignored if bit 4 is set.      Thusi, for rnd > 3, it sufficient to
    consider the two cases rnd = 8 and rnd = 16.

    If den(x) > 2, cfsim(x, 8) returns the value of the penultimate simple
    continued-fraction approximant to x, i.e. if:

      x = a_0 + 1/(a_1 + 1/(a_2 + ... + 1/a_n) ...)),

    where a_0 is an integer, a_1, ..., a_n are positive integers,
    and a_n >= 2, the value returned is that of the continued fraction
    obtained by dropping the last quotient 1/a_n.

    If den(x) > 2, cfsim(x, 16) returns the nearest number to x with
    denominator less than den(x).  In the continued-fraction representation
    of x described above, this is given by replacing a_n by a_n - 1.

    If den(x) = 2, the definition adopted is to round towards zero for the
    approximant case (rnd = 8) and from zero for the "nearest" case (rnd = 16).

    For integral x, cfsim(x, 8) returns zero, cfsim(x,16) returns x - sgn(x).

    In summary, for cfsim(x, rnd) when rnd = 8 or 16, the results are:

      rnd         integer x   half-integer x          den(x) > 2

       8          0           x - sgn(x)/2            approximant
```

```
 16          x - sgn(x)  x + sgn(x)/2            nearest
```

From either cfsim(x, 0) and cfsim(x, 1), the other is easily determined: if one of them has value w, the other has value (num(x) - num(w))/(den(x) - den(w)). From x and w one may find other optimal rational numbers near x; for example, the smallest-denominator number between x and w is (num(x) + num(w))/(den(x) + den(w)).

If x = n/d and cfsim(x, 8) = u/v, then for k * v < d, the k-th member of the sequence of nearest approximations to x with decreasing denominators on the other side of x is (n - k * u)/(d - k * v). This is nearer to or further from x than u/v according as 2 * k * v < or > d.

Iteration of cfsim(x,8) until an integer is obtained gives a sequence of "good" approximations to x with decreasing denominators and correspondingly decreasing accuracy; each denominator is less than half the preceding denominator. (Unlike the "forward" sequence of continued-fraction approximants these are not necessarily alternately greater than and less than x.)

Some other properties:

For rnd = 0 or 1 and any x, or rnd = 8 or 16 and x with den(x) > 2:

```
cfsim(n + x, rnd) = n + cfsim(x, rnd).
```

This equation also holds for the other values of rnd if n + x and x have the same sign.

For rnd = 2, 3, 8 or 16, and any x:

```
cfsim(-x, rnd) = -cfsim(x, rnd).
```

If rnd = 8 or 16, except for integer x or 1/x for rnd = 8, and zero x for rnd = 16:

```
cfsim(1/x, rnd) = 1/cfsim(x, rnd).
```

EXAMPLE

```
; c = config("mode", "frac");

; print cfsim(43/30, 0), cfsim(43/30, 1), cfsim(43/30, 8), cfsim(43/30,16)
10/7 33/23 10/7 33/23

; x = pi(1e-20); c = config("mode", "frac");
; while (!isint(x)) {x = cfsim(x,8); if (den(x) < 1e6) print x,:;}
1146408/364913 312689/99532 104348/33215 355/113 22/7 3
```

LIMITS

none

LINK LIBRARY

```
NUMBER *qcfsim(NUMBER *x, long rnd)
```

SEE ALSO

cfappr

char - character corresponding to a value

```
SYNOPSIS
    char(j)

TYPES
    j       integer, 0 <= j < 256

    return  string

DESCRIPTION
    For j > 0, returns a string of length 1 with a character that has
    the same value as j.  For j = 0, returns the null string "".

EXAMPLE
    ; print char(0102), char(0x6f), char(119), char(0145), char(0x6e)
    B o w e n

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    ord
```

cmp - compare two values of certain simple or object types

```
SYNOPSIS
    cmp(x, y)

TYPES
    If x is an object of type xx, or x is not an object and y is an object
    of type xx, the function xx_rel has to have been defined; any
    further conditions on x and y, and the type of the returned
    value depends on the definition of xx_rel.

    For non-object x and y:

    x       any
    y       any

    return  if x and y are both real: -1, 0, or 1
            if x and y are both numbers but not both real:
                  -1, 0, 1, -1+1i, 1i, 1+1i, -1-1i, -1i, or 1-1i
            if x and y are both strings: -1, 0, or 1
            all other cases: the null value

DESCRIPTION

    x and y both real: cmp(x, y) = sgn(x - y), i.e. -1, 0, or 1
      according as x < y, x == y, or x > y

    x and y both numbers, at least one being complex:
      cmp(x,y) = sgn(re(x) - re(y)) + sgn(im(x) - im(y)) * 1i

    x and y both strings: successive characters are compared until either
      different characters are encountered or at least one string is
      completed.  If the comparison ends because of different characters,
      cmp(x,y) = 1 or -1 according as the greater character is in x or y.
      If all characters compared in both strings are equal, then
      cmp(x,y) = -1, 0 or 1 according as the length of x is less than,
      equal to, or greater than the length of y.  (This comparison
      is performed via the strcmp() libc function.)

    objects: comparisons of objects are usually intended for some total or
      partial ordering and appropriate definitions of cmp(a,b) may
      make use of comparison of numerical or string components.
      definitions using comparison of numbers or strings are usually
      appropriate.  For example, after

            obj point {x,y};

      if points with real components are to be partially ordered by their
      euclidean distance from the origin, an appropriate point_rel
      function may be that given by

            define point_rel(a,b) = sgn(a.x^2 + a.y^2 - b.x^2 - b.y^2);

      A total "lexicographic" ordering is that given by:

            define point_rel(a,b) {
                  if (a.y != b.y)
```

```
                 return sgn(a.y - b.y);
             return (a.x - b.x);
         }
```

A comparison function that compares points analogously to cmp(a,b) for real and complex numbers is that given by

```
define point_rel(P1, P2) {
      return obj point = {sgn(P1.x-P2.x), sgn(P1.y-P2.y)};
}
```

The range of this function is the set of nine points with zero or unit components.

Some properties of cmp(a,b) for real or complex a and b are:

```
cmp(a + c, b + c) = cmp(a, b)

cmp(a, b) == 0 if and only if a == b

cmp(b, a) = -cmp(a, b)

if c is real or pure imaginary, cmp(c * a, c * b) = c * cmp(a,b)
```

Then a function that defines "b is between a and c" in an often useful sense is

```
define between(a,b,c) = (cmp(a,b) == cmp(b,c)).
```

For example, in this sense, 3 + 4i is between 1 + 5i and 4 + 2i.

Note that using cmp to compare non-object values of different types, for example, cmp(2, "2"), returns the null value.

EXAMPLE

```
; print cmp(3,4), cmp(4,3), cmp(4,4), cmp("a","b"), cmp("abcd","abc")
-1 1 0 -1 1

; print cmp(3,4i), cmp(4,4i), cmp(5,4i), cmp(-5,4i), cmp(-4i,5), cmp(-4i,-5)
1-1i 1-1i 1-1i -1-1i -1-1i 1-1i

; print cmp(3i,4i), cmp(4i,4i), cmp(5i,4i), cmp(3+4i,5), cmp(3+4i,-5)
-1i 0 1i -1+1i 1+1i

; print cmp(3+4i,3+4i), cmp(3+4i,3-4i), cmp(3+4i,2+3i), cmp(3+4i,-4-5i)
0 1i 1+1i 1+1i
```

LIMITS

none

LINK LIBRARY

```
FLAG qrel(NUMBER *q1, NUMBER *q2)
FLAG zrel(ZVALUE z1, ZVALUE z2)
```

SEE ALSO

sgn, test, operator

comb - combinatorial number

```
SYNOPSIS
    comb(x, y)

TYPES
    x        integer
    y        integer

    return   integer

DESCRIPTION
    Return the combinatorial number C(x,y) which is defined as:

               x!
            ---------
            y!*(x-y)!

    This function computes the number of combinations in which y things
    may be chosen from x items ignoring the order in which they are chosen.

EXAMPLE
    ; print comb(7,3), comb(7,4), comb(7,5), comb(3,0), comb(0,0)
    35 35 21 1 1

    ; print comb(2^31+1,2^31-1)
    2305843010287435776

LIMITS
    x >= y >= 0
    y < 2^24
    x-y < 2^24

LINK LIBRARY
    void zcomb(ZVALUE x, ZVALUE y, ZVALUE *res)

SEE ALSO
    fact, perm, randperm
```

conj - complex conjugate

```
SYNOPSIS
    conj(x)

TYPES
    If x is an object of type xx, conj(x) calls xx_conj(x).

    For non-object x:

    x       real, complex, or matrix

    return  real, complex, or matrix

DESCRIPTION

    For real x, conj(x) returns x.

    For complex x, conj(x) returns re(x) - im(x) * 1i.

    For matrix x, conj(x) returns a matrix of the same structure as x
    in which each element t of x has been replaced by conj(t).

    For xx objects, xx_conj(a) may return any type of value, but
    for the properties usually expected of conjugates, xx_conj(a)
    would return an xx object in which each number component is the
    conjugate of the corresponding component of a.

EXAMPLE
    ; print conj(3), conj(3 + 4i)
    3 3-4i

LIMITS
    none

LINK LIBRARY
    void conjvalue(VALUE *x, *res)

SEE ALSO
    norm, abs, arg
```

cos - cosine

```
SYNOPSIS
    cos(x [,eps])

TYPES
    x       number (real or complex)
    eps            nonzero real, defaults to epsilon()

    return  number

DESCRIPTION
    Calculate the cosine of x to a multiple of eps with error less in
    absolute value than .75 * eps.

EXAMPLE
    ; print cos(1, 1e-5), cos(1, 1e-10), cos(1, 1e-15), cos(1, 1e-20)
    .5403 .5403023059 .54030230586814 .5403023058681397174

    ; print cos(2 + 3i, 1e-5), cos(2 + 3i, 1e-10)
    -4.18963-9.10923i -4.189625691-9.1092278938i

    ; pi = pi(1e-20)
    ; print cos(pi/3, 1e-10), cos(pi/2, 1e-10), cos(pi, 1e-10)
    .5 0 -1

LIMITS
    none

LINK LIBRARY
    NUMBER *qcos(NUMBER *x, NUMBER *eps)
    COMPLEX *c_cos(COMPLEX *x, NUMBER *eps)

SEE ALSO
    sin, tan, sec, csc, cot, epsilon
```

cosh - hyperbolic cosine

```
SYNOPSIS
    cosh(x [,eps])

TYPES
    x       real
    eps             nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the cosh of x to the nearest or next to nearest multiple of
    epsilon, with absolute error less than .75 * abs(eps).

    cosh(x) = (exp(x) + exp(-x))/2

EXAMPLE
    ; print cosh(1, 1e-5), cosh(1, 1e-10), cosh(1, 1e-15), cosh(1, 1e-20)
    1.54308 1.5430806348 1.543080634815244 1.54308063481524377848

LIMITS
    none

LINK LIBRARY
    NUMBER *qcosh(NUMBER *x, NUMBER *eps)

SEE ALSO
    sinh, tanh, sech, csch, coth, epsilon
```

cot - trigonometric cotangent

```
SYNOPSIS
    cot(x [,eps])

TYPES
    x       nonzero real
    acc           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the cotangent of x to a multiple of eps, with error less
    in absolute value than .75 * eps.

EXAMPLE
    ; print cot(1, 1e-5), cot(1, 1e-10), cot(1, 1e-15), cot(1, 1e-20)
    .64209 .6420926159 .642092615934331 .64209261593433070301

LIMITS
    none

LINK LIBRARY
    NUMBER *qcot(NUMBER *x, NUMBER *eps)

SEE ALSO
    sin, cos, tan, sec, csc, epsilon
```

coth - hyperbolic cotangent

```
SYNOPSIS
    coth(x [,eps])

TYPES
    x       nonzero real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the coth of x to a multiple of eps with error less in
    absolute value than .75 * eps.

    coth(x) = (exp(2*x) + 1)/(exp(2*x) - 1)

EXAMPLE
    ; print coth(1, 1e-5), coth(1, 1e-10), coth(1, 1e-15), coth(1, 1e-20)
    1.31304 1.3130352855 1.313035285499331 1.31303528549933130364

LIMITS
    none

LINK LIBRARY
    NUMBER *qcoth(NUMBER *x, NUMBER *eps)

SEE ALSO
    sinh, cosh, tanh, sech, csch, epsilon
```

count - count elements of list or matrix satisfying a stated condition

```
SYNOPSIS
    count(x, y)

TYPES
    x        list or matrix
    y        string

    return   integer

DESCRIPTION
    For count(x, y), y is to be the name of a user-defined function;
    count(x,y) then returns the number of elements of x for which y
    tests as "true".

EXAMPLE
    ; define f(a) = (a < 5)
    ; A = list(1,2,7,6,4,8)
    ; count(A, "f")
      3

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    select, modify
```

cp - cross product of two 3 element vectors

```
SYNOPSIS
    cp(x, y)

TYPES
    x, y    1-dimensional matrices with 3 elements

    return  1-dimensional matrix with 3 elements

DESCRIPTION
    Calculate the product of two 3 1-dimensional matrices.
    If x has elements (x0, x1, x2), and y has elements (y0, y1, y2),
    cp(x,y) returns the matrix of type [0:2] with elements:

      {x1 * y2 - x2 * y1, x3 * y1 - x1 * y3, x1 * y2 - x2 * y1}

EXAMPLE
    ; mat x[3] = {2,3,4}
    ; mat y[3] = {3,4,5}
    ; print cp(x,y)

    mat [3] (3 elements, 3 nonzero):
      [0] = -1
      [1] = 2
      [2] = -1

LIMITS
    The components of the matrices are to be of types for which the
    required algebraic operations have been defined.

LINK LIBRARY
    MATRIX *matcross(MATRIX *x, MATRIX *y)

SEE ALSO
    dp
```

csc - trigonometric cosecant function

```
SYNOPSIS
    csc(x [,eps])

TYPES
    x       real
    eps             nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the cosecant of x to a multiple of eps, with error less
    in absolute value than .75 * eps.

EXAMPLE
    ; print csc(1, 1e-5), csc(1, 1e-10), csc(1, 1e-15), csc(1, 1e-20)
    1.1884 1.1883951058 1.188395105778121 1.18839510577812121626

LIMITS
    none

LINK LIBRARY
    NUMBER *qcsc(NUMBER *x, NUMBER *eps)

SEE ALSO
    sin, cos, tan, sec, cot, epsilon
```

csch - hyperbolic cosecant

```
SYNOPSIS
    csch(x [,eps])

TYPES
    x       nonzero real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the csch of x to a multiple of epsilon, with error less in
    absolute value than .75 * eps.

    csch(x) = 2/(exp(x) - exp(-x))

EXAMPLE
    ; print csch(1, 1e-5), csch(1, 1e-10), csch(1, 1e-15), csch(1, 1e-20)
    .85092 .8509181282 .850918128239322 .85091812823932154513

LIMITS
    none

LINK LIBRARY
    NUMBER *qcsch(NUMBER *x, NUMBER *eps)

SEE ALSO
    sinh, cosh, tanh, sech, coth, epsilon
```

ctime - current local time

```
SYNOPSIS
    ctime()

TYPES
    return  string

DESCRIPTION
   The ctime() builtin returns the string formed by the first 24
   characters returned by the C library function, ctime(): E.g.

      "Mon Oct 28 00:47:00 1996"

   The 25th ctime() character, '\n' is removed.

EXAMPLE
    ; printf("The time is now %s.\n", ctime())
    The time is now Mon Apr 15 12:41:44 1996.

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    runtime, time
```

delete - delete an element from a list at a specified position

```
SYNOPSIS
    delete(lst, index)

TYPES
    lst           list
    index   nonnegative integer less than the size of the list

    return  type of the deleted element

DESCRIPTION
     Deletes element at the specified index from list lst, and returns
     the value of this element.

EXAMPLE
    ; lst = list(2,3,4,5)

    list (4 elements, 4 nonzero):
      [[0]] = 2
      [[1]] = 3
      [[2]] = 4
      [[3]] = 5

    ; delete(lst, 2)
          4
    ; print lst

    list (3 elements, 3 nonzero):
      [[0]] = 2
      [[1]] = 3
      [[2]] = 5

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     append, insert, islist, pop, push, remove, rsearch, search,
     select, size
```

den - denominator of a real number

```
SYNOPSIS
    den(x)

TYPES
    x       real

    return  integer

DESCRIPTION
    For real x, den(x) returns the denominator of x when x is expressed
    in lowest terms with positive denominator.  In calc,
    real values are actually rational values.  Each calc real
    value can be uniquely expressed as:

      n / d

    where:

      n and d are integers
      gcd(n,d) == 1
      d > 0

    The denominator for this n/d is d.

EXAMPLE
    ; print den(7), den(-1.25), den(121/33)
    1 4 3

LIMITS
    none

LINK LIBRARY
    NUMBER *qden(NUMBER *x)

SEE ALSO
    num
```

det - determinant

```
SYNOPSIS
    det(m)

TYPES
    m       square matrix with elements of suitable type

    return  zero or value of type determined by types of elements

DESCRIPTION
    The matrix m has to be square, i.e. of dimension 2 with:

      matmax(m,1) - matmin(m,1) == matmax(m,2) - matmin(m,2).

    If the elements of m are numbers (real or complex), det(m)
    returns the value of the determinant of m.

    If some or all of the elements of m are not numbers, the algorithm
    used to evaluate det(m) assumes the definitions of *, unary -, binary -,
    being zero or nonzero, are consistent with commutative ring structure,
    and if the m is larger than 2 x 2, division by nonzero elements is
    consistent with integral-domain structure.

    If m is a 2 x 2 matrix with elements a, b, c, d, where a tests as
    nonzero, det(m) is evaluated by

      det(m) = (a * d) - (c * b).

    If a tests as zero, det(m) = - ((c * b) - (a * d)) is used.

    If m is 3 * 3 with elements a, b, c, d, e, f, g, h, i, where a and
    a * e - d * b test as nonzero, det(m) is evaluated by

      det(m) = ((a * e - d * b) * (a * i - g * c)
                   - (a * h - g * b) * (a * f - d * c))/a.

EXAMPLE
    ; mat A[3,3] = {2, 3, 5, 7, 11, 13, 17, 19, 23}
    ; c = config("mode", "frac")
    ; print det(A), det(A^2), det(A^3), det(A^-1)
    -78 6084 -474552 -1/78

    ; obj res {r}
    ; global md
    ; define res_test(a) = !ismult(a.r, md)
    ; define res_sub(a,b) {local obj res v = {(a.r - b.r) % md}; return v;}
    ; define res_mul(a,b) {local obj res v = {(a.r * b.r) % md}; return v;}
    ; define res_neg(a) {local obj res v = {(-a.r) % md}; return v;}
    ; define res(x) {local obj res v = {x % md}; return v;}
    ; md = 0
    ; mat A[2,2] = {res(2), res(3), res(5), res(7)}
    ; md = 5
    ; print det(A)
    obj res {4}
    ; md = 6
    ; print det(A)
```

```
    obj res {5}

    Note that if A had been a 3 x 3 or larger matrix, res_div(a,b) for
    non-zero b would have had to be defined (assuming at least one
    division is necessary); for consistent results when md is composite,
    res_div(a,b) should be defined only when b and md are relatively
    prime; there is no problem when md is prime.
```

LIMITS

    none

LINK LIBRARY

    VALUE matdet(MATRIX *m)

SEE ALSO

    matdim, matmax, matmin, inverse

digit - digit at specified position in a "decimal" representation

```
SYNOPSIS
    digit(x, n [, b])

TYPES
    x       real
    n       integer
    b       integer >= 2, default = 10

    return  integer

DESCRIPTION

    d(x,n,b) returns the digit with index n in a standard base-b "decimal"
    representation of x, which may be described as follows:

    For an arbitrary base b >= 2, following the pattern of decimal (base 10)
    notation in elementary arithmetic, a base-b "decimal" representation of
    a positive real number may be considered to be specified by a finite or
    infinite sequence of "digits" with possibly a "decimal" point
    to indicate where the fractional part of the representation begins.
    Just as the digits for base 10 are the integers 0, 1, 2, ..., 9, the
    digits for a base-b representation are the integers d for which
    0 <= d < b.   The index for a digit position is the count, positively to
    the left, of the number from the "units" position immediately to the
    left of the "decimal" point; the digit  d_n  at position n contributes
    additively  d_n * b^n  to the value of x.  For example,

      ; d_2 d_1 d_0 . d_-1 d_-2

    represents the number

      ; d_2 * b^2 + d_1 * b + d0 + d_-1 * b^-1 + d_-2 * b^-2

    The sequence of digits has to be infinite if den(x) has a prime factor
    which is not a factor of the base b.  In cases where the representation
    may terminate, the digits are considered to continue with an infinite
    string of zeros rather than the other possibility of an infinite
    sequence of (b - 1)s.  Thus, for the above example, d_n = 0 for
    n = -3, -4, ...  Similarly, a representation may be considered to
    continue with an infinite string of zeros on the left, so that in the
    above example d_n = 0 also for n >= 3.

    For negative x, digit(x,n,b) is given by digit(abs(x),n,b); the
    standard "decimal" representation of this x is a - sign followed by
    the representation of abs(x).

    In calc, the "real" numbers are all rational and for these the
    digits following the decimal point eventually form a recurring sequence.

    With base-b digits for x as explained above, the integer whose base-b
    representation is

      ; b_n+k-1 b_n_k-2 ... b_n,

    i.e. the k digits with last digit b_n, is given by
```

```
        ; digit(b^-r * x, q, b^k)

    if r and q satisfy  n = q * b + r.
```

EXAMPLE

```
        ; a = 123456.789
        ; for (n = 6; n >= -6; n++) print digit(a, n),; print
        0 1 2 3 4 5 6 7 8 9 0 0 0

        ; for (n = 6; n >= -6; n--) print digit(a, n, 100),; print
        0 0 0 0 12 34 56 78 90 0 0 0 0

        ; for (n = 6; n >= -6; n--) print digit(a, n, 256),; print
        0 0 0 0 1 226 64 201 251 231 108 139 67

        ; for (n = 1; n >= -12; n++) print digit(10/7, n),; print
        ; 0 1 4 2 8 5 7 1 4 2 8 5 7 1

        ; print digit(10/7, -7e1000, 1e6)
        428571
```

LIMITS

The absolute value of the integral part of x is assumed to be less than 2^2^31, ensuring that digit(x, n, b) will be zero if n >= 2^31. The size of negative n is limited only by the capacity of the computer being used.

LINK LIBRARY

```
    NUMBER * qdigit(NUMBER *q, ZVALUE dpos, ZVALUE base)
```

SEE ALSO

bit

digits - return number of "decimal" digits in an integral part

```
SYNOPSIS
    digits(x [,b])

TYPES
    x        real
    b        integer >= 2, defaults to 10

    return   integer

DESCRIPTION
    Returns number of digits in the standard base-b representation
    when x is truncated to an integer and the sign is ignored.

    To be more precise: when abs(int(x)) > 0, this function returns
    the value 1 + ilog(x, b).  When abs(int(x)) == 0, then this
    function returns the value 1.

    If omitted, b is assumed to be 10.  If given, b must be an
    integer > 1.

    One should remember these special cases:

      digits(12.3456) == 2     computes with the integer part only
      digits(-1234) == 4       computes with the absolute value only
      digits(0) == 1           specical case
      digits(-0.123) == 1      combination of all of the above

EXAMPLE
    ; print digits(100), digits(23209), digits(2^72)
    3 5 22

    ; print digits(0), digits(1), digits(-1)
    1 1 1

    ; print digits(-1234), digits(12.3456), digits(107.207)
    4 2 3

    ; print digits(17^463-1, 17), digits(10000, 100), digits(21701, 2)
    3, 15 14

LIMITS
    b > 1

LINK LIBRARY
    long qdigits(NUMBER *q, ZVALUE base)

SEE ALSO
    digit, places
```

display - set and/or return decimal digits for displaying numbers

```
SYNOPSIS
    display([d])

TYPES
    d       integer >= 0

    return  integer

DESCRIPTION
    When given an argument, this function sets the maximum number of
    digits after the decimal point to be printed in real or exponential
    mode in normal unformatted printing (print, strprint, fprint) or in
    formatted printing (printf, strprintf, fprintf) when precision is
    not specified.  The return value is the previous display digit value.

    When given no arguments, this function returns the current
    display digit value.

    The builtin function:

      display(d)
      display()

    is an alias for:

      config("display", d)
      config("display")

    The display digit value does not change the stored value of a number.
    It only changes how a stored value is displayed.

    Where rounding is necessary to display up to d decimal places,
    the type of rounding to be used is controlled by config("outround").

EXAMPLE
    ; print display(), 2/3
    20 ~0.66666666666666666667

    ; print display(40), 2/3
    20 ~0.6666666666666666666666666666666666666667

    ; print display(5), 2/3
    40 ~0.66667

LIMITS
    d >= 0

LINK LIBRARY
    none

SEE ALSO
    config
```

dp - dot product of two vectors

```
SYNOPSIS
    dp(x, y)

TYPES
    x, y    1-dimensional matrices of the same size

    return  depends on the nature of the elements of x and y

DESCRIPTION
    Compute the dot product of two 1-dimensional matrices.

    Let:

      x = {x0, x1, ... xn}
      y = {y0, y1, ... yn}

    Then dp(x,y) returns the result of the calculation:

      x0*y0 + x1*y1 + ... + xn*yn

EXAMPLE
    ; mat x[3] = {2,3,4}
    ; mat y[1:3] = {3,4,5}
    ; print dp(x,y)
    38

LIMITS
    none

LINK LIBRARY
    VALUE matdot(MATRIX *x, MATRIX *y)

SEE ALSO
    cp
```

epsilon - set or read the stored epsilon value

```
SYNOPSIS
    epsilon([eps])

TYPES
    eps           real number greater than 0 and less than 1

    return  real number greater than 0 and less than 1

DESCRIPTION
    Without args, epsilon() returns the current epsilon value.

    With one arg, epsilon(eps) returns the current epsilon value
    and sets the stored epsilon value to eps.

    The stored epsilon value is used as default value for eps in
    the functions appr(x, eps, rnd), sqrt(x, eps, rnd), etc.

EXAMPLE
    ; oldeps = epsilon(1e-6)
    ; print epsilon(), sqrt(2), epsilon(1e-4), sqrt(2), epsilon(oldeps)
    ; .000001 1.414214 .000001 1.4142 .0001

LIMITS
    none

LINK LIBRARY
    void setepsilon(NUMBER *eps)
    NUMBER *_epsilon_

SEE ALSO
    config
```

errcount - return or set the internal error count

```
SYNOPSIS
    errcount([num])

TYPES
    num           integer

    return  integer

DESCRIPTION
    An internal variable keeps count of the number of functions
    evaluating to an error value either internally or by a call to
    error() or newerror().

    The errcount() with no args returns the current error count.  Calling
    errcount(num) returns the current error count and resets it to num.

    If the count exceeds the current value of errmax, execution is aborted
    with a message displaying the errno for the error.

    If an error value is assigned to a variable as in:

                  infty = 1/0;

    then a function returning that variable does not contribute to
    errcount.

EXAMPLE
    ; errmax(10)
          0
    ; errcount()
          0
    ; a = 1/0; b = 2 + ""; c = error(27); d = newerror("a");
    ; print errcount(), a, errcount(), errmax();
    4 Error 10001 4 10

LIMITS
    0 <= num < 2^32

LINK LIBRARY
    none

SEE ALSO
    errmax, error, strerror, iserror, errno, newerror, errorcodes,
    stoponerror
```

errmax - return or set maximum error-count before execution stops

```
SYNOPSIS
    errmax([num])

TYPES
    num           integer

    return  integer

DESCRIPTION
    Without an argument, errmax() returns the current value of an
    internal variable errmax.  Calling errmax(num) returns this value
    but then resets its value to num.  Execution is aborted if
    evaluation of an error value if this makes errcount > errmax
    and errmax is >= 0.

    When errmax is -1, there is no limit on the number of errors.

EXAMPLE
    ; errmax(2)
      0
    ; errcount()
      0
    ; a = 1/0; b = 2 + ""; c = error(27); d = newerror("alpha");
    Error 27 caused errcount to exceed errmax

    ## Here global variables c and d were created when compiling the line
    ## but execution was aborted before the intended assignments to c and d.

    ; print c, d
    0 0

    ; errmax(-1)
      2

LIMITS
    -1 <= num <= 2147483647

LINK LIBRARY
    none

SEE ALSO
    errcount, error, strerror, iserror, errno, newerror, errorcodes,
    stoponerror
```

errno - return or set a stored error-number

```
SYNOPSIS
    errno([errnum])

TYPES
    errnum  integer, 0 <= errnum <= 32767

    return  integer

DESCRIPTION

    Whenever an operation or evaluation of function returns an error-value,
    the numerical code for that value is stored as calc_errno.

    errno() returns the current value of calc_errno.

    errno(errnum) sets calc_errno to the value errnum and returns its
    previous value.

    To detect whether an error occurs during some sequence of operations,
    one may immediately before that sequence set the stored error-number
    to zero by errno(0), and then after the operations, whether or not
    an error has occurred will be indicated by errno() being nonzero or
    zero.  If a non-zero value is returned, that value will be the code
    for the most recent error encountered.

    The default argument for the functions error() and strerror() is the
    currently stored error-number; in particular, if no error-value has
    been returned after the last errno(0), strerror() will return
    "No error".

EXAMPLE
    Assuming there is no file with name "not_a_file"
    ; errno(0)
          0
    ; errmax(errcount()+4)
          20
    ; badfile = fopen("not_a_file", "r")
    ; print errno(), error(), strerror()
    2 System error 2 No such file or directory

    ; a = 1/0
    ; print errno(), error(), strerror()
    10001 Error 10001 Division by zero

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    errmax, errcount, error, strerror, iserror, newerror, errorcodes,
    stoponerror
```

error - generate a value of specified error type

```
SYNOPSIS
    error([n])

TYPES
    n        integer, 0 <= n <= 32767; defaults to errno()

    return   null value or error value

DESCRIPTION

    If n is zero,  error(n) returns the null value.

    For positive n, error(n) returns a value of error type n.

    error(n) sets calc_errno to n so that until another error-value
    is returned by some function, errno() will return the value n.

EXAMPLE
    ; errmax(errcount()+1)
         20
    ; a = error(10009)
    ; a
         Error 10009
    ; strerror(a)
         "Bad argument for inverse"

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    errcount, errmax, errorcodes, iserror, errno, strerror, newerror,
    stoponerror
```

estr - represent some types of value by text strings

```
SYNOPSIS
    estr(x)

TYPES
    x       null, string, real or complex number, list, matrix,
            object. block, named block, error

    return  string

DESCRIPTION
    This function attempts to represent x exactly by a string s of
    ordinary text characters such that eval(s) == x.

    If x is null, estr(x) returns the string "".

    If x is a string, estr(x) returns the string in which occurrences of
    newline, tab, ", \,  etc. have been converted to \n, \t, \",
    \\, etc., '\0' to \000 or \0 according as the next character is
    or is not an octal digit, and other non-text characters to their
    escaped hex representation, e.g. char(165) becomes \xa5.

    For real x, estr(x) represebts x in fractional mode.


EXAMPLE
    ; estr("abc\0xyz\00023\n\xa5\r\n")
      ""abc\0xyz\00023\n\xa5\r\n""
    ; estr(1.67)
      "167/100"
    ; estr(mat[3] = {2, list(3,5), "abc"})
      "mat[3]={2,list(3,5),"abc""

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    str, strprintf
```

euler - Euler number

```
SYNOPSIS
    euler(n)

TYPES
    n       integer, n <= 1000000 if even

    return  integer

DESCRIPTION
    Returns the Euler number with index n, i.e. the coefficient E_n in
    the expansion

            sech(t) = Sigma E_n * t^n/n!

    When euler(n) is computed for positive even n, the values for
    n and smaller positive even indices are stored in a table so that
    a later call to euler(k) with 0 <= k <= n will be executed quickly.
    If euler(k) is called with negative k, zero is returned and the
    memory used by the table iu freed.

    Considerable runtime and memery are required for calculating
    euler(n) for large even n.

EXAMPLE
    ; for (n = 0; n <= 6; n++) print euler(n),; print;
    1 0 -1 0 5 0 -61

LIMITS
    none

LINK LIBRARY
    NUMBER *qeuler(long n)

SEE ALSO
    bernoulli, bell, catalan, comb, fact, perm
```

eval - evaluate a string

```
SYNOPSIS
    eval(str)

TYPES
    str           string

    return  any

DESCRIPTION
   For eval(str), the value of str is to be a string that could be the body
   of the definition of a function f().    This string may declare local
   variables and include keywords (while, for, ...) other than the
   reserved keywords (define, show, help, read, write, show, cd) intended
   for interactive use or for reading from a file.

   If str is the empty string "", eval(str) returns the null value.

   The call to eval(str) may return a value by explicit use of a return
   statement: "return;" returns the null value, "return expr;" returns the
   value of expr.  If execution reaches the end of str and the
   value on the execution stack is not null, eval(str) returns that value;
   otherwise eval(str) returns the most recently saved value.

   Each time eval(str) is called, a temporary function is compiled from
   the commands in str, and if there are no syntax errors, this function
   is then evaluated.  If str contains syntax errors, eval(str) displays
   the scanerror messages and returns the value error(49).

EXAMPLE
   ; str1 = "2 + 3"; print eval(str1);
   5

   ; i = 10; str2 = "local i = 0; 7; while (i++ < 5) print i^2,:;"
   ; print i, eval(str2), i
   10 1 4 9 16 25 7 10

   (The print statements in str2 return the null value, so execution of
   eval(str2) ends by returning the saved value 7.  The global variable
   i is unchanged.)

   ; eval("2 + ");
   Missing expression
      49

LIMITS
   The string str in eval(str) should not include a call to itself as in

      str = "2 + eval(str)"

   For this str, eval(str) causes an "Evaluation stack depth exceeded" error.
   Similarly, if str1 = "2 + eval(str2)", str2 should not include a call
   to eval(str1), etc.

LINK LIBRARY
   none
```

```
SEE ALSO
    command, expression, define, prompt
```

exp - exponential function

```
SYNOPSIS
    exp(x [,eps])

TYPES
    x       real or complex
    eps           nonzero real, defaults to epsilon()

    return  real or complex

DESCRIPTION
    Approximate the exponential function of x by a multiple of epsilon,
    the error having absolute value less than 0.75 * eps.
    If n is a positive integer, exp(x, 10^-n) will usually be
    correct to the n-th decimal place, which, for large positive x
    will give many significant figures.

EXAMPLE
    ; print exp(2, 1e-5), exp(2,1e-10), exp(2, 1e-15), exp(2, 1e-20)
    7.38906 7.3890560989 7.38905609893065 7.38905609893065022723

    ; print exp(30, 1e5), exp(30, 1), exp(30, 1e-10)
    10686474600000 10686474581524 10686474581524.4621469905

    ; print exp(-20, 1e-5), exp(-20, 1e-10), exp(-20, 1e-15), exp(-20, 1e-20)
    0 .0000000021 .000000002061154 .00000000206115362244

    ; print exp(1+2i, 1e-5), exp(1+2i, 1e-10)
    -1.1312+2.47173i -1.1312043838+2.471726672i

LIMITS
    x < 693093

LINK LIBRARY
    NUMBER *qexp(NUMBER *x, NUMBER *eps)
    COMPLEX *c_exp(COMPLEX *x, NUMBER *eps)

SEE ALSO
    ln, cosh, sinh, tanh
```

fact - factorial

```
SYNOPSIS
    fact(x)

TYPES
    x        int

    return   int

DESCRIPTION
    Return the factorial of a number.  Factorial is defined as:

         x! = 1 * 2 * 3 * ... * x-1 * x
         0! = 1

EXAMPLE
    ; print fact(10), fact(5), fact(2), fact(1), fact(0)
    3628800 120 2 1 1

    ; print fact(40)
    815915283247897734345611269596115894272000000000

LIMITS
    2^24 > x >= 0
    y < 2^24

LINK LIBRARY
    void zfact(NUMBER x, *ret)

SEE ALSO
    comb, perm, randperm
```

factor - smallest prime factor not exceeding specified limit

```
SYNOPSIS
    factor(n [, limit [, err]])

TYPES
    n       integer
    limit   integer with abs(limit) < 2^32, defaults to 2^32 - 1
    err             integer

    return  positive integer or err

DESCRIPTION
    This function ignores the signs of n and limit, so here we shall
    assume n and limit are both nonnegative.

    If n has a prime proper factor less than or equal to limit, then
    factor(n, limit) returns the smallest such factor.

      NOTE: A proper factor of n>1 is a factor < n.  In other words,
            for n>1 is not a proper factor of itself.  The value 1
            is a special case because 1 is a proper factor of 1.

    When every prime proper factor of n is greater than limit, 1 is
    returned.  In particular, if limit < 2, factor(n, limit) always
    returns 1.  Also, factor(n,2) returns 2 if and only if n is even
    and n > 2.

    If 1 < n < nextprime(limit)^2, then f(n, limit) == 1 <==> n is prime.
    For example, if 1 < n < 121, n is prime if and only if f(n,7) == 1.

    If limit >= 2^32, factor(n, limit) causes an error and factor(n,
    limit, err) returns the value of err.

EXAMPLE
    ; print factor(35,4), factor(35,5), factor(35), factor(-35)
    1 5 5 5

    ; print factor(2^32 + 1), factor(2^47 - 1), factor(2^59 - 1)
    641 2351 179951

LIMITS
    limit < 2^32

LINK LIBRARY
    FLAG zfactor(ZVALUE n, ZVALUE limit, ZVALUE *res)

SEE ALSO
    isprime, lfactor, nextcand, nextprime, prevcand, prevprime,
    pfact, pix, ptest
```

fcnt - count of number of times a specified integer divides an integer

```
SYNOPSIS
    fcnt(x,y)

TYPES
    x         integer
    y         integer

    return    non-negative integer

DESCRIPTION
    If x is nonzero and abs(y) > 1, fcnt(x,y) returns the greatest
    non-negative n for which y^n is a divisor of x.  In particular,
    zero is returned if x is not divisible by y.

    If x is zero or if y = -1, 0 or 1, fcnt(x,y) is defined to be zero.

EXAMPLE
    ; print fcnt(7,4), fcnt(24,4), fcnt(48,4)
    0 1 2

LIMITS
    none

LINK LIBRARY
    long zfacrem(ZVALUE x, ZVALUE y, ZVALUE *rem)

SEE ALSO
    frem, gcdrem
```

floor - floor

```
SYNOPSIS
    floor(x)

TYPES
    x       real, complex, list, matrix

    return  real or complex, list, matrix

DESCRIPTION
    For real x, floor(x) is the greatest integer not greater than x.

    For complex, floor(x) returns the real or complex number v for
    which re(v) = floor(re(x)), im(v) = floor(im(x)).

    For list or matrix x, floor(x) returns the list or matrix of the
    same structure as x for which each element t of x has been replaced
    by floor(t).

EXAMPLE
    ; print floor(27), floor(1.23), floor(-4.56), floor(7.8 - 9.1i)
    27 1 -5 7-10i

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    ceil, int
```

forall - to evaluate a function for all values of a list or matrix

```
SYNOPSIS
    forall(x, y)

TYPES
    x       list or matrix
    y       string

    return  null value

DESCRIPTION
    In forall(x,y), y is the name of a function; that function
    is performed in succession for all elements of x.  This is similar
    to modify(x, y) but x is not changed.

EXAMPLE
    ; global n = 0
    ; define s(a) {n += a;}
    ; A = list(1,2,3,4)
    ; forall(A, "s")
    ; n
          10

    ; define e(a) {if (iseven(a)) print a;}
    ; forall(A, "e")
          2
          4

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    modify
```

frac - return the fractional part of a number or of numbers in a value

```
SYNOPSIS
    frac(x)

TYPES
    If x is an object of type xx, frac(x) requires xx_frac to have been
    defined; other conditions on x and the value returned depend on
    the definition of xx_frac.

    For other x:

    x       number (real or complex), matrix

    return  number or matrix

DESCRIPTION
    If x is an integer, frac(x) returns zero.  For other real values of x,
    frac(x) returns the real number f for which x = i + f, where i is an
    integer, sgn(f) = sgn(x), and abs(f) < 1.

    If x is complex, frac(x) returns frac(re(x)) + frac(im(x))*1i.

    If x is a matrix, frac(x) returns the matrix m with the same structure
    as x in which m[[i]] = frac(x[[i]]).

EXAMPLE
    ; c = config("mode", "frac")
    ; print frac(3), frac(22/7), frac(27/7), frac(-3.125), frac(2.15 - 3.25i)
    0 1/7 6/7 -1/8 3/20-1i/4

LIMITS
    none

LINK LIBRARY
    NUMBER *qfrac(NUMBER *x)
    COMPLEX *c_frac(COMPLEX *x)
    MATRIX *matfrac(MATRIX *x)

SEE ALSO
    int, ceil, floor
```

free - free the memory used to store values of lvalues

```
SYNOPSIS
    free(a, b, ...)

TYPES
    a, b, ...     any

    return  null value

DESCRIPTION
    Those of the arguments a, b, ... that specify lvalues are assigned
    the null value, effectively freeing whatever memory is used to
    store their current values.      Other arguments are ignored.

    free(.) frees the current "old value".

EXAMPLE
    ; a = 7
    ; mat M[3] = {1, list(2,3,4), list(5,6)}
    ; print memsize(a), memsize(M)
    80 736

    ; free(a, M[1])
    ; print memsize(a), memsize(M)
    16 424

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    freeglobals, freestatics, freeredc
```

freebernoulli - free stored Bernoulli numbers

```
SYNOPSIS
    freebernoulli()

TYPES
    return  none

DESCRIPTION
    The memory used to store calculated bernoulli numbers is freed by
    freebernoulli().

EXAMPLE
    ; freebernoulli();

LIMITS
    none

LINK LIBRARY
    void qfreebern(void);

SEE ALSO
    bernoulli
```

freeeuler - free stored Euler numbers

```
SYNOPSIS
    freeeuler()

TYPES
    return  none

DESCRIPTION
    The memory used to store calculated Euler numbers is freed by
    freeeuler().

EXAMPLE
    ; freeeuler();

LIMITS
    none

LINK LIBRARY
    void qfreeeuler(void);

SEE ALSO
    euler, bernoulli, freebernoulli
```

freeglobals - free memory used for values of global variables

```
SYNOPSIS
    freeglobals()

TYPES
    return  null value

DESCRIPTION
    This function frees the memory used for the values of all global
    and not unscoped static variables by assigning null values.
    The oldvalue (.) is not freed by this function.

EXAMPLE
    ; global a = 1, b = list(2,3,4), c = mat[3]
    ; static a = 2
    ; show globals

    Name    Level    Type
    ----    -----    -----
    a        1     real = 2
    a        0     real = 1
    b        0     list
    c        0     matrix

    Number: 4
    ; freeglobals()
    ; show globals

    Name    Level    Type
    ----    -----    -----
    a        1     null
    a        0     null
    b        0     null
    c        0     null

    Number: 4

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    free, freestatics, freeredc
```

freeredc - free the memory used to store redc data

```
SYNOPSIS
    freeredc()

TYPES
    return  null value

DESCRIPTION
    This function frees the memory used for any redc data currently stored by
    calls to rcin, rcout, etc.

EXAMPLE
    ; a = rcin(10,27)
    ; b = rcin(10,15)
    ; show redc
    0     1     27
    1     2     15
    ; freeredc()
    ; show redc

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    free, freeglobals, freestatics
```

freestatics - free memory used for static variables

```
SYNOPSIS
    freestatics()

TYPES

    return  null value

DESCRIPTION
    This function frees the memory used for the values of all unscoped
    static variables by in effect assigning null values to them.  As this
    will usually have significant effects of any functions in whose
    definitions these variables have been used, it is primarily intended
    for use when these functions are being undefined or redefined..

EXAMPLE
    ; static a = 5
    ; define f(x) = a++ * x;
    f() defined
    ; global a
    ; f(1)
          5
    ; show statics

    Name    Scopes    Type
    ----    ------    -----
    a       1      0    real = 6

    Number: 1
    ; freestatics()
    ; f(1)
          Error 10005
    ; strerror(.)
          "Bad arguments for *"

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    free, freeglobals, freeredc
```

frem - remove specified integer factors from specified integer

```
SYNOPSIS
    frem(x,y)

TYPES
    x        integer
    y        integer

    return   non-negative integer

DESCRIPTION
    If x and y are not zero and n is the largest non-negative integer
    for which y^n is a divisor of x, frem(x,y) returns abs(x/y^n).
    In particular, abs(x) is returned if x is not divisible by
    y or if abs(y) = 1.  If abs(y) > 1, frem(x,y) is the greatest
    divisor of x not divisible by y.

    For all x, frem(x,0) is defined to equal abs(x).

    For all y, frem(0,y) is defined to be zero.

    For all x and y, abs(x) = frem(x,y) * abs(y) ^ fcnt(x,y).

EXAMPLE
    ; print frem(7,4), frem(24,4), frem(48,4), frem(-48,4)
    7 6 3 3

LIMITS
    none

LINK LIBRARY
    NUMBER *qfacrem(NUMBER *x, NUMBER *y);

SEE ALSO
    fcnt, gcdrem
```

```
*************
* intro
*************

What is calc?

    Calc is an interactive calculator which provides for easy large
    numeric calculations, but which also can be easily programmed
    for difficult or long calculations.    It can accept a command line
    argument, in which case it executes that single command and exits.
    Otherwise, it enters interactive mode.  In this mode, it accepts
    commands one at a time, processes them, and displays the answers.
    In the simplest case, commands are simply expressions which are
    evaluated.    For example, the following line can be input:

          3 * (4 + 1)

    and the calculator will print:

          15

    Calc as the usual collection of arithmetic operators +, -, /, *
    as well as ^ (exponentiation), % (modulus) and // (integer divide).
    For example:

          3 * 19^43 - 1

     will produce:

          29075426613099201338473141505176993450849249622191102976

    Notice that calc values can be very large.  For example:

          2^23209-1

    will print:

         402874115778988778181873329071 ... many digits ... 3779264511

    The special '.' symbol (called dot), represents the result of the
    last command expression, if any.  This is of great use when a series
    of partial results are calculated, or when the output mode is changed
    and the last result needs to be redisplayed.  For example, the above
    result can be modified by typing:

          . % (2^127-1)

    and the calculator will print:

          47385033654019111249345128555354223304

    For more complex calculations, variables can be used to save the
    intermediate results.  For example, the result of adding 7 to the
    previous result can be saved by typing:

          curds = 15
          whey = 7 + 2*curds
```

Functions can be used in expressions. There are a great number of pre-defined functions. For example, the following will calculate the factorial of the value of 'old':

```
fact(whey)
```

and the calculator prints:

```
13763753091226345046315979581580902400000000
```

The calculator also knows about complex numbers, so that typing:

```
(2+3i) * (4-3i)
cos(.)
```

will print:

```
17+6i
-55.50474777265624667147+193.92652357489279865371i
```

The calculator can calculate transcendental functions, and accept and display numbers in real or exponential format. For example, typing:

```
config("display", 70)
epsilon(1e-70)
sin(1)
```

prints:

```
0.8414709848078965066525023216302989996225630607983710656727517099919104
```

Calc can output values in terms of fractions, octal or hexadecimal. For example:

```
config("mode", "fraction"),
(17/19)^23
base(16),
(19/17)^29
```

will print:

```
19967568900859523802559065713/257829627945307727248226067259
0x9201e65bdbb801eaf403f657efcf863/0x5cd2e2a01291ffd73bee6aa7dcf7d1
```

All numbers are represented as fractions with arbitrarily large numerators and denominators which are always reduced to lowest terms. Real or exponential format numbers can be input and are converted to the equivalent fraction. Hex, binary, or octal numbers can be input by using numbers with leading '0x', '0b' or '0' characters. Complex numbers can be input using a trailing 'i', as in '2+3i'. Strings and characters are input by using single or double quotes.

Commands are statements in a C-like language, where each input line is treated as the body of a procedure. Thus the command line can contain variable declarations, expressions, labels, conditional tests, and loops. Assignments to any variable name will automatically define that name as a global variable. The other important thing to know is that all non-assignment expressions which are evaluated are automatically printed. Thus, you can evaluate an expression's value by simply typing it in.

Many useful built-in mathematical functions are available. Use the:

```
help builtin
```

command to list them.

You can also define your own functions by using the 'define' keyword, followed by a function declaration very similar to C.

```
define f2(n)
{
      local       ans;

      ans = 1;
      while (n > 1)
            ans *= (n -= 2);
      return ans;
}
```

Thus the input:

```
f2(79)
```

will produce;

```
1009847364737869270905302433221592504062302663202724609375
```

Functions which only need to return a simple expression can be defined using an equals sign, as in the example:

```
define sc(a,b) = a^3 + b^3
```

Thus the input:

```
sc(31, 61)
```

will produce;

```
256772
```

Variables in functions can be defined as either 'global', 'local', or 'static'. Global variables are common to all functions and the command line, whereas local variables are unique to each function level, and are destroyed when the function returns. Static variables are scoped within single input files, or within functions, and are never destroyed. Variables are not typed at definition time, but dynamically change as they are used.

For more information about the calc language and features, try:

```
help overview
```

In particular, check out the other help functions listed in the overview help file.

gcd - greatest common divisor of a set of rational numbers

```
SYNOPSIS
    gcd(x1, x2, ...)

TYPES
    x1, x2, ... rational number

    return  rational number

DESCRIPTION
    If at least one xi is nonzero, gcd(x1, x2, ...) is the
    greatest positive number g for which each xi is a multiple of g.
    If all xi are zero, the gcd is zero.

EXAMPLE
    ; print gcd(12, -24, 30), gcd(9/10, 11/5, 4/25), gcd(0,0,0,0,0)
    6 .02 0

LIMITS
    The number of arguments may not to exceed 1024.

LINK LIBRARY
    NUMBER *qgcd(NUMBER *x1, NUMBER *x2)

SEE ALSO
    lcm
```

gcdrem - result of removing factors of integer common to a specified integer

```
SYNOPSIS
    gcdrem(x, y)

TYPES
    x       integer
    y       integer

    return  non-negative integer

DESCRIPTION

    If x and y are not zero, gcdrem(x, y) returns the greatest integer
    divisor d of x relatively prime to y, i.e. for which gcd(d,y) = 1.
    In particular, gcdrem(x,y) = abs(x) if x and y are relatively
    prime.

    For all x, gcdrem(x, 0) = 1.

    For all nonzero y, gcdrem(0, y) = 0.

PROPERTIES
    gcdrem(x,y) = gcd(abs(x), abs(y)).

    If x is not zero, gcdrem(x,y) = gcdrem(x, gcd(x,y)) = gcdrem(x, y % x).

    For fixed nonzero x, gcdrem(x,y) is periodic with period abs(x).

    gcdrem(x,y) = 1 if and only if every prime divisor of x
      is a divisor of y.

    If x is not zero, gcdrem(x,y) == abs(x) if and only if gcd(x,y) = 1.

    If y is not zero and p_1, p_2, ..., p_k are the prime divisors of y,

      gcdrem(x,y) = frem(...(frem(frem(x,p_1),p_2)...,p_k)

EXAMPLE
    ; print gcdrem(6,15), gcdrem(15,6), gcdrem(72,6), gcdrem(6,72)
    2 5 1 1

    ; print gcdrem(630,6), gcdrem(6,630)
    35 1

LIMITS
    none

LINK LIBRARY
    NUMBER *qgcdrem(NUMBER *x, NUMBER *y)

SEE ALSO
    gcd, frem, isrel
```

gd - gudermannian function

```
SYNOPSIS
    gd(z [,eps])

TYPES
    z       number (real or complex)
    eps           nonzero real, defaults to epsilon()

    return  number or "Log of zero or infinity" error value

DESCRIPTION
    Calculate the gudermannian of z to a nultiple of eps with errors in
    real and imaginary parts less in absolute value than .75 * eps,
    or return an error value if z is close to one of the branch points
    at odd multiples of (pi/2) * i.

    gd(z) is usually defined initially for real z by one of the formulae

            gd(z) = 2 * atan(exp(z)) - pi/2

                  = 2 * atan(tanh(z/2))

                  = atan(sinh(z)),

    or as the integral from 0 to z of (1/cosh(t))dt.  For complex z, the
    principal branch, approximated by gd(z, eps), has the cut:
    re(z) = 0, abs(im(z)) >= pi/2; on the cut calc takes gd(z) to be
    the limit as z is approached from the right or left according as
    im(z) > or < 0.

    If z = x + y*i and abs(y) < pi/2, gd(z) is given by

      gd(z) = atan(sinh(x)/cos(y)) + i * atanh(sin(y)/cosh(x)).

EXAMPLE
    ; print gd(1, 1e-5), gd(1, 1e-10), gd(1, 1e-15)
    .86577 .8657694832 .865769483239659

    ; print gd(2+1i, 1e-5), gd(2+1i, 1e-10)
    1.42291+.22751i 1.4229114625+.2275106584i

LIMITS
    none

LINK LIBRARY
    COMPLEX *c_gd(COMPLEX *x, NUMBER *eps)

SEE ALSO
    agd, exp, ln, sin, sinh, etc.
```

hash - return the calc hash value

```
SYNOPSIS
    hash(x_1 [, x_2, x_3, ...])

TYPES
    x_1, x_1, ... any

    return        integer v, 0 <= v < 2^32

DESCRIPTION
    Returns a hash value for one or more values of arbitrary types.

    The calc hash value is based on the core Fowler/Noll/Vo hash
    known as FNV-1.  The return value, however, cannot be used
    as an FNV hash value because calc's internal function also
    takes into account more abstract concepts such as data types.

    See:

       http://www.isthe.com/chongo/tech/comp/fnv/

    information about the Fowler/Noll/Vo (FNV) hash.

EXAMPLE
    ; a = isqrt(2e1000); s = "xyz";
    ; hash(a,s)
          2378490456

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    ishash, sha1
```

head - create a list of specified size from the head of a list

```
SYNOPSIS
    head(x, y)

TYPES
    x       list
    y       int

    return  list

DESCRIPTION
    If 0 <= y <= size(x), head(x,y) returns a list of size y whose
    elements in succession have values x[[0]]. x[[1]], ..., x[[y - 1]].

    If y > size(x), head(x,y) is a copy of x.

    If -size(x) < y < 0, head(x,y) returns a list of size (size(x) + y)
    whose elements in succession have values x[[0]]. x[[1]], ...,
    i.e. a copy of x from which the last -y members have been deleted.

    If y <= -size(x), head(x,y) returns a list with no members.

    For any integer y, x == join(head(x,y), tail(x,-y)).

EXAMPLE
    ; A = list(2, 3, 5, 7, 11)
    ; head(A, 2)

    list (2 members, 2 nonzero):
        [[0]] = 2
        [[1]] = 3

    ; head(A, -2)

    list (3 members, 3 nonzero):
        [[0]] = 2
        [[1]] = 3
        [[2]] = 5

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    tail, segment
```

highbit - index of highest bit in binary representation of integer

```
SYNOPSIS
    highbit(x)

TYPES
    x       nonzero integer

    return  integer

DESCRIPTION
    If x is a nonzero integer, highbit(x) returns the index of the
    highest bit in the binary representation of abs(x).      Equivalently,
    highbit(x) = n if 2^n <= abs(x) < 2^(n + 1);  the binary
    representation of x then has n + 1 digits.

EXAMPLE
    ; print highbit(2), highbit(3), highbit(4), highbit(-15), highbit(2^27)
    1 1 2 3 27

LIMITS
    none

LINK LIBRARY
    LEN zhighbit(ZVALUE x);

SEE ALSO
    lowbit, digits
```

hmean - harmonic mean of a number of values

```
SYNOPSIS
    hmean(x_1, x_2, ...)

TYPES
    x_1, ...       arithmetic or list

    return  determined by types of arguments, or null

DESCRIPTION
    The null value is returned if there are no arguments.

    If there are n non-list arguments x_1, x_2, ... and the
    required operations are defined, hmean(x_1, x_2, ...) returns the
    value of:

      n/(inverse(x_1) + inverse(x_2) + ... + inverse(x_n)).

    If an argument x_i is a list as defined by list(y_1, ..., y_m)
    this is treated as if in (x_1, x_2, ...), x_i is replaced by
    y_1, ..., y_m.


EXAMPLE
    ; c = config("mode", "frac")
    ; print hmean(1), hmean(1,2), hmean(1,2,3), hmean(1,2,3,4), hmean(1,2,0,3)
    1 4/3 18/11 48/25 0

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    avg
```

hnrmod - compute mod h * 2^n +r

```
SYNOPSIS
    hnrmod(v, h, n, r)

TYPES
    v       integer
    h       integer
    n       integer
    r       integer

    return  integer

DESCRIPTION
    Compute the value:

      v % (h * 2^n +r)

    where:

      h > 0
      n > 0
      r == -1, 0 or 1

    This builtin in faster than the standard mod in that is makes use
    of shifts and additions when h == 1.  When h > 1, a division by h
    is also needed.

EXAMPLE
    ; print hnrmod(2^177-1, 1, 177, -1), hnrmod(10^40, 17, 51, 1)
    0 33827019788296445

LIMITS
    h > 0
    2^31 > n > 0
    r == -1, 0 or 1

LINK LIBRARY
    void zhnrmod(ZVALUE v, ZVALUE h, ZVALUE zn, ZVALUE zr, ZVALUE *res)

SEE ALSO
    mod
```

hypot - hypotenuse of a right-angled triangle given the other sides

```
SYNOPSIS
    hypot(x, y [,eps])

TYPES
    x, y    real
    eps           nonzero real

    return  real

DESCRIPTION
    Returns sqrt(x^2 + y^2) to the nearest multiple of eps.
    The default value for eps is epsilon().

EXAMPLE
    ; print hypot(3, 4, 1e-6), hypot(2, -3, 1e-6)
    5 3.605551

LIMITS
    none

LINK LIBRARY
    NUMBER *qhypot(NUMBER *q1, *q2, *epsilon)

SEE ALSO
    ltol
```

ilog - floor of logarithm to specified integer base

```
SYNOPSIS
    ilog(x, b)

TYPES
    x       nonzero real
    b       integer greater than 1

    return  integer

DESCRIPTION
    Returns the greatest integer n for which b^n <= abs(x).

EXAMPLE
    ; print ilog(2, 3), ilog(8, 3), ilog(8.9, 3), ilog(1/8, 3)
    0 1 1 -2

LIMITS
    x > 0
    b > 1

LINK LIBRARY
    long zlog(ZVALUE x, ZVALUE b)

SEE ALSO
    ilog2, ilog10
```

ilog10 - floor of logarithm to base 10

```
SYNOPSIS
    ilog10(x)

TYPES
    x       nonzero real

    return  integer

DESCRIPTION
    Returns the greatest integer n for which 10^n <= x.

EXAMPLE
    ; print ilog10(7), ilog10(77.7), ilog10(777), ilog10(.00777), ilog10(-1e27)
    0 1 2 -3 27

LIMITS
    none

LINK LIBRARY
    long qilog10(NUMBER *q)

SEE ALSO
    ilog2, ilog
```

ilog2 - floor of logarithm to base 2

```
SYNOPSIS
    ilog2(x)

TYPES
    x       nonzero real

    return  integer

DESCRIPTION
    Returns the greatest integer n for which 2^n <= abs(x).

EXAMPLE
    ; print ilog2(1), ilog2(2), ilog2(3), ilog2(4), ilog(1/15)
    0 1 1 2 -4

LIMITS
    none

LINK LIBRARY
    long qilog2(NUMBER *q)

SEE ALSO
    ilog10, ilog
```

im - imaginary part of a real or complex number

```
SYNOPSIS
    im(x)

TYPES
    x       real or complex

    return  real

DESCRIPTION
    If x = u + v * 1i where u and v are real, im(x) returns v.

EXAMPLE
    ; print im(2), im(2 + 3i), im(-4.25 - 7i)
    0 3 -7

LIMITS
    none

LINK LIBRARY
    COMPLEX *c_imag(COMPLEX *x)

SEE ALSO
    re
```

indices - indices for specified matrix or association element

```
SYNOPSIS
    indices(V, index)

TYPES
    V       matrix or association
    index   integer

    return  list with up to 4 elements

DESCRIPTION
    For 0 <= index < size(V), indices(V, index) returns list(i_0, i_1, ...)
    for which V[i_0, i_1, ...] is the same lvalue as V[[index]].

    For other values of index, a null value is returned.

    This function can be useful for determining those elements for which
    the indices satisfy some condition.    This is particularly so for
    associations since these have no simple relation between the
    double-bracket index and the single-bracket indices, which may be
    non-integer numbers or strings or other types of value.  The
    information provided by indices() is often required after the use
    of search() or rsearch() which, when successful, return the
    double-bracket index of the item found.

EXAMPLE
    ; mat M[2,3,1:5]

    ; indices(M, 11)
    list (3 elements, 2 nonzero):
          [[0]] = 0
          [[1]] = 2
          [[2]] = 2

    ; A = assoc();

    ; A["cat", "dog"] = "fight";
    ; A[2,3,5,7] = "primes";
    ; A["square", 3] = 9

    ; indices(A, search(A, "primes"))
    list (4 elements, 4 nonzero):
          [[0]] = 2
          [[1]] = 3
          [[2]] = 5
          [[3]] = 7

LIMITS
    abs(index) < 2^31

LINK LIBRARY
    LIST* associndices(ASSOC *ap, long index)
    LIST* matindices(MATRIX *mp, long index)

SEE ALSO
    assoc, mat
```

inputlevel - current input level

```
SYNOPSIS
    inputlevel()

TYPES
    return  nonnegative integer

DESCRIPTION
    This function returns the input level at which it is called.
    When calc starts, it is at level zero.  The level is increased
    by 1 each time execution starts of a read file command or a call to
    eval(S) for some expression S which evaluates to a string.    It
    decreases by 1 when a file being read reaches EOF or a string
    being eval-ed reaches '\0', or earlier if a quit statement is
    encountered at top calculation-level in the flle or string.    It
    decreases to zero if an abort statement is encountered at any
    function-level in the file or string.  If a quit or abort
    statement is encountered at top calculation-level at top input-level,
    calc is exited.

    Zero input level is also called top input level; greater values
    of inputlevel() indicate reading at greater depths.

EXAMPLE
    n/a

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    read, eval, quit, abort, calclevel
```

insert - insert one or more elements into a list at a given position

```
SYNOPSIS
    insert(x, y, z_0, z_1, ...)

TYPES
    x       lvalue whose value is a list
    y       int
    z_0, ...      any

    return  null value

DESCRIPTION
     If after evaluation of z_0, z_1, ..., x is a list with contents
     (x_0, x_1, ..., x_y-1, x_y, ..., x_n-1), then after insert(),
     x has contents (x_0, x_1, ..., x_y-1, z_0, z_1, ..., x_y, ..., x_n-1),
     i.e. z_0, z_1, ... are inserted in order immediately before the
     element with index y (so that z_0 is now x[[y]]), or if y = n,
     after the last element x_n-1.  An error occurs if y > n.

EXAMPLE
    ; A = list(2,3,4)
    ; print A

    list (3 elements, 3 nonzero):
      [[0]] = 2
      [[1]] = 3
      [[2]] = 4

    ; insert(A, 1, 5, 6)
    ; print A

    list (5 elements, 5 nonzero):
      [[0]] = 1
      [[1]] = 5
      [[2]] = 6
      [[3]] = 3
      [[4]] = 4

    ; insert(A, 2, remove(A))
    ; print A

    list (5 elements, 5 nonzero):
      [[0]] = 1
      [[1]] = 5
      [[2]] = 4
      [[3]] = 6
      [[4]] = 3

LIMITS
    insert() can have at most 100 arguments
    o <= y <= size(x)

LINK LIBRARY
    none

SEE ALSO
```

append, delete, islist, pop, push, remove, rsearch, search, select, size

int - return the integer part of a number or of numbers in a value

```
SYNOPSIS
    int(x)

TYPES
    If x is an object of type xx, int(x) requires xx_int to have been
    defined; other conditions on x and the value returned depend on
    the definition of xx_int.

    For other x:

    x       number (real or complex), matrix

    return  number or matrix

DESCRIPTION
    If x is an integer, int(x) returns x.  For other real values of x,
    int(x) returns the value of i for which x = i + f, where i is an
    integer, sgn(f) = sgn(x) and abs(f) < 1.

    If x is complex, int(x) returns int(re(x)) + int(im(x))*1i.

    If x is a matrix, int(x) returns the matrix m with the same structure
    as x in which m[[i]] = int(x[[i]]).

EXAMPLE
    ; print int(3), int(22/7), int(27/7), int(-3.125), int(2.15 - 3.25i)
    3 3 3 -3 2-3i

LIMITS
    none

LINK LIBRARY
    NUMBER *qint(NUMBER *x)
    COMPLEX *c_int(COMPLEX *x)
    MATRIX *matint(MATRIX *x)

SEE ALSO
    frac, ceil, floor, quo
```

Interrupts

While a calculation is in progress, you can generate the SIGINT signal, and the calculator will catch it. At appropriate points within a calculation, the calculator will check that the signal has been given, and will abort the calculation cleanly. If the calculator is in the middle of a large calculation, it might be a while before the interrupt has an effect.

You can generate the SIGINT signal multiple times if necessary, and each time the calculator will abort the calculation at a more risky place within the calculation. Each new interrupt prints a message of the form:

```
[Abort level n]
```

where n ranges from 1 to 3. For n equal to 1, the calculator will abort calculations at the next statement boundary specified by an ABORT opcode as described below. For n equal to 2, the calculator will abort calculations at the next opcode boundary. For n equal to 3, the calculator will abort calculations at the next attempt to allocate memory for the result of an integer arithmetic operation; this level may be appropriate for stopping a builtin operation like inversion of a large matrix.

If a final interrupt is given when n is 3, the calculator will immediately abort the current calculation and longjmp back to the top level command level. Doing this may result in corrupted data structures and unpredictable future behavior, and so should only be done as a last resort. You are advised to quit the calculator after this has been done.

ABORT opcodes

If config("trace") & 2 is zero, ABORT opcodes are introduced at various places in the opcodes for evaluation of command lines and functions defined by "define ... { ... }" commands. In the following, config("trace") has been set equal to 8 so that opcodes are displayed when a function is defined. The function f(x) evaluates x + (x - 1) + (x - 2) + ... until a zero term is encountered. If f() is called with a negative or fractional x, the calculation is never completed and to stop it, an interruption (on many systems, by ctrl-C) will be necessary.

```
; config("trace", 8),
; define f(x) {local s; while (x) {s += x--} return s}
0: DEBUG line 2
2: PARAMADDR x
4: JUMPZ 19
6: DEBUG line 2
8: LOCALADDR s
10: DUPLICATE
11: PARAMADDR x
13: POSTDEC
14: POP
15: ADD
16: ASSIGNPOP
```

```
  17: JUMP 2
  19: DEBUG line 2
  21: LOCALADDR s
  23: RETURN
  f(x) defined
```

(The line number following DEBUG refers to the line in the file from which the definition is read.)   If an attempt is made to evaluate f(-1), the effect of the DEBUG at opcode 6 ensures that a single SIGINT will stop the calculation at a start of {s += x--} loop. In interactive mode, with ^C indicating input of ctrl-C, the displayed output is as in:

```
  ; f(-1)
  ^C
  [Abort level 1]
  "f": line 2: Calculation aborted at statement boundary
```

The DEBUG opcodes are disabled by nonzero config("trace") & 2. Changing config("trace") to achieve this, and defining g(x) with the same definition as for f(x) gives:

```
  ; define g(x) {local s; while (x) {s += x--} return s}
  0: PARAMADDR x
  2: JUMPZ 15
  4: LOCALADDR s
  6: DUPLICATE
  7: PARAMADDR x
  9: POSTDEC
  10: POP
  11: ADD
  12: ASSIGNPOP
  13: JUMP 0
  15: LOCALADDR s
  17: RETURN
  g(x) defined
```

If g(-1) is called, two interrupts are necessary, as in:

```
  ; g(-1)
  ^C
  [Abort level 1]
  ^C
  [Abort level 2]
  "g": Calculation aborted in opcode
```

What is calc?

Calc is an interactive calculator which provides for easy large numeric calculations, but which also can be easily programmed for difficult or long calculations. It can accept a command line argument, in which case it executes that single command and exits. Otherwise, it enters interactive mode. In this mode, it accepts commands one at a time, processes them, and displays the answers. In the simplest case, commands are simply expressions which are evaluated. For example, the following line can be input:

```
3 * (4 + 1)
```

and the calculator will print:

```
15
```

Calc as the usual collection of arithmetic operators +, -, /, * as well as ^ (exponentiation), % (modulus) and // (integer divide). For example:

```
3 * 19^43 - 1
```

will produce:

```
29075426613099201338473141505176993450849249622191102976
```

Notice that calc values can be very large. For example:

```
2^23209-1
```

will print:

```
402874115778988778181873329071 ... many digits ... 3779264511
```

The special '.' symbol (called dot), represents the result of the last command expression, if any. This is of great use when a series of partial results are calculated, or when the output mode is changed and the last result needs to be redisplayed. For example, the above result can be modified by typing:

```
. % (2^127-1)
```

and the calculator will print:

```
47385033654019111249345128555354223304
```

For more complex calculations, variables can be used to save the intermediate results. For example, the result of adding 7 to the previous result can be saved by typing:

```
curds = 15
whey = 7 + 2*curds
```

Functions can be used in expressions. There are a great number of pre-defined functions. For example, the following will calculate the factorial of the value of 'old':

```
fact(whey)
```

and the calculator prints:

```
13763753091226345046315979581580902400000000
```

The calculator also knows about complex numbers, so that typing:

```
(2+3i) * (4-3i)
cos(.)
```

will print:

```
17+6i
-55.50474777265624667147+193.9265235748927986537i
```

The calculator can calculate transcendental functions, and accept and display numbers in real or exponential format. For example, typing:

```
config("display", 70)
epsilon(1e-70)
sin(1)
```

prints:

```
0.8414709848078965066525023216302989996225630607983710656727517099919104
```

Calc can output values in terms of fractions, octal or hexadecimal. For example:

```
config("mode", "fraction"),
(17/19)^23
base(16),
(19/17)^29
```

will print:

```
19967568900859523802559065713/257829627945307727248226067259
0x9201e65bdbb801eaf403f657efcf863/0x5cd2e2a01291ffd73bee6aa7dcf7d1
```

All numbers are represented as fractions with arbitrarily large numerators and denominators which are always reduced to lowest terms. Real or exponential format numbers can be input and are converted to the equivalent fraction. Hex, binary, or octal numbers can be input by using numbers with leading '0x', '0b' or '0' characters. Complex numbers can be input using a trailing 'i', as in '2+3i'. Strings and characters are input by using single or double quotes.

Commands are statements in a C-like language, where each input line is treated as the body of a procedure. Thus the command line can contain variable declarations, expressions, labels, conditional tests, and loops. Assignments to any variable name will automatically define that name as a global variable. The other important thing to know is that all non-assignment expressions which are evaluated are automatically printed. Thus, you can evaluate an expression's value by simply typing it in.

Many useful built-in mathematical functions are available. Use the:

```
help builtin
```

command to list them.

You can also define your own functions by using the 'define' keyword, followed by a function declaration very similar to C.

```
define f2(n)
{
      local       ans;

      ans = 1;
      while (n > 1)
            ans *= (n -= 2);
      return ans;
}
```

Thus the input:

```
f2(79)
```

will produce;

```
1009847364737869270905302433221592504062302663202724609375
```

Functions which only need to return a simple expression can be defined using an equals sign, as in the example:

```
define sc(a,b) = a^3 + b^3
```

Thus the input:

```
sc(31, 61)
```

will produce;

```
256772
```

Variables in functions can be defined as either 'global', 'local', or 'static'. Global variables are common to all functions and the command line, whereas local variables are unique to each function level, and are destroyed when the function returns. Static variables are scoped within single input files, or within functions, and are never destroyed. Variables are not typed at definition time, but dynamically change as they are used.

For more information about the calc language and features, try:

```
help overview
```

In particular, check out the other help functions listed in the overview help file.

inverse - inverse of value

```
SYNOPSIS
    inverse(x)

TYPES
    If x is an object of type xx, the function xx_inv has to have
    been defined; any conditions on x and the nature of the returned
    value will depend on the definition of xx_inv.

    For non-object x:

    x       nonzero number (real or complex) or nonsingular matrix

    return  number or matrix

DESCRIPTION
    For real or complex x, inverse(x) returns the value of 1/x.

    If x is a nonsingular n x n matrix and its elements are numbers or
    objects for which the required arithmetic operations are defined,
    inverse(x) returns the matrix m for which m * x = x * m = the unit
    n x n matrix.  The inverse m will have the same index limits as x.

EXAMPLE
    ; print inverse(5/4), inverse(-2/7), inverse(3 + 4i)
    .8 -3.5 .12-.16i

    ; mat A[2,2] = {2,3,5,7}
    ; print inverse(A)

    mat [2,2] (4 elements, 4 nonzero):
      [0,0] = -7
      [0,1] = 3
      [1,0] = 5
      [1,1] = -2

LIMITS
    none

LINK LIBRARY
    void invertvalue(VALUE *x, VALUE *vres)
    NUMBER *qinv(NUMBER *x)
    COMPLEX *c_inv(COMPLEX *x)
    MATRIX *matinv(MATRIX *x)

SEE ALSO
```

iroot - integer part of specified root

```
SYNOPSIS
    iroot(x, n)

TYPES
    x       nonnegative real
    n       positive integer

    return  nonnegative real

DESCRIPTION
    Return the greatest integer v for which v^n <= x.

EXAMPLE
    ; print iroot(100,3), iroot(274,3), iroot(1,9), iroot(pi()^8,5)
    4 6 1 6

LIMITS
    n > 0

LINK LIBRARY
    NUMBER *qiroot(NUMBER *x, NUMBER* n)

SEE ALSO
    isqrt, sqrt
```

isassoc - whether a value is an association.

```
SYNOPSIS
    isassoc(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is an association.  This function will return 1 if x is
    an association, 0 otherwise.

EXAMPLE
    ; a = assoc()
    ; print isassoc(a), isassoc(1)
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    assoc,
    isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isblk - whether or not a value is a block

```
SYNOPSIS
    isblk(val)

TYPES
    val           any

    return  0, 1, or 2

DESCRIPTION
    isblk(val) returns 1 if val is an unnamed block, 2 if val is a
    named block, 0 otherwise.

    Note that a named block B retains its name after its data block is
    freed by rmblk(B).  That a named block B has null data block may be
    tested using sizeof(B);  this returns 0 if and only if the memory
    has been freed.

EXAMPLE
    ; A = blk()
    ; isblk(A)
      1

    ; B = blk("beta")
    ; isblk(B)
      2

    ; isblk(3)
      0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    blk, blocks, blkfree,
    isassoc, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isconfig - whether a value is a configuration state

```
SYNOPSIS
    isconfig(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a configuration state.  This function will return
    1 if x is a file, 0 otherwise.

EXAMPLE
    ; a = config("all")
    ; print isconfig(a), isconfig(0);
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    config,
    isassoc, isblk, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isdefined - whether a string names a defined function

```
SYNOPSIS
    isdefined(str)

TYPES
    str           string

    return  0, 1, or 2

DESCRIPTION
    isdefined(str) returns 1 if str is the name of a builtin function,
    2 if str is the name of a user-defined function, 0 otherwise.

EXAMPLE
    ; isdefined("abs")
      1

    ; isdefined("fun")
      0

    ; define fun() { }
    fun() defined

    ; isdefined("fun")
      2

    ; undefine fun
    ; isdefined("fun")
      0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    define, undefine,
    isassoc, isblk, isconfig, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

iserror - test whether a value is an error value

```
SYNOPSIS
    iserror(x)

TYPES
    x       any

    return  zero or positive integer < 32768

DESCRIPTION
    If x is not an error value, zero is returned.
    If x is an error value, iserror(x) returns its error type.

EXAMPLE
    ; a = error(99)
    print iserror(a), iserror(2 + a), iserror(2 + "a"), iserror(2 + 3)
    99 99 3 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    error, errorcodes, stoponerror,
    isassoc, isblk, isconfig, isdefined, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

iseven - whether a value is an even integer

```
SYNOPSIS
    iseven(x)

TYPES
    x        any, &any

    return   int

DESCRIPTION
    Determine if x is an even integer.    This function will return 1 if x is
    even integer, 0 otherwise.

EXAMPLE
    ; print iseven(2.0), iseven(1), iseven("0")
    1 0 0

    ; print iseven(2i), iseven(1e20), iseven(1/3)
    0 1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

ishash - whether a value is a hash state

```
SYNOPSIS
    ishash(x)

TYPES
    x       any

    return  integer

DESCRIPTION
    The value returned by ishash(x) is:

      0 if x is not a hash state,
      2 if x is a sha1 hash state,

EXAMPLE
    ; a = 1; b = sha1(0); c = sha1(a)
    ; print ishash(0), ishash(a), ishash(b), ishash(c), ishash(sha1(a))
    0 0 2 2 2

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, israndom, isrand, isreal, isrel,
    issimple, issq, isstr, istype, sha1
```

isident - returns 1 if matrix is an identity matrix

```
SYNOPSIS
    isident(m)

TYPES
    m        any

    return   int

DESCRIPTION
    This function returns 1 if m is an 2 dimensional identity matrix,
    0 otherwise.

EXAMPLE
    ; mat x[3,3] = {1,0,0,0,1,0,0,0,1};
    ; isident(x)
          1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mat, matdim, matfill, matmax, matmin, matsum, mattrans,
    isassoc, , isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isint - whether a value is an integer

```
SYNOPSIS
    isint(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is an integer.  This function will return 1 if x is
    integer, 0 otherwise.

EXAMPLE
    ; print isint(2.0), isint(1), isint("0")
    1 1 0

    ; print isint(2i), isint(1e20), isint(1/3)
    0 1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    int,
    isassoc, , isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

islist - whether a value is a list

```
SYNOPSIS
    islist(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a list.  This function will return 1 if x is
    a list, 0 otherwise.

EXAMPLE
    ; lst = list(2,3,4)
    ; print islist(lst), islist(1)
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, pop, push, remove, rsearch, search,
     select, size,

     isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
     ishash, isident, isint, ismat, ismult, isnull, isnum, isobj,
     isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
     issimple, issq, isstr, istype
```

ismat - whether a value is a matrix

```
SYNOPSIS
    ismat(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a matrix.      This function will return 1 if x is
    a matrix, 0 otherwise.

EXAMPLE
    ; mat a[2]
    ; print ismat(a), ismat(1)
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mat, matdim, matfill, matmax, matmin, matsum, mattrans,
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, israndom, isrand, isreal, isrel,
    issimple, issq, isstr, istype
```

ismult - whether a value is a multiple of another

```
SYNOPSIS
    ismult(x, y)

TYPES
    x       real
    y       real

    return  int

DESCRIPTION
    Determine if x exactly divides y.  If there exists an integer k
    such that:

      x == y * k

    then return 1, otherwise return 0.

EXAMPLE
    ; print ismult(6, 2), ismult(2, 6), ismult(7.5, 2.5)
    1 0 1

    ; print ismult(4^67, 2^59), ismult(13, 4/67), ismult(13, 7/56)
    1 0 1

LIMITS
    none

LINK LIBRARY
    BOOL qdivides(NUMBER *x, *y)
    BOOL zdivides(ZVALUE x, y)

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isnull - whether a value is a null value

```
SYNOPSIS
    isnull(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a null value.  This function will return 1 if x is
    a null value, 0 otherwise.

EXAMPLE
    ; mat a[2]
    ; print isnull(a), isnull(1)
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isnum - whether a value is a numeric value

```
SYNOPSIS
    isnum(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a numeric value.    This function will return 1 if x
    is a a numeric value, 0 otherwise.

EXAMPLE
    ; print isnum(2.0), isnum(1), isnum("0")
    1 1 0

    ; print isnum(2i), isnum(1e20), isnum(1/3)
    1 1 1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isobj - whether a value is an object

```
SYNOPSIS
    isobj(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is an object.  This function will return 1 if x is
    an object, 0 otherwise.

EXAMPLE
    ; obj surd {a, b} a;
    ; print isobj(a), isobj(1)
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    obj,
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isobjtype - whether a string names an object type

```
SYNOPSIS
    isobjtype(str)

TYPES
    str           string

    return  0 or 1

DESCRIPTION
    isobjtype(str) returns 1 or 0 according as an object type with name
    str has been defined or not defined.

EXAMPLE
    ; isobjtype("xy")
      0

    ; obj xy {x, y}
    ; isobjtype("xy")
      1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    obj,
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isodd - whether a value is an odd integer

```
SYNOPSIS
    isodd(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is an odd integer.  This function will return 1 if x is
    odd integer, 0 otherwise.

EXAMPLE
    ; print isodd(2.0), isodd(1), isodd("1")
    0 1 0

    ; print isodd(2i), isodd(1e20+1), isodd(1/3)
    0 1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isprime, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

isprime - whether a small integer is prime

```
SYNOPSIS
    isprime(x [,err])

TYPES
    x       int
    err           int

    return  int

DESCRIPTION
    Determine if x is is a small prime.    This function will return
    1 if x is a small prime.  If x is even, this function will
    return 0.  If x is negative or a small composite (non-prime),
    0 will be returned.

    If x is a large positive odd value and the err argument is
    given, this function return err.  If x is a large positive odd
    value and the err argument is not given, an error will be
    generated.

    Note that normally this function returns the integer 0 or 1.
    If err is given and x is a large positive odd value, then err
    will be returned.

EXAMPLE
    ; print isprime(-3), isprime(1), isprime(2)
    0 0 1

    ; print isprime(21701), isprime(1234577), isprime(1234579)
    1 1 0

    ; print isprime(2^31-9), isprime(2^31-1), isprime(2^31+11)
    0 1 1

    ; print isprime(2^32+1, -1), isprime(3^99, 2), isprime(4^99, 2)
    -1 2 0

LIMITS
    err not given  and  (y is even  or    y < 2^32)

LINK LIBRARY
    FLAG zisprime(ZVALUE x)   (return 1 if prime, 0 not prime, -1 if >= 2^32)

SEE ALSO
    factor, lfactor, nextcand, nextprime, prevcand, prevprime,
    pfact, pix, ptest
```

isptr - whether a value is a pointer

```
SYNOPSIS
    isptr(x)

TYPES
    x        any

    return  0, 1, 2, 3, or 4

DESCRIPTION
    isptr(x) returns:

            0 if x is a not pointer
            1 if x is an octet-pointer
            2 if x is a value-pointer
            3 if x is a string-pointer
            4 if x is a number-pointer

     Pointers are initially defined by using the addreess (&) operator
     with an "addressable" value; currently, these are octets, lvalues,
     strings and real numbers.

EXAMPLE
    ; a = "abc", b = 3, B = blk()
    ; p1 = &B[1]
    ; p2 = &a
    ; p3 = &*a
    ; p4 = &*b
    ; print isptr(a), isptr(p1), isptr(p2), isptr(p3), isptr(p4)
    0 1 2 3 4

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isnum, isstr, isblk, isoctet
```

isqrt - integer part of square root

```
SYNOPSIS
    isqrt(x)

TYPES
    x       nonnegative real

    return  nonnegative real

DESCRIPTION
    Return the greatest integer n for which n^2 <= x.

EXAMPLE
    ; print isqrt(8.5), isqrt(200), isqrt(2e6), isqrt(2e56)
    2 14 1414 14142135623730950488016887242

LIMITS
    x > 0

LINK LIBRARY
    NUMBER *qisqrt(NUMBER *x)

SEE ALSO
    sqrt, iroot
```

isrand - whether a value is an additive 55 state

```
SYNOPSIS
    isrand(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is an additive 55 pseudo-random number generator state.
    This function will return 1 if x is a file, 0 otherwise.

EXAMPLE
    ; a = srand(0)
    ; print isrand(a), isrand(0);
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    seed, rand, srand, randbit,
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, israndom, isreal, isrel,
    issimple, issq, isstr, istype
```

israndom - whether a value is a Blum generator state

```
SYNOPSIS
    israndom(x)

TYPES
    x        any, &any

    return   int

DESCRIPTION
    Determine if x is a Blum-Blum-Shub pseudo-random number generator state.
    This function will return 1 if x is a file, 0 otherwise.

EXAMPLE
    ; a = srandom(0)
    ; print israndom(a), israndom(0);
    1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    seed, random, srandom, randombit,
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, isreal, isrel,
    issimple, issq, isstr, istype
```

isreal - whether a value is a real value

```
SYNOPSIS
    isreal(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a real value.  This function will return 1 if x
    is a real value, 0 otherwise.

EXAMPLE
    ; print isreal(2.0), isreal(1), isreal("0")
    1 1 0

    ; print isreal(2i), isreal(1e20), isreal(1/3)
    0 1 1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isrel,
    issimple, issq, isstr, istype
```

isrel - whether two values are relatively prime

```
SYNOPSIS
    isrel(x, y)

TYPES
    x       int
    y       int

    return  int

DESCRIPTION
    Determine if x and y are relatively prime.  If gcd(x,y) == 1, then
    return 1, otherwise return 0.

EXAMPLE
    ; print isrel(6, 5), isrel(5, 6), isrel(-5, 6)
    1 1 1

    ; print isrel(6, 2), isrel(2, 6), isrel(-2, 6)
    0 0 0

LIMITS
    none

LINK LIBRARY
    BOOL zrelprime(ZVALUE x, y)

SEE ALSO
    gcd,
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal,
    issimple, issq, isstr, istype
```

issimple - whether a value is a simple type

```
SYNOPSIS
    issimple(x)

TYPES
    x       any, &any

    return  int

DESCRIPTION
    Determine if x is a simple type.  This function will return 1 if x
    is a simple type, 0 otherwise.  Simple types are real numbers,
    complex numbers, strings and null values.

EXAMPLE
    ; print issimple(2.0), issimple(1), issimple("0")
    1 1 1

    ; print issimple(2i), issimple(1e20), issimple(1/3), issimple(null())
    1 1 1 1

    ; mat a[2]
    ; b = list(1,2,3)
    ; c = assoc()
    ; obj chongo {was, here} d;
    ; print issimple(a), issimple(b), issimple(c), issimple(d)
    0 0 0 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issq, isstr, istype
```

issq - whether a value is a square

```
SYNOPSIS
    issq(x)

TYPES
    x       real

    return  int

DESCRIPTION
    Determine if x is a square.      If there exists integers a, b such that:

      x == a^2 / b^2          (b != 0)

    return 1, otherwise return 0.

    Note that issq() works on rational values, so:

      issq(25/16) == 1

    If you want to test for prefect square integers, you need to exclude
    non-integer values before you test:

      isint(curds) && issq(curds)

EXAMPLE
    ; print issq(25), issq(3), issq(0)
    1 0 1

    ; print issq(4/25), issq(-4/25), issq(pi())
    1 0 0

LIMITS
    none

LINK LIBRARY
    BOOL qissquare(NUMBER *x)
    BOOL zissquare(ZVALUE x)

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, isstr, istype
```

isstr - whether a value is a string

```
SYNOPSIS
    isstr(x)

TYPES
    x        any, &any

    return   int

DESCRIPTION
    Determine if x is a string.      This function will return 1 if x is
    a string, 0 otherwise.

EXAMPLE
    ; print isstr("1"), isstr(1), isstr("")
    1 0 1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, istype
```

istype - whether the type of a value is the same as another

```
SYNOPSIS
    istype(x, y)

TYPES
    x        any, &any
    y        any, &any

    return   int

DESCRIPTION
    Determine if x has the same type as y.  This function will return 1
    if x and y are of the same type, 0 otherwise.

EXAMPLE
    ; print istype(2, 3), istype(2, 3.0), istype(2, 2.3)
    1 1 1

    ; print istype(2, 3i), istype(2, "2"), istype(2, null())
    0 0 0

    ; mat a[2]
    ; b = list(1,2,3)
    ; c = assoc()
    ; obj chongo {was, here} d;
    ; print istype(a,b), istype(b,c), istype(c,d)
    0 0 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isblk, isconfig, isdefined, iserror, iseven, isfile,
    ishash, isident, isint, islist, ismat, ismult, isnull, isnum, isobj,
    isobjtype, isodd, isprime, isrand, israndom, isreal, isrel,
    issimple, issq, isstr
```

jacobi - Jacobi symbol function

```
SYNOPSIS
    jacobi(x, y)

TYPES
    x       integer
    y       integer

    return  1, -1, or 0

DESCRIPTION
    If y is a positive odd prime and x is an integer not divisible
    by y, jacobi(x,y) returns the Legendre symbol function, usually
    denoted by (x/y) as if x/y were a fraction; this has the value
    1 or -1 according as x is or is not a quadratic residue modulo y.
    x is a quadratic residue modulo y if for some integer u,
    x = u^2 (mod y); if for all integers u, x != u^2 (mod y), x
    is said to be a quadratic nonresidue modulo y.

    If y is a positive odd prime and x is divisible by y, jacobi(x,y)
    returns the value 1.  (This differs from the zero value usually
    given in number theory books for (x/y) when x and y
    are not relatively prime.)
    assigned to (x/y) O

    If y is an odd positive integer equal to p_1 * p_2 * ... * p_k,
    where the p_i are primes, not necessarily distinct, the
    jacobi symbol function is given by

            jacobi(x,y)  =    (x/p_1) * (x/p_2) * ... * (x/p_k).

    where the functions on the right are Legendre symbol functions.

    This is also often usually by (x/y).

    If jacobi(x,y) = -1, then x is a quadratic nonresidue modulo y.
    Equivalently, if x is a quadratic residue modulo y, then
    jacobi(x,y) = 1.

    If jacobi(x,y) = 1 and y is composite, x may be either a quadratic
    residue or a quadratic nonresidue modulo y.

    If y is even or negative, jacobi(x,y) as defined by calc returns
    the value 0.

EXAMPLE
    ; print jacobi(2,3), jacobi(2,5), jacobi(2,15)
    -1 -1 1

    ; print jacobi(80,199)
    1

LIMITS
    none

LINK LIBRARY
```

```
    NUMBER *qjacobi(NUMBER *x, NUMBER *y)
    FLAG zjacobi(ZVALUE z1, ZVALUE z2)
```

SEE ALSO

join - form a list by concatenation of specified lists

```
SYNOPSIS
    join(x, y, ...)

TYPES
    x, y, ...     lists

    return  list or null

DESCRIPTION
    For lists x, y, ..., join(x, y, ...) returns the list whose length
    is the sum of the lengths of x, y, ..., in which the members of each
    argument immediately follow those of the preceding argument.
    The lists x, y, ... are not changed.

    If any argument is not a list, a null value is returned.

EXAMPLE
    ; A = list(1, 2, 3)
    ; B = list(4, 5)
    ; join(A, B)

    list (5 elements, 5 nonzero):
        [[0]] = 1
        [[1]] = 2
        [[2]] = 3
        [[3]] = 4
        [[4]] = 5

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    reverse, sort
```

lcm - least common multiple of a set of rational numbers

```
SYNOPSIS
    lcm(x1, x2, ...)

TYPES
    x1, x2, ... rational number

    return  rational number

DESCRIPTION
    Compute the least common multiple of one or more rational numbers.

    If no xi is zero, lcm(x1, x2, ...) is the least positive number v
    for which v is a multiple of each xi.  If at least one xi is zero,
    the lcm is zero.

EXAMPLE
    ; print lcm(12, -24, 30), lcm(9/10, 11/5, 4/25), lcm(2)
    -120 79.2 2

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    NUMBER *qlcm(NUMBER *x1, NUMBER *x2)

SEE ALSO
    gcd
```

lcmfact - lcm of positive integers up to specified integer

```
SYNOPSIS
    lcmfact(n)

TYPES
    n       positive integer

    return  positive integer

DESCRIPTION
    Returns the lcm of the integers 1, 2, ..., n.

EXAMPLE
    ; for (i = 1; i <= 15; i++) print lcmfact(i),:;
    1 2 6 12 60 60 420 840 2520 2520 27720 27720 360360 360360 360360

LIMITS
    n < 2^24

LINK LIBRARY
    NUMBER *qlcmfact(NUMBER *n)
    void zlcmfact(ZVALUE z, ZVALUE *dest)

SEE ALSO
    lcm, fact
```

lfactor - smallest prime factor in first specified number of primes

```
SYNOPSIS
    lfactor(n, m)

TYPES
    n       integer
    m       nonnegative integer <= 203280221 (= number of primes < 2^32)

    return  positive integer

DESCRIPTION
    This function ignores the signs of n and m, so here we shall
    assume n and limit are both nonnegative.

    If n is nonzero and abs(n) has a prime proper factor in the first
    m primes (2, 3, 5, ...), then lfactor(n, m) returns the smallest
    such factor.  Otherwise 1 is returned.

    If n is nonzero and m = pix(limit), then lfactor(n, m) returns the
    same as factor(n, limit).

    Both lfactor(n, 0) and lfactor(1, m) return 1 for all n and m.
    Also lfactor(0, m) always returns 1, and factor(0, limit) always
    returns 2 if limit >= 2.

EXAMPLE
    ; print lfactor(35,2), lfactor(35,3), lfactor(-35, 3)
    1 5 5

    ; print lfactor(2^32+1,115), lfactor(2^32+1,116), lfactor(2^59-1,1e5)
    1 641 179951

LIMITS
    m <= 203280221 (= number of primes < 2^32)

LINK LIBRARY
    NUMBER *qlowfactor(NUMBER *n, NUMBER *count)
    FULL zlowfactor(ZVALUE z, long count)

SEE ALSO
    factor, isprime, nextcand, nextprime, prevcand, prevprime,
    pfact, pix, ptest
```

list - create list of specified values

```
SYNOPSIS
    list([x, [x, ... ]])

TYPES
    x       any, &any

    return  list

DESCRIPTION
    This function returns a list that is composed of the arguments x.
    If no args are given, an empty list is returned.

    Lists are a sequence of values which are doubly linked so that
    elements can be removed or inserted anywhere within the list.
    The function 'list' creates a list with possible initial elements.
    For example,

          x = list(4, 6, 7);

    creates a list in the variable x of three elements, in the order
    4, 6, and 7.

    The 'push' and 'pop' functions insert or remove an element from
    the beginning of the list.      The 'append' and 'remove' functions
    insert or remove an element from the end of the list.  The 'insert'
    and 'delete' functions insert or delete an element from the middle
    (or ends) of a list.  The functions which insert elements return
    the null value, but the functions which remove an element return
    the element as their value.      The 'size' function returns the number
    of elements in the list.

    Note that these functions manipulate the actual list argument,
    instead of returning a new list.  Thus in the example:

          push(x, 9);

    x becomes a list of four elements, in the order 9, 4, 6, and 7.
    Lists can be copied by assigning them to another variable.

    An arbitrary element of a linked list can be accessed by using the
    double-bracket operator.  The beginning of the list has index 0.
    Thus in the new list x above, the expression x[[0]] returns the
    value of the first element of the list, which is 9.      Note that this
    indexing does not remove elements from the list.

    Since lists are doubly linked in memory, random access to arbitrary
    elements can be slow if the list is large.  However, for each list
    a pointer is kept to the latest indexed element, thus relatively
    sequential accesses to the elements in a list will not be slow.

    Lists can be searched for particular values by using the 'search'
    and 'rsearch' functions.  They return the element number of the
    found value (zero based), or null if the value does not exist in
    the list.
```

```
EXAMPLE
    ; list(2,"three",4i)

    list (3 elements, 3 nonzero):
      [[0]] = 2
      [[1]] = "three"
      [[2]] = 4i

    ; list()
          list (0 elements, 0 nonzero)

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    append, delete, insert, islist, pop, push, remove, rsearch, search, size
```

ln - logarithm function

```
SYNOPSIS
    ln(x [,eps])

TYPES
    x       nonzero real or complex
    eps           nonzero real, defaults to epsilon()

    return  real or complex

DESCRIPTION
    Approximate the natural logarithm function of x by a multiple of
    epsilon, the error having absolute value less than 0.75 * eps.
    If n is a positive integer, ln(x, 10^-n) will usually be correct
    to the n-th decimal place.

EXAMPLE
    ; print ln(10, 1e-5), ln(10, 1e-10), ln(10, 1e-15), ln(10, 1e-20)
    2.30259 2.302585093 2.302585092994046 2.30258509299404568402

    ; print ln(2+3i, 1e-5), ln(2+3i, 1e-10)
    1.28247+.98279i 1.2824746787+.9827937232i

LIMITS
    x != 0
    eps > 0

LINK LIBRARY
    NUMBER *qln(NUMBER *x, NUMBER *eps)
    COMPLEX *c_ln(COMPLEX *x, NUMBER *eps)

SEE ALSO
    exp, acosh, asinh, atanh, log
```

log - base 10 logarithm

```
SYNOPSIS
    log(x [,eps])

TYPES
    x       nonzero real or complex
    eps             nonzero real, defaults to epsilon()

    return  real or complex

DESCRIPTION
    Approximate the base 10 logarithm function of x by a multiple of
    epsilon, the error having absolute value less than 0.75 * eps.
    If n is a positive integer, log(x, 10^-n) will usually be correct
    to the n-th decimal place.

EXAMPLE
    ; print log(10), log(100), log(1e10), log(1e500)
    1 2 10 500

    ; print log(128), log(23209), log(2^17-19)
    ~2.10720996964786836649 ~4.36565642852838930424 ~5.11744696704937330414

    ; print log(2+3i, 1e-5)
    ~0.55696845725899964822+~0.42681936428109216144i

LIMITS
    x != 0
    eps > 0

LINK LIBRARY
    NUMBER *qlog(NUMBER *x, NUMBER *eps)
    COMPLEX *c_log(COMPLEX *x, NUMBER *eps)

SEE ALSO
    ln
```

lowbit - index of lowest nonzero bit in binary representation of integer

```
SYNOPSIS
    lowbit(x)

TYPES
    x       nonzero integer

    return  integer

DESCRIPTION
    If x is a nonzero integer, lowbit(x) returns the index of the
    lowest nonzero bit in the binary representation of abs(x).    Equivalently,
    lowbit(x) is the greatest integer for which x/2^n is an integer;
    the binary representation of x then ends with n zero bits.

EXAMPLE
    ; print lowbit(2), lowbit(3), lowbit(4), lowbit(-15), lowbit(2^27)
    1 0 2 0 27

LIMITS
    none

LINK LIBRARY
    long zlowbit(ZVALUE x);

SEE ALSO
    highbit, digits
```

ltol - "leg to leg", third side of a right-angled triangle with

```
            unit hypotenuse, given one other side

SYNOPSIS
    ltol(x, [,eps])

TYPES
    x       real
    eps           nonzero real

    return  real

DESCRIPTION
    Returns sqrt(1 - x^2) to the nearest multiple of eps.
    The default value for eps is epsilon().

EXAMPLE
    ; print ltol(0.4, 1e-6), hypot(0.5, 1e-6)
    .6 .866025

LIMITS
    abs(x) <= 1

LINK LIBRARY
    NUMBER *qlegtoleg(NUMBER *q1, *epsilon, BOOL wantneg)

SEE ALSO
    hypot
```

makelist - create a list with a specified number of null members

```
SYNOPSIS
    makelist(x)

TYPES
    x       int

    return  list

DESCRIPTION
    For non-negative integer x, makelist(x) returns a list of size x
    all members of which have null value.

EXAMPLE
    ; A = makelist(4)
    ; A

    list (4 members, 4 nonzero):
        [[0]] = NULL
        [[1]] = NULL
        [[2]] = NULL
        [[3]] = NULL

LIMITS
    0 <= x < 2^31

LINK LIBRARY
    none

SEE ALSO
    modify
```

mat - keyword to create a matrix value

```
SYNOPSIS
    mat [index-range-list] [ = {value_0. ...} ]
    mat [] [= {value_0, ...}]
    mat variable_1 ... [index-range-list] [ = {value_0, ...} ]
    mat variable_1 ... [] [ = {value_0, ...} ]

    mat [index-range-list_1[index-ranges-list_2] ... [ = { { ...} ...}  ]

    decl id_1 id_2 ... [index-range-list] ...

TYPES
    index-range-list    range_1 [, range_2, ...]  up to 4 ranges
    range_1, ...        integer, or integer_1 : integer_2
    value, value_1, ... any
    variable_1 ...      lvalue
    decl                declarator = global, static or local
    id_1, ...           identifier

DESCRIPTION
    The expression  mat [index-range-list]  returns a matrix value.
    This may be assigned to one or more lvalues A, B, ... by either

      mat A B ... [index-range-list]

    or

      A = B = ... = mat[index-range-list]

    If a variable is specified by an expression that is not a symbol with
    possibly object element specifiers, the expression should be enclosed
    in parentheses.  For example, parentheses are required in
    mat (A[2]) [3]  and mat (*p) [3]  but  mat P.x [3] is acceptable.

    When an index-range is specified as integer_1 : integer_2, where
    integer_1 and integer_2 are expressions which evaluate to integers,
    the index-range consists of all integers from the minimum of the
    two integers to the maximum of the two integers.  For example,
    mat[2:5, 0:4] and mat[5:2, 4:0] return the same matrix value.

    If an index-range is an expression which evaluates to an integer,
    the range is as if specified by 0 : integer - 1.  For example,
    mat[4] and mat[0:3] return the same 4-element matrix; mat[-2] and
    mat[-3:0] return the same 4-element matrix.

    If the variable A has a matrix value, then for integer indices
    i_1, i_2, ..., equal in number to the number of ranges specified at
    its creation, and such that each index is in the corresponding range,
    the matrix element associated with those index list is given as an
    lvalue by the expressions A[i_1, i_2, ...].

    The elements of the matrix are stored internally as a linear array
    in which locations are arranged in order of increasing indices.
    For example, in order of location, the six element of A = mat [2,3]
    are
```

```
  A[0,0], A[0,1], A[0,2], A[1,0], A[1,,1], A[1,2].
```

These elements may also be specified using the double-bracket operator with a single integer index as in A[[0]], A[[1]], ..., A[[5]]. If p is assigned the value &A[0.0], the address of A[[i]] for 0 <= i < 6 is p + i as long as A exists and a new value is not assigned to A.

When a matrix is created, each element is initially assigned the value zero.   Other values may be assigned then or later using the "= {...}" assignment operation. Thus

```
  A = {value_0, value_1, ...}
```

assigns the values value_0, value_1, ... to the elements A[[0]], A[[1]], ...   Any blank "value" is passed over. For example,

```
  A = {1, , 2}
```

will assign the value 1 to A[[0]], 2 to A[[2]] and leave all other elements unchanged. Values may also be assigned to elements by simple assignments, as in A[0,0] = 1, A[0,2] = 2;

If the index-range is left blank but an initializer list is specified as in:

```
  ; mat A[] = {1, 2 }
  ; B = mat[] = {1, , 3, }
```

the matrix created is one-dimensional. If the list contains a positive number n of values or blanks, the result is as if the range were specified by [n], i.e. the range of indices is from 0 to n - 1.   In the above examples, A is of size 2 with A[0] = 1 and A[1] = 2; B is of size 4 with B[0] = 1, B[1] = B[3] = 0, B[2] = 3. The specification mat[] = { } creates the same as mat[1].

If the index-range is left blank and no initializer list is specified, as in mat C[] or C = mat[], the matrix assigned to C has zero dimension; this has one element C[].

To assign a value using "= { ...}" at the same time as creating C, parentheses are required as in (mat[]) = {value} or (mat C[]) = {value}. Later a value may be assigned to C[] by C[] = value or C = {value}.

The value assigned at any time to any element of a matrix can be of any type - number, string, list, matrix, object of previously specified type, etc. For some matrix operations there are of course conditions that elements may have to satisfy: for example, addition of matrices requires that addition of corresponding elements be possible. If an element of a matrix is a structure for which indices or an object element specifier is required, an element of that structure is referred to by appropriate uses of [ ] or ., and so on if an element of that element is required.

For example, one may have an expressions like:

```
  ; A[1,2][3].alpha[2];
```

if A[1,2][3].alpha is a list with at least three elements, A[1,2][3] is an object of a type like obj {alpha, beta}, A[1,2] is a matrix of type mat[4] and A is a mat[2,3] matrix. When an element of a matrix is a matrix and the total number of indices does not exceed 4, the indices can be combined into one list, e.g. the A[1,2][3] in the above example can be shortened to A[1,2,3]. (Unlike C, A[1,2] cannot be expressed as A[1][2].)

The function ismat(V) returns 1 if V is a matrix, 0 otherwise.

isident(V) returns 1 if V is a square matrix with diagonal elements 1, off-diagonal elements zero, or a zero- or one-dimensional matrix with every element 1; otherwise zero is returned. Thus isident(V) = 1 indicates that for V * A and A * V where A is any matrix of for which either product is defined and the elements of A are real or complex numbers, that product will equal A.

If V is matrix-valued, test(V) returns 0 if every element of V tests as zero; otherwise 1 is returned.

The dimension of a matrix A, i.e. the number of index-ranges in the initial creation of the matrix, is returned by the function matdim(A). For 1 <= i <= matdim(A), the minimum and maximum values for the i-th index range are returned by matmin(A, i) and matmax(A,i), respectively. The total number of elements in the matrix is returned by size(A). The sum of the elements in the matrix is returned by matsum(A).

The default method of printing matrices is to give a line of information about the matrix, and to list on separate lines up to 15 elements, the indices and either the value (for numbers, strings, objects) or some descriptive information for lists or matrices, etc. Numbers are displayed in the current number-printing mode. The maximum number of elements to be printed can be assigned any nonnegative integer value m by config("maxprint", m).

Users may define another method of printing matrices by defining a function mat_print(M); for example, for a not too big 2-dimensional matrix A it is a common practice to use a loop like:

```
  define mat_print(A) {
        local i,j;

        for (i = matmin(A,1); i <= matmax(A,1); i++) {
              if (i != matmin(A,1))
                    printf("\t");
              for (j = matmin(A,2); j <= matmax(A,2); j++)
                    printf(" [%d,%d]: %e", i, j, A[i,j]);
              if (i != matmax(A,1))
                    printf("\n");
        }
  }
```

So that when one defines a 2D matrix such as:

```
  ; mat X[2,3] = {1,2,3,4,5,6}
```

then printing X results in:

```
  [0,0]: 1 [0,1]: 2 [0,2]: 3
  [1,0]: 4 [1,1]: 5 [1,2]: 6
```

```
The default printing may be restored by

  ; undefine mat_print;

The keyword "mat" followed by two or more index-range-lists returns a
matrix with indices specified by the first list, whose elements are
matrices as determined by the later index-range-lists.  For
example  mat[2][3]  is a 2-element matrix, each of whose elements has
as its value a 3-element matrix.  Values may be assigned to the
elements of the innermost matrices by nested = {...} operations as in

  ; mat [2][3] = {{1,2,3},{4,5,6}}

An example of the use of mat with a declarator is

  ; global mat A B [2,3], C [4]

This creates, if they do not already exist, three global variables with
names A, B, C, and assigns to A and B the value mat[2,3] and to C mat[4].

Some operations are defined for matrices.

A == B
  Returns 1 if A and B are of the same "shape" and "corresponding"
  elements are equal; otherwise 0 is returned.  Being of the same
  shape means they have the same dimension d, and for each i <= d,

      matmax(A,i) - matmin(A,i) == matmax(B,i) - matmin(B,i),

  One consequence of being the same shape is that the matrices will
  have the same size.   Elements "correspond" if they have the same
  double-bracket indices; thus A == B implies that A[[i]] == B[[i]]
  for 0 <= i < size(A) == size(B).

A + B
A - B
  These are defined A and B have the same shape, the element
  with double-bracket index j being evaluated by A[[j]] + B[[j]] and
  A[[j]] - B[[j]], respectively.      The index-ranges for the results
  are those for the matrix A.

A[i,j]
  If A is two-dimensional, it is customary to speak of the indices
  i, j in A[i,j] as referring to rows and columns;  the number of
  rows is matmax(A,1) - matmin(A,1) + 1; the number of columns if
  matmax(A,2) - matmin(A,2) + 1.      A matrix is said to be square
  if it is two-dimensional and the number of rows is equal to the
  number of columns.

A * B
  Multiplication is defined provided certain conditions by the
  dimensions and shapes of A and B are satisfied.  If both have
  dimension 2 and the column-index-list for A is the same as
  the row-index-list for B, C = A * B is defined in the usual
  way so that for i in the row-index-list of A and j in the
  column-index-list for B,

        C[i,j] =  Sum A[i,k] * B[k,j]
```

```
  the sum being over k in the column-index-list of A.  The same
  formula is used so long as the number of columns in A is the same
  as the number of rows in B and k is taken to refer to the offset
  from matmin(A,2) and matmin(B,1), respectively, for A and B.
  If the multiplications and additions required cannot be performed,
  an execution error may occur or the result for C may contain
  one or more error-values as elements.

  If A or B has dimension zero, the result for A * B is simply
  that of multiplying the elements of the other matrix on the
  left by A[] or on the right by B[].

  If both A and B have dimension 1, A * B is defined if A and B
  have the same size; the result has the same index-list as A
  and each element is the product of corresponding elements of
  A and B.  If A and B have the same index-list, this multiplication
  is consistent with multiplication of 2D matrices if A and B are
  taken to represent 2D matrices for which the off-diagonal elements
  are zero and the diagonal elements are those of A and B.
  the real and complex numbers.

  If A is of dimension 1 and B is of dimension 2, A * B is defined
  if the number of rows in B is the same as the size of A.  The
  result has the same index-lists as B; each row of B is multiplied
  on the left by the corresponding element of A.

  If A is of dimension 2 and B is of dimension 1, A * B is defined
  if number of columns in A is the same as the size of A.      The
  result has the same index-lists as A; each column of A is
  multiplied on the right by the corresponding element of B.

  The algebra of additions and multiplications involving both one-
  and two-dimensional matrices is particularly simple when all the
  elements are real or complex numbers and all the index-lists are
  the same, as occurs, for example, if for some positive integer n,
  all the matrices start as  mat [n]  or    mat [n,n].

det(A)
  If A is a square, det(A) is evaluated by an algorithm that returns
  the determinant of A if the elements of A are real or complex
  numbers, and if such an A is non-singular, inverse(A) returns
  the inverse of A indexed in the same way as A.  For matrix A of
  dimension 0 or 1, det(A) is defined as the product of the elements
  of A in the order in which they occur in A, inverse(A) returns
  a matrix indexed in the same way as A with each element inverted.


The following functions are defined to return matrices with the same
  index-ranges as A and the specified operations performed on all
  elements of A.    Here num is an arbitrary complex number (nonzero
  when it is a divisor), int an integer, rnd a rounding-type
  specifier integer, real a real number.

      num * A
      A * num
      A / num
      - A
      conj(A)
```

```
A << int, A >> int
scale(A, int)
round(A, int, rnd)
bround(A, int, rnd)
appr(A, real, rnd)
int(A)
frac(A)
A // real
A % real
A ^ int
```

If A and B are one-dimensional of the same size dp(A, B) returns their dot-product, i.e. the sum of the products of corresponding elements.

If A and B are one-dimension and of size 3, cp(A, B) returns their cross-product.

randperm(A) returns a matrix indexed the same as A in which the elements of A have been randomly permuted.

sort(A) returns a matrix indexed the same as A in which the elements of A have been sorted.

If A is an lvalue whose current value is a matrix, matfill(A, v) assigns the value v to every element of A, and if also, A is square, matfill(A, v1, v2) assigns v1 to the off-diagonal elements, v2 to the diagonal elements. To create and assign to A the unit n * n matrix, one may use matfill(mat A[n,n], 0, 1).

For a square matrix A, mattrace(A) returns the trace of A, i.e. the sum of the diagonal elements. For zero- or one-dimensional A, mattrace(A) returns the sum of the elements of A.

For a two-dimensional matrix A, mattrans(A) returns the transpose of A, i.e. if A is mat[m,n], it returns a mat[n,m] matrix with [i,j] element equal to A[j,i]. For zero- or one-dimensional A, mattrace(A) returns a matrix with the same value as A.

The functions search(A, value, start, end]) and rsearch(A, value, start, end]) return the first or last index i for which A[[i]] == value and start <= i < end, or if there is no such index, the null value. For further information on default values and the use of an "accept" function, see the help files for search and rsearch.

reverse(A) returns a matrix with the same index-lists as A but the elements in reversed order.

The copy and blkcpy functions may be used to copy data to a matrix from a matrix or list, or from a matrix to a list. In copying from a matrix to a matrix the matrices need not have the same dimension; in effect they are treated as linear arrays.

EXAMPLE

```
; obj point {x,y}
; mat A[5] = {1, 2+3i, "ab", mat[2] = {4,5}, obj point = {6,7}}
; A
mat [5] (5 elements, 5 nonzero):
```

```
      [0] = 1
      [1] = 2+3i
      [2] = "ab"
      [3] = mat [2] (2 elements, 2 nonzero)
      [4] = obj point {6, 7}

    ; print A[0], A[1], A[2], A[3][0], A[4].x
    1 2+3i ab 4 6

    ; define point_add(a,b) = obj point = {a.x + b.x, a.y + b.y}
    point_add(a,b) defined

    ; mat [B] = {8, , "cd", mat[2] = {9,10}, obj point = {11,12}}
    ; A + B

    mat [5] (5 elements, 5 nonzero):
      [0] = 9
      [1] = 2+3i
      [2] = "abcd"
      [3] = mat [2] (2 elements, 2 nonzero)
      [4] = obj point {17, 19}

    ; mat C[2,2] = {1,2,3,4}
    ; C^10

    mat [2,2] (4 elements, 4 nonzero):
      [0,0] = 4783807
      [0,1] = 6972050
      [1,0] = 10458075
      [1,1] = 15241882

    ; C^-10

    mat [2,2] (4 elements, 4 nonzero):
      [0,0] = 14884.650390625
      [0,1] = -6808.642578125
      [1,0] = -10212.9638671875
      [1,1] = 4671.6865234375

    ; mat A[4] = {1,2,3,4}, A * reverse(A);

    mat [4] (4 elements, 4 nonzero):
      [0] = 4
      [1] = 6
      [2] = 6
      [3] = 4
```

LIMITS

The theoretical upper bound for the absolute values of indices is 2^31 - 1, but the size of matrices that can be handled in practice will be limited by the availability of memory and what is an acceptable runtime. For example, although it may take only a fraction of a second to invert a 10 * 10 matrix, it will probably take about 1000 times as long to invert a 100 * 100 matrix.

LINK LIBRARY

n/a

SEE ALSO

```
ismat, matdim, matmax, matmin, mattrans, mattrace, matsum, matfill,
det, inverse, isident, test, config, search, rsearch, reverse, copy,
blkcpy, dp, cp, randperm, sort
```

matdim - matrix dimension

```
SYNOPSIS
    matdim(m)

TYPES
    m       matrix

    return  1, 2, 3, or 4

DESCRIPTION
    Returns the number of indices required to specify elements of the matrix.

EXAMPLE
    ; mat A[3]; mat B[2,3]; mat C[1, 2:3, 4]; mat D[2, 3, 4, 5]
    ; print matdim(A), matdim(B), matdim(C), matdim(D)
    1 2 3 4

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mat, ismat, matmax, matmin, mattrans, mattrace, matsum, matfill,
    det, inverse, isident, test, config, search, rsearch, reverse, copy,
    blkcpy, dp, cp, randperm, sort
```

matfill - fill a matrix with specified value or values

```
SYNOPSIS
    mat(m, x [, y])

TYPES
    m       matrix
    x       any
    y       any

    return  null

DESCRIPTION
    For any matrix m, matfill(m, x) assigns to every element of m the
    value x.  For a square matrix m, matfill(m, x, y) assigns the value
    x to the off-diagonal elements, y to the diagonal elements.

EXAMPLE
    ; mat A[3]; matfill(A, 2); print A
    mat [3] (3 elements, 3 nonzero):
      [0] = 2
      [1] = 2
      [2] = 2

    ; mat B[2, 1:2]; matfill(B,3,4); print B
    mat [2,1:2] (4 elements, 4 nonzero):
      [0,1] = 4
      [0,2] = 3
      [1,1] = 3
      [1,2] = 4

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mat, ismat, matdim, matmax, matmin, mattrans, mattrace, matsum,
    det, inverse, isident, test, config, search, rsearch, reverse, copy,
    blkcpy, dp, cp, randperm, sort
```

matmax - maximum value for specified index of matrix

```
SYNOPSIS
    matmax(m, i)

TYPES
    m       matrix
    i       0, 1, 2, 3

    return  integer

DESCRIPTION
    Returns the maximum value for i-th index (i counting from zero)
    for the matrix m.

EXAMPLE
    ; mat A[3]; mat B[1:3, -4:4, 5]
    ; print matmax(A,0), matmax(B,0), matmax(B,1), matmax(B,2)
    2 3 4 4

LIMITS
    i < matdim(m)

LINK LIBRARY
    none

SEE ALSO
    mat, ismat, matdim, matmin, mattrans, mattrace, matsum, matfill,
    det, inverse, isident, test, config, search, rsearch, reverse, copy,
    blkcpy, dp, cp, randperm, sort
```

matmin - minimum value for specified index of matrix

```
SYNOPSIS
    matmin(m, i)

TYPES
    m       matrix
    i       0, 1, 2, 3

    return  integer

DESCRIPTION
    Returns the minimum value for i-th index (i counting from zero)
    for the matrix m.

EXAMPLE
    ; mat A[3]; mat B[1:3, -4:4, 5]
    ; print matmin(A,0), matmin(B,0), matmin(B,1), matmin(B,2)
    0 1 -4 0

LIMITS
    i < matdim(m)

LINK LIBRARY
    none

SEE ALSO
    mat, ismat, matdim, matmax, mattrans, mattrace, matsum, matfill,
    det, inverse, isident, test, config, search, rsearch, reverse, copy,
    blkcpy, dp, cp, randperm, sort
```

matsum - sum the elements of a matrix

```
SYNOPSIS
    matsum(m)

TYPES
    m       matrix with any types of elements

    return  number

DESCRIPTION
    Returns the sum of the numeric (real or complex) elements of m.
    Non-numeric elements are ignored.

EXAMPLE
    ; mat A[2,2] = {1, 2, 3, list(1,2,3)}
    print matsum(A)
      6

LIMITS
    none

LINK LIBRARY
    void matsum(MATRIX *m, VALUE *vres);

SEE ALSO
    mat, ismat, matdim, matmax, matmin, mattrans, mattrace, matfill,
    det, inverse, isident, test, config, search, rsearch, reverse, copy,
    blkcpy, dp, cp, randperm, sort
```

mattrace - trace of a square matrix

```
SYNOPSIS
    mattrace(m)

TYPES
    m       square matrix with summable diagonal elements

    return  determined by addition of elements

DESCRIPTION
    For a two-dimensional square matrix, mattrace(m) returns the sum of
    the elements on the principal diagonal.  In particular, if m
    has been created by mat m[N,N] where N > 0, mattrace(m) returns

            m[0,0] + m{1,1] + ... + m[N-1,N-1]

EXAMPLE
    ; mat m[2,2] = {1,2,3,4}
    ; print mattrace(m), mattrace(m^2)
    5 29

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mat, mattrans
```

mattrans - matrix transpose

```
SYNOPSIS
    matdim(m)

TYPES
    m       2-dimensional matrix

    return  2-dimensional matrix

DESCRIPTION
    Returns the matrix whose [i,j] element is the [j,1] element of m.

EXAMPLE
    ; mat A[2, 1:3] = {1,2,3,4,5,6}
    ; print mattrans(A)

    mat [1:3,2] (6 elements, 6 nonzero):
      [1,0] = 1
      [1,1] = 4
      [2,0] = 2
      [2,1] = 5
      [3,0] = 3
      [3,1] = 6

LIMITS
    none

LINK LIBRARY
    MATRIX *mattrans(MATRIX *m)

SEE ALSO
    mat, ismat, matdim, matmax, matmin, mattrace, matsum, matfill,
    det, inverse, isident, test, config, search, rsearch, reverse, copy,
    blkcpy, dp, cp, randperm, sort
```

max - maximum, or maximum of defined maxima

```
SYNOPSIS
    max(x_1, x_2, ...)

TYPES
    x_1, x_2, ... any

    return        any

DESCRIPTION
    If an argument x_i is a list with elements e_1, e_2, ..., e_n, it
    is treated as if x_i were replaced by e_1, e_2, ..., e_n; this may
    continue recurively if any of the e_j is a list.

    If an argument x_i is an object of type xx, then x_i is replaced by
    xx_max(x_i) if the function xx_max() has been defined. If the
    type xx has been defined by:

            obj xx = {x, y, z},

    an appropriate definition of xx_max(a) is sometimes max(a.x, a.y, a.z).
    max(a) then returns the maximum of the elements of a.

    If x_i has the null value, it is ignored.  Thus, sum(a, , b, , c)

    If x_i has the null value, it is ignored.  Thus, max(a, , b, , c)
    will return the same as max(a, b, c).

    Assuming the above replacements, and that the x_1, x_2, ..., are
    of sufficently simple ordered types (e.g. real numbers or strings),
    or, if some are objects, the relevant xx_rel(a,b) has been defined
    and returns a real-number value for any comparison that has to be made,
    max(x_1, x_2, ...) returns the value determined by max(x_1) = x_1,
    and succesively for later arguments, by the use of the equivalent of
    max(a,b) = (a < b) ? b : a.      If the ordering determined by < is total,
    max(x_1, ...) will be the maximum value among the arguments.  For a
    preorder relation it may be one of several maximal values.    For
    other relations, it may be difficult to predict the result.

EXAMPLE
    ; print max(2), max(5, 3, 7, 2, 9), max(3.2, -0.5, 8.7, -1.2, 2.5)
    2 9 8.7

    ; print max(list(3,5), 7, list(6, list(7,8), 2))
    8

    ; print max("one", "two", "three", "four")
    two

    ; obj point {x, y}
    ; define point_rel(a,b) = sgn(a.x - b.x)
    ; obj point A = {1, 5}
    ; obj point B = {1, 4}
    ; obj point C = {3, 3}
    ; print max(A, B, C)
    obj point {3, 3}
```

```
    ; define point_max(a) = a.x
    ; print max(A, B, C)
    3
```

```
LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    NUMBER *qmax(NUMBER *x1, NUMBER *x2)

SEE ALSO
    max, obj
```

memsize - number of bytes required for value including overhead

```
SYNOPSIS
    memsize(x)

TYPES
    x       any

    return  integer

DESCRIPTION
    This is analogous to the C operator sizeof.  It attempts to assess
    the number of bytes in memory used to store a value and all its
    components plus all of the related structue overhead.  Unlike
    sizeof(x), this builtin includes overhead.

    Unlike size(x), this builtin incldues the trailing \0 byte on the
    end of strings.

    Unlike sizeof(x), this builtin includes the size demonitor for integers
    and the imaginary part for complex values.  Storage for holding
    0, 1 and -1 values are also included.

    The number returned by memsize(x) may be less than the actual number
    used because, for example, more memory may have been allocated for
    a string than is used: only the characters up to and including the
    first '\0' are counted in calculating the contribution of the
    string to memsize(x).

    The number returned by memsize(x) may be greater (and indeed
    substantially greater) than the number of bytes actually used.
    For example, after:

            a = sqrt(2);
            mat A[3] = {a, a, a};

    the numerical information for a, A[0], A[1], A[2] are stored in the
    same memory, so the memory used for A is the same as if
    its 3 elements were null values.  The value returned by
    memsize(A) is calculated as A were defined by:

            mat A[3] = {sqrt(2), sqrt(2), sqrt(2)}.

    Similar sharing of memory occurs with literal strings.

    For associative arrays, both the name part and the value part of
    the name/value pair are counted.

    The minimum value for memsize(x) occurs for the null and error values.

EXAMPLES

    The results for examples like these will depend to some extent on
    the system being used.  The following were for an SGI R4k machine
    in 32-bit mode:
```

```
    ; print memsize(null())
    8

    ; print memsize(0), memsize(3), memsize(2^32 - 1), memsize(2^32)
    68 68 68 72

    ; x = sqrt(2, 1e-100); print memsize(x), memsize(num(x)), memsize(den(x))
    148 108 108

    ; print memsize(list()), memsize(list(1)), memsize(list(1,2))
    28 104 180

    ; print memsize(list())
    28

    ; print ,memsize(list(1)),memsize(list(1,2)),memsize(list(1,2,3))
    104 180 256

    ; mat A[] = {1}; mat B[] = {1,2}; mat C[] = {1,2,3}; mat D[100,100];
    ; print memsize(A), memsize(B), memsize(C), memsize(D)
    124 192 260 680056

    ; obj point {x,y,z}
    ; obj point P = {1,2,3}; print memsize(P)
    274
```

```
LIMITS
    It is assumed memsize(x) will fit into a system long integer.

LINK LIBRARY
    none

SEE ALSO
    size, sizeof, fsize, strlen, digits
```

meq - test for equality modulo a specifed number

```
SYNOPSIS
    meq(x, y, md)

TYPES
    x       real
    y       real
    md            real

    return  0 or 1

DESCRIPTION
    Returns 1 if and only if for some integer n, x - y = n * md, i.e.
    x is congruent to y modulo md.

    If md = 0, this is equivalent to x == y.

    For any x, y, md, meq(x, y, md) = ismult(x - y, md).

EXAMPLE
    ; print meq(5, 33, 7), meq(.05, .33, -.07), meq(5, 32, 7)
    1 1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mne, ismult
```

min - minimum, or minimum of defined minima

```
SYNOPSIS
    min(x_1, x_2, ...)

TYPES
    x_1, x_2, ... any

    return        any

DESCRIPTION
    If an argument x_i is a list with elements e_1, e_2, ..., e_n, it
    is treated as if x_i were replaced by e_1, e_2, ..., e_n; this may
    continue recurively if any of the e_j is a list.

    If an argument x_i is an object of type xx, then x_i is replaced by
    xx_min(x_i) if the function xx_min() has been defined.  If the
    type xx has been defined by:

            obj xx = {x, y, z},

    an appropriate definition of xx_min(a) is sometimes min(a.x, a.y, a.z).
    min(a) then returns the minimum of the elements of a.

    If x_i has the null value, it is ignored.  Thus, sum(a, , b, , c)

    If x_i has the null value, it is ignored.  Thus, min(a, , b, , c)
    will return the same as min(a, b, c).

    Assuming the above replacements, and that the x_1, x_2, ..., are
    of sufficently simple ordered types (e.g. real numbers or strings),
    or, if some are objects, the relevant xx_rel(a,b) has been defined
    and returns a real-number value for any comparison that has to be made,
    min(x_1, x_2, ...) returns the value determined by min(x_1) = x_1,
    and succesively for later arguments, by the use of the equivalent of
    min(a,b) = (a < b) ? a : b.      If the ordering determined by < is total,
    min(x_1, ...) will be the minimum value among the arguments.  For a
    preorder relation it may be one of several minimal values.    For other
    relations, it may be difficult to predict the result.

EXAMPLE
    ; print min(2), min(5, 3, 7, 2, 9), min(3.2, -0.5, 8.7, -1.2, 2.5)
    2 2 -1.2

    ; print min(list(3,5), 7, list(6, list(7,8), 2))
    2

    ; print min("one", "two", "three", "four")
    four

    ; obj point {x, y}
    ; define point_rel(a,b) = sgn(a.x - b.x)
    ; obj point A = {1, 5}
    ; obj point B = {1, 4}
    ; obj point C = {3, 3}
    ; print min(A, B, C)
    obj point {1, 5}
```

```
    ; define point_min(a) = a.x
    ; print min(A, B, C)
    1
```

LIMITS

    The number of arguments is not to exceed 1024.

LINK LIBRARY

```
    NUMBER *qmin(NUMBER *x1, NUMBER *x2)
```

SEE ALSO

    max, obj, sum, ssq

minv - inverse of an integer modulo a specified integer

```
SYNOPSIS
    minv(x, md)

TYPES
    x        integer
    md             integer

    return   integer

DESCRIPTION
    If x and md are not relatively prime, zero is returned.
    Otherwise v = minv(x, md) is the canonical residue v modulo md
    for which v * x is congruent to 1 modulo md.   The canonical
    residues modulo md are determined as follows by md and bits 0, 2
    and 4 of config("mod") (other bits are ignored).

            config("mod")       md > 0           md < 0

                0       0 < v < md      md < v < 0
                1       -md < v < 0      0 < v < -md
                4       0 < v < md       0 < v < -md
                5       -md < v < 0     md < v < 0
               16      -md/2 < v <= md/2   md/2 <= v < -md/2
               17    -md/2 <= v < md/2     md/2 < v <= -md/2
               20     -md/2 < v <= md/2    md/2 < v <= -md/2
               21    -md/2 <= v < md/2    md/2 <= v < -md/2

EXAMPLE
    ; c = config("mod", 0)
    ; print minv(3,10), minv(-3,10), minv(3,-10), minv(-3,-10), minv(4,10)
    7 3 -3 -7 0

    ; c = config("mod",16)
    ; print minv(3,10), minv(-3,10), minv(3,-10), minv(-3,-10), minv(4,10)
    -3 3 -3 3 0

LIMITS
    none

LINK LIBRARY
    NUMBER *qminv(NUMBER *x, NUMBER *md)

SEE ALSO
    mod, pmod
```

mmin - least-absolute-value residues modulo a specified number

```
SYNOPSIS
    mmin(x, md)

TYPES
    x       number (real or complex), matrix, list, object
    md            real

    return  real

DESCRIPTION
    If x is real and md is nonzero, mmin(x, md) returns the real
    number v congruent to x modulo md for which abs(v) <= md/2
    and if abs(v) = md/2, then v = md/2.

    If x is real and md is zero, mmin(x, md) returns x.

    For complex, matrix, list or object x, see the help file for mod: for
    all x and md, mmin(x, md) returns the same as mod(x, md, 16).

EXAMPLE
    ; print mmin(3,6), mmin(4,6), mmin(5,6), mmin(6,6), mmin(7,6)
    3 -2 -1 0 1

    ; print mmin(1.25, 2.5), mmin(-1.25,2.5), mmin(1.25, -2.5)
    1.25 1.25 -1.25

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    mod
```

mne - test for inequality of real numbers modulo a specifed number

```
SYNOPSIS
    mne(x, y, md)

TYPES
    x        real number
    y        real number
    md             real number

    return  0 or 1

DESCRIPTION
    Returns 1 if and only if x is not congruent to y modulo md, i.e.
    for every integer n, x - y != n * md.

EXAMPLE
    print mne(5, 33, 7), mne(5, -23, 7), mne(5, 15, 7), mne(5, 7, 0)
    0 0 1 1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    meq
```

mod - compute the remainder for an integer quotient

```
SYNOPSIS
    mod(x, y, rnd)
    x % y

TYPES
    If x is a matrix or list, the returned value is a matrix or list v of
    the same structure for which each element v[[i]] = mod(x[[i]], y, rnd).

    If x is an xx-object or x is not an object and y is an xx-object,
    this function calls the user-defined function xx_mod(x, y, rnd);
    the types of arguments and returned value are as required by the
    definition of xx_mod().

    If neither x nor y is an object, or x is not a matrix or list:

    x       number (real or complex)
    y       real
    rnd           integer, defaults to config("mod")

    return  number

DESCRIPTION
    The expression:

      x % y

    is equivalent to call:

      mod(x, y)

    The function:

      mod(x, y, rnd)

    is equivalent to:

      config("mod", rnd), x % y

    except that the global config("mod") value does not change.

    If x is real or complex and y is zero, mod(x, y, rnd) returns x.

    If x is complex, mod(x, y, rnd) returns
            mod(re(x), y, rnd) + mod(im(x), y, rnd) * 1i.

    In the following it is assumed x is real and y is nonzero.

    If x/y is an integer mod(x, y, rnd) returns zero.

    If x/y is not an integer, mod(x, y, rnd) returns one of the two
    values of r for which for some integer q exists such that x = q * y + r
    and abs(r) < abs(y).  Which of the two values or r that is returned is
    controlled by rnd.

    If bit 4 of rnd is set (e.g. if 16 <= rnd < 32) abs(r) <= abs(y)/2;
```

```
    this uniquely determines r if abs(r) < abs(y)/2.  If bit 4 of rnd is
    set and abs(r) = abs(y)/2, or if bit 4 of r is not set, the result for
    r depends on rnd as in the following table:

           rnd & 15        sign of r            parity of q

            0          sgn(y)
            1         -sgn(y)
            2          sgn(x)
            3         -sgn(x)
            4           +
            5           -
            6          sgn(x/y)
            7         -sgn(x/y)
            8                            even
            9                            odd
             10                       even if x/y > 0, otherwise odd
             11                       odd if x/y > 0, otherwise even
             12                       even if y > 0, otherwise odd
             13                       odd if y > 0, otherwise even
             14                       even if x > 0, otherwise odd
             15                       odd if x > 0, otherwise even

            NOTE: Blank entries in the table above indicate that the
                 description would be complicated and probably not of
                 much interest.

    The C language method of modulus and integer division is:

          config("quomod", 2)
          config("quo", 2)
          config("mod", 2)

    This dependence on rnd is consistent with quo(x, y, rnd) and
    appr(x, y, rnd) in that for any real x and y and any integer rnd,

          x = y * quo(x, y, rnd) + mod(x, y, rnd).
          mod(x, y, rnd) = x - appr(x, y, rnd)

    If y and rnd are fixed and mod(x, y, rnd) is to be considered as
    a canonical residue of x % y, bits 1 and 3 of rnd should be
    zero: if 0 <= rnd < 32, it is only for rnd = 0, 1, 4, 5, 16, 17,
    20, or 21, that the set of possible values for mod(x, y, rnd)
    form an interval of length y, and for any x1, x2,

          mod(x1, y, rnd) = mod(x2, y, rnd)

    is equivalent to:

          x1 is congruent to x2 modulo y.

    This is particularly relevant when working with the ring of
    integers modulo an integer y.

EXAMPLE
    ; print mod(11,5,0), mod(11,5,1), mod(-11,5,2), mod(-11,-5,3)
    1 -4 -1 4

    ; print mod(12.5,5,16), mod(12.5,5,17), mod(12.5,5,24), mod(-7.5,-5,24)
```

```
    2.5 -2.5 2.5 2.5

    ; A = list(11,13,17,23,29)
    ; print mod(A,10,0)

    list (5 elements, 5 nonzero):
      [[0]] = 1
      [[1]] = 3
      [[2]] = 7
      [[3]] = 3
      [[4]] = 9
```

LIMITS

```
    none
```

LINK LIBRARY

```
    void modvalue(VALUE *x, VALUE *y, VALUE *rnd, VALUE *result)
    NUMBER *qmod(NUMBER *y, NUMBER *y, long rnd)
```

SEE ALSO

```
    quo, quomod, //, %
```

modify - modify a list or matrix by changing the values of its elements

```
SYNOPSIS
    modify(A, fname)

TYPES
    A       lvalue with list, matrix or objectvalue
    fname   string, the name of a user-defined function

    return  null value if successful, otherwise an error value

DESCRIPTION
    The value of each element of A is replaced by the value at that
    value of the user-defined function with name fname. Thus,
    modify(A, "f") has the effect of

      for (i = 0; i < size(A); i++) A[[i]] = f(A[[i]]);

    An error value is returned if A is not of acceptable type, if A has
    no-change protection, or if there is no user-defined function with
    name fname. The assignments are executed even if the protection
    status of some elements A[[i]] would normally prevent the
    assignment of f(A[[i]]) to A[[i]]. The modified elements retain
    whatever kinds of protection they had as well as gaining any
    other kinds of protection in the values returned by the function.

    To obtain a modified copy of A without changing values in A,
    one may use

      Amod = A; modify(A, fname)

    or more simply

      modify(Amod = A, fname).

EXAMPLE
    ; define f(x) = x^2
    ; A = list(2,4,6)
    ; modify(A, "f")
    ; print A

    list (3 elements, 3 nonzero):
        [[0]] = 4
        [[1]] = 16
        [[3]] = 36

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    makelist
```

name - return name of some kinds of structure

```
SYNOPSIS
    name(val)

TYPES
    val           any

    return  string or null value

DESCRIPTION
    If val is a named block or open file stream, name(val) returns the
    name associated with val.  Otherwise the null value is returned.

    Since the name associated with a file stream is that used when the stream
    was opened, different names may refer to the same file, e.g. "foo"
    and "./foo".

EXAMPLE
    ; A = blk("alpha");
    ; name(A)
      "alpha"

    ; f = fopen("/tmp/beta", "w")
    ; name(f)
      "/tmp/beta"

    ; name(files(0))
      "(stdin)"

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    blk, fopen
```

near - compare nearness of two numbers with a standard

```
SYNOPSIS
    near(x, y [,eps])

TYPES
    x       real
    y       real
    eps           real, defaults to epsilon()

    return  -1, 0 or 1

DESCRIPTION
    Returns:
      -1    if abs(x - y) < abs(eps)
      0     if abs(x - y) = abs(eps)
      1     if abs(x - y) > abs(eps)

EXAMPLE
    ; print near(22/7, 3.15, .01), near(22/7, 3.15, .005)
    -1 1

LIMITS
    eps >= 0

LINK LIBRARY
    FLAG qnear(NUMBER *x, NUMBER *y, NUMBER *eps)

SEE ALSO
    epsilon, abs
```

newerror - create or recall a described error-value

```
SYNOPSIS
    newerror([str])

TYPES
    str           string

    return   error-value

DESCRIPTION
    If str is not "" and has not earlier been used as an argument for
    this function, newerror(str) creates a new described error-value so
    that any future use of newerror(str) with the same str will return
    the same error-value.

    If x = newerror(str), both strerror(x) and strerror(iserro(x)) will
    return str and iserror(x) will return the error code value of the
    new error.

    The null cases newerror() and newerror("") are equivalent to
    newerror("???").

EXAMPLE
    Note that by default, errmax() is 0 so unless errmax() is
    increased you will get:

    ; ba = newerror("curds n' whey");
    Error 20000 caused errcount to exceed errmax

    ; errmax(errcount()+5)
          0
    ; e1 = newerror("triangle side length <= 0")
    ; iserror(e1)
          20000
    ; error(20000)
          Error 20000
    ; strerror(error(20000))
          "triangle side length <= 0"
    ; strerror(e1);
          "triangle side length <= 0"
    ; strerror(error(iserror(e1)))
          "triangle side length <= 0"

    ; define area(a,b,c) {
    ;;      local s;
    ;;      if (!(a > 0) || !(b > 0) || !(c > 0)) return e1;
    ;;      s = (a + b + c)/2;
    ;;      if (s <= a || s <= b || s <= c) return newerror("Non-triangle sides");
    ;;      return sqrt(s * (s - a) * (s - b) * (s - c));
    ;; }
    "area" defined

    ; A = area(8,2,5);
    ; if (iserror(A)) print strerror(A) : ":", iserror(A);
    Non-triangle sides: 20001
```

```
    ; A = area(-3,4,5)
    ; if (iserror(A)) print strerror(A) : ":", iserror(A);
    triangle side length <= 0: 20000
```

LIMITS
    The number of new described error-values is not to exceed 12767.

LINK LIBRARY
    none

SEE ALSO
    errmax, errcount, error, strerror, iserror, errno, errorcodes,
    stoponerror

nextcand - next candidate for primeness

```
SYNOPSIS
    nextcand(n [,count [, skip [, residue [,modulus]]]])

TYPES
    n       integer
    count   integer with absolute value less than 2^24, defaults to 1
    skip    integer. defaults to 1
    residue integer, defaults to 0
    modulus integer, defaults to 1

    return  integer

DESCRIPTION
    If modulus is nonzero, nextcand(n, count, skip, residue, modulus)
    returns the least integer i greater than abs(n) expressible as
    residue + k * modulus, where k is an integer, for which
    ptest(i,count,skip) == 1, or if there is no such integer, zero.

    If abs(n) < 2^32, count >= 0, and the returned value i is not zero, then
    i is definitely prime.  If count is not zero and the returned
    value i is greater than 2^32, then i is probably prime, particularly
    if abs(count) > 1.

    If skip == 0, and abs(n) >= 2^32 or count < 0, the probabilistic test with
    random bases is used so that if n is composite the
    probability that it passes ptest() is less than 4^-abs(count).

    If skip == 1 (the default value), the bases used in the probabilistic
    test are the first abs(count) primes 2, 3, 5, ...

    For other values of skip, the bases used in the probabilistic tests
    are the abs(count) consecutive integers, skip, skip + 1, skip + 2, ...

    In any case, if the integer returned by nextcand() is not zero,
    all integers between abs(n) and that integer are composite.

    If modulus is zero, the value returned is that of residue if this is
    greater than abs(n) and ptest(residue,count,skip) = 1. Otherwise
    zero is returned.

RUNTIME
    The runtime for v = nextcand(n, ...) will depend strongly on the
    number and nature of the integers between n and v.      If this number
    is reasonably large the size of count is largely irrelevant as the
    compositeness of the numbers between n and v will usually be
    determined by the test for small prime factors or one pseudoprime
    test with some base b.  If N > 1, count should be positive so that
    candidates divisible by small primes will be passed over quickly.

    On the average for random n with large word-count N, the runtime seems
    to be roughly K/N^3 some constant K.

EXAMPLE
    ; print nextcand(50), nextcand(112140,-2), nextcand(112140,-3)
    53 112141 112153
```

```
    ; print nextcand(100,1,1,1,6), nextcand(100,1,1,-1,6)
    103 101

    ; print nextcand(100,1,1,2,6), nextcand(100,1,1,303,202)
    1 101

    ; print nextcand(2e60, 1, 1, 31, 1e30)
    2000000000000000000000000000053000000000000000000000000000031
```

LIMITS
    none

LINK LIBRARY
```
    int znextcand(ZVALUE n, long count, long skip, ZVALUE res, ZVALUE mod,
      ZVALUE *cand)
```

SEE ALSO
    factor, isprime, lfactor, nextprime, prevcand, prevprime,
    pfact, pix, ptest

nextprime - nearest prime greater than specified number

```
SYNOPSIS
    nextprime(n [,err])

TYPES
    n       real
    err           integer

    return  positive integer or err

DESCRIPTION
    If n is an integer less than 2^32, nextprime(n) returns the
    first prime greater than n.

    If n >= 2^32 or n is fractional, nextprime(n, err) returns the value
    of err.

    Other cases cause a runtime error.

EXAMPLE
    ; print nextprime(10), nextprime(100), nextprime(1e6)
    11 101 1000003

    ; print nextprime(3/2,-99), nextprime(2^32-1,-99), nextprime(2^32,-99)
    -99 4294967311 -99

LIMITS
    n <= 2^32

LINK LIBRARY
    FULL znprime(ZVALUE z)

SEE ALSO
    factor, isprime, lfactor, nextcand, prevcand, prevprime,
    pfact, pix, ptest
```

norm - calculate a norm of a value

```
SYNOPSIS
    norm(x)

TYPES
    If x is an object of type xx, the function xx_norm has to have been
    defined; what this does will be determined by the definition.

    For non-object x:

    x       number (real or complex)

    return  real

DESCRIPTION
    For real x, norm(x) returns:

       x^2.

    For complex x, norm(x) returns:

      re(x)^2 + im(x)^2.

EXAMPLE
    ; print norm(3.4), norm(-3.4), norm(3 + 4i), norm(4 - 5i)
    11.56 11.56 25 41

LIMITS
    none

LINK LIBRARY
    void normvalue(VALUE *x, VALUE *result)

SEE ALSO
    cmp, epsilon, hypot, abs, near, obj
```

null - null value

```
SYNOPSIS
    null([v_1, v_2,...])

TYPES
    v_1, v_2,...  any
    return        null

DESCRIPTION
    After evaluating in order any arguments it may have, null(...)
    returns the null value.  This is a particular value, different from
    all other types; it is the only value v for which isnull(v) returns
    TRUE.  The null value tests as FALSE in conditions, and normally
    delivers no output in print statements, except that when a list or
    matrix is printed, null elements are printed as "NULL".

    A few builtin functions may return the null value, e.g.
    printf(...) returns the null value; if L = list(), then both
    pop(L) and remove(L) return the null value; when successful,
    file-handling functions like fclose(fs), fflush(fs), fputs(fs, ...)
    return the null value when successful (when they fail they return an
    error-value).  User-defined functions where execution ends
    without a "return" statement or with "return ;" return the null
    value.

    Missing expressions in argument lists are assigned the null value.
    For example, after
            define f(a,b,c) = ...
    calling
            f(1,2)
    is as if c == null().  Similarly, f(1,,2) is as if b == null().
    (Note that this does not occur in initialization lists; missing
    expressions there indicate no change.)

    The null value may be used as an argument in some operations, e.g.
    if v == null(), then for any x, x + v returns x.

    When calc is used interactively, a function that returns the null value
    causes no printed output and does not change the "oldvalue".  Thus,
    null(config("mode", "frac")) may be used to change the output mode
    without printing the current mode or changing the stored oldvalue.

EXAMPLE
    ; L = list(-1,0,1,2);
    ; while (!isnull(x = pop(L)) print x,; print
    -1 0 1 2

    ; printf("%d %d %d\n", 2, , 3);
    2  3

    ; L = list(,1,,2,)
    ; print L

    list (5 elements, 5 nonzero):
      [[0]] = NULL
      [[1]] = 1
```

```
        [[2]] = NULL
        [[3]] = 2
        [[4]] = NULL

    ; a = 27
    ; null(pi = pi(1e-1000))
    ; .
      27
```

LIMITS

    The number of arguments is not to exceed 1024.

LINK LIBRARY

    none

SEE ALSO

    isnull, test

num - numerator of a real number

```
SYNOPSIS
    num(x)

TYPES
    x       real

    return  integer

DESCRIPTION
    For real x, den(x) returns the denominator of x.  In calc,
    real values are actually rational values.  Each calc real
    value can be uniquely expressed as:

      n / d

    where:

      n and d are integers
      gcd(n,d) == 1
      d > 0

    If x = n/x, then den(x) == n.

EXAMPLE
    ; print num(7), num(-1.25), num(121/33)
    7 -5 11

LIMITS
    none

LINK LIBRARY
    NUMBER *qnum(NUMBER *x)

SEE ALSO
    den
```

. - oldvalue

```
SYNOPSIS
    .  (with no adjacent letters or digits or _ or .)

TYPES
    return  any

DESCRIPTION
    The "old value" is essentially a global variable with identifier "."
    which at top level when directly from a file or keyboard
    is automatically assigned the saved value for a line
    of statements when evaluation of that line is completed and this saved
    value is not null.  A line of statements is normally completed by a
    '\n' not within a block bounded by braces or an expression bounded by
    parentheses.

    Disabling of saving by calling saveval(0) causes lines to return a
    null value and . then becomes in effect a global variable whose
    value may be changed by assignments and operations like ++ and --.

    A null value may be assigned to . by . = null() or free(.).

EXAMPLE
    ; saveval(1);
    ; a = 2
    ; .
          2
    ; . += 3; b = . + 4
    ; print ., b
    9 9
    ; . += 3; b = . + 4; null()
    ; print ., b
    12 16
    ; list(a, b, a + b)

    list (3 elements, 3 nonzero):
          [[0]] = 2
          [[1]] = 16
          [[2]] = 18

    ; saveval(0)
    ; print pop(.), .[[1]]
    2 18

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    saveval
```

ord - return integer corresponding to character value

```
SYNOPSIS
    ord(c)

TYPES
    c       string

    return  int

DESCRIPTION
    Return the integer value of the first character of a string.

EXAMPLE
    ; print ord("DBell"), ord("chongo"), ord("/\../\"), ord("!")
    68 99 47 33

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    char
```

perm - permutation number

```
SYNOPSIS
    perm(x, y)

TYPES
    x        int
    y        int

    return   int

DESCRIPTION
    Return the permutation number P(x,y) which is defined as:

              x!
           --------
            (x-y)!

    This function computes the number of permutations in which y things
    may be chosen from x items where order in which they are chosen matters.

EXAMPLE
    ; print perm(7,3), perm(7,4), perm(7,5), perm(3,0), perm(0,0)
    210 840 2520 3 0

    ; print perm(2^31+1,3)
    9903520314283042197045510144

LIMITS
    x >= y >= 0
    y < 2^24

LINK LIBRARY
    void zperm(NUMBER x, y, *ret)

SEE ALSO
    comb, fact, randperm
```

pfact - product of primes up to specified integer

```
SYNOPSIS
    pfact(n)

TYPES
    n       nonnegative integer

    return  positive integer

DESCRIPTION
    Returns the product of primes p_i for which p_i <= n.

EXAMPLE
    ; for (i = 0; i <= 16; i++) print pfact(i),:;
    1 1 2 6 6 30 30 210 210 210 210 2310 2310 30030 30030 30030 30030

LIMITS
    n < 2^24

LINK LIBRARY
    NUMBER *qpfact(NUMBER *n)
    void zpfact(ZVALUE z, ZVALUE *dest)

SEE ALSO
    factor, isprime, lfactor, nextcand, nextprime, prevcand, prevprime,
    pix, ptest, fact, lcmfact
```

pi - evaluate pi to specified accuracy

```
SYNOPSIS
    pi([eps])

TYPES
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Returns a multiple of eps differing from the true value of pi by
    less than 0.75 eps, and in nearly all cases by less than 0.5 eps.

EXAMPLE
    ; print pi(1e-5), pi(1e-10), pi(1e-15), pi(1e-20)
    3.14159 3.1415926536 3.141592653589793 3.14159265358979323846

LIMITS
    eps > 0

LINK LIBRARY
    NUMBER *qpi(NUMBER *eps)

SEE ALSO
    atan2
```

pix - number of primes not exceeding specified number

```
SYNOPSIS
    pix(n [,err])

TYPES
    n       real
    err           integer

    return  nonnegative integer, or err

DESCRIPTION
    If n is fractional or n >= 2^32, pix(n) causes an error,
    pix(n, err) returns the value of err.

    If n is an integer < 2^32, pix(n) returns the number of primes
    (2, 3, 5, ...) less or equal to n.

EXAMPLE
    ; for (i = 0; i <= 20; i++) print pix(i),:;
    0 0 1 2 2 3 3 4 4 4 4 5 5 6 6 6 6 7 7 8 8

    ; print pix(100), pix(1000), pix(1e4), pix(1e5), pix(1e6)
    25 168 1229 9592 78498

    ; print pix(2^32 - 1, -1), pix(2^32, -1)
    203280221 -1

LIMITS
    none

LINK LIBRARY
    long zpix(ZVALUE z)
    FULL pix(FULL x)

SEE ALSO
    factor, isprime, lfactor, nextcand, nextprime, prevcand, prevprime,
    pfact, ptest
```

places - return number of "decimal" places in a fractional part

```
SYNOPSIS
    places(x [,b])

TYPES
    x       real
    b       integer >= 2, defaults to 10

    return  integer

DESCRIPTION
    Returns the least non-negative integer n for which  b^n * x  is an
    integer, or -1 if there is no such integer.

    places(x,b) = 0 if and only if x is an integer.

    If omitted, b is assumed to be 10.  If given, b must be an
    integer > 1.

    places(x,b) = n > 0 if and only if the fractional part of abs(x)
    has a finite base-b "decimal" representation with n digits of which
    the last digit is nonzero.  This occurs if and only if every prime
    factor of den(x) is a factor of b.

EXAMPLE
    ; print places(3), places(0.0123), places(3.70), places(1e-10), places(3/7)
    0 4 1 10 -1

    ; print places(0.0123, 2), places(.625, 2), places(.625, 8)
    -1 3 1

LIMITS
    b > 1

LINK LIBRARY
    long qplaces(NUMBER *q, ZVALUE base)

SEE ALSO
    digit, digits
```

pmod - integral power of an integer modulo a specified integer

```
SYNOPSIS
    pmod(x, n, md)

TYPES
    x       integer
    n       nonnegative integer
    md            integer

    return  integer

DESCRIPTION
    pmod(x, n, md) returns the integer value of the canonical residue
    of x^n modulo md, where the set of canonical residues is determined
    by md and bits 0, 2, and 4 of config("mod") (other bits are ignored).

    If md is zero, the value is simply x^n.

    For nonzero md, the canonical residues v modulo md are as follows:

            config("mod")       md > 0           md < 0

                0       0 < v < md     md < v < 0
                1       -md < v < 0     0 < v < -md
                4       0 < v < md      0 < v < -md
                5       -md < v < 0    md < v < 0
               16     -md/2 < v <= md/2   md/2 <= v < -md/2
               17    -md/2 <= v < md/2     md/2 < v <= -md/2
               20     -md/2 < v <= md/2    md/2 < v <= -md/2
               21    -md/2 <= v < md/2    md/2 <= v < -md/2

EXAMPLE
    ; c = config("mod",0)
    ; print pmod(2,3,10), pmod(2,5,10), pmod(2,3,-10), pod(2,5,-10)
    8 2 -2 -8

    ; c = config("mod",16)
    ; print pmod(2,3,10), pmod(2,5,10), pmod(2,3,-10), pmod(2,5,-10)
    -2 2 -2 2

LIMITS
    none

LINK LIBRARY
    NUMBER *qpowermod(NUMBER *x, NUMBER *n, NUMBER *md)

SEE ALSO
    mod, minv
```

polar - specify a complex number by modulus (radius) and argument (angle)

```
SYNOPSIS
    polar(r, t [, eps])

TYPES
    r       real
    t       real
    eps             nonzero real, defaults to epsilon()

    return  number (real or complex)

DESCRIPTION
    Returns the real or complex number with real and imaginary parts
    multiples of epps nearest or next to nearest to r * cos(t) and
    r * sin(t) respectively.  The error for each part will be less
    than 0.75 * abs(eps), but usually less than 0.5 * abs(eps).

EXAMPLE
    ; print polar(2, 0), polar(1, 2, 1e-5), polar(1, 2, 1e-10)
    2 -.41615+.9093i -.4161468365+.90929742682i

    ; pi = pi(1e-10); eps = 1e-5
    ; print polar(2, pi/4, eps), polar(2, pi/2, eps), polar(2, 3*pi/4, eps)
    1.41421+1.41421i 2i -1.414215+1.41421i

LIMITS
    none

LINK LIBRARY
    COMPLEX *c_polar(NUMBER *r, NUMBER *t, NUMBER *eps);

SEE ALSO
    abs, arg, re, im
```

poly - evaluate a polynomial

```
SYNOPSIS
    poly(a, b, ..., x)
    poly(clist, x, y, ...)

TYPES
    For first case:

    a, b, ...     Arithmetic

    x       Arithmetic

    return  Depends on argument types

    For second case:

    clist   List of coefficients

    x, y, ...     Coefficients

    return  Depends on argument types

    Here an arithmetic type is one for which the required + and *
    operations are defined, e.g. real or complex numbers or square
    matrices of the same size.      A coefficient is either of arithmetic
    type or a list of coefficients.

DESCRIPTION
    If the first argument is not a list, and the necessary operations are
    defined:

      poly(a_0, a_1, ..., a_n, x)

     returns the value of:

      a_n + (a_n-1 + ... + (a_1 + a_0 * x) * x ...) * x

    If the coefficients a_0, a_1, ..., a_n and x are elements of a
    commutative ring, e.g. if the coefficients and x are real or complex
    numbers, this is the value of the polynomial:

      a_0 * x^n + a_1 * x^(n-1) + ... + a_(n-1) * x + a_n.

    For other structures (e.g. if addition is not commutative),
    the order of operations may be relevant.

    In particular:

      poly(a, x) returns the value of a.

      poly(a, b, x) returns the value of b + a * x

      poly(a, b, c, x) returns the value of c + (b + a * x) * x


    If the first argument is a list as if defined by:
```

```
      clist = list(a_0, a_1, ..., a_n)

    and the coefficients a_i and x are are of arithmetic type,
    poly(clist, x) returns:

      a_0 + (a_1 + (a_2 + ... + a_n * x) * x)

    which for a commutative ring, expands to:

      a_0 + a_1 * x + ... + a_n * x^n.

    If clist is the empty list, the value returned is the number 0.

    Note that the order of the coefficients for the list case is the
    reverse of that for the non-list case.

    If one or more elements of clist is a list and there are more than
    one arithmetic arguments x, y, ..., the coefficient corresponding
    to such an element is the value of poly for that list and the next
    argument in x, y, ...  For example:

      poly(list(list(a,b,c), list(d,e), f), x, y)

    returns:

       (a + b * y + c * y^2) + (d + e * y) * x + f * x^2.

    Arguments in excess of those required for clist are ignored, e.g.:

      poly(list(1,2,3), x, y)

    returns the same as poly(list(1,2,3), x).  If the number of
    arguments is less than greatest depth of lists in clist, the
    "missing" arguments are deemed to be zero.  E.g.:

      poly(list(list(1,2), list(3,4), 5), x)

    returns the same as:

      poly(list(1, 3, 5), x).

    If in the clist case, one or more of x, y, ... is a list, the
    arguments to be applied to the polynomial are the successive
    non-list values in the list or sublists.  For example, if the x_i
    are not lists:

      poly(clist, list(x_0, x_1), x_2, list(list(x_3, x_4), x_5))

    returns the same as:

      poly(clist, x_0, x_1, x_2, x_3, x_4, x_5).

EXAMPLE
    ; print poly(2, 3, 5, 7), poly(list(5, 3, 2), 7), 5 + 3 * 7 + 2 * 7^2
    124 124 124

    ; mat A[2,2] = {1,2,3,4}
    ; mat I[2,2] = {1,0,0,1}
```

```
    print poly(2 * I, 3 * I, 5 * I, A)

    mat [2,2] (4 elements, 4 nonzero)
      [0,0] = 22
      [0,1] = 26
      [1,0] = 39
      [1,1] = 61

    ; P = list(list(0,0,1), list(0,2), 3); x = 4; y = 5
    ; print poly(P,x,y), poly(P, list(x,y)), y^2 + 2 * y * x + 3 * x^2
    113 113 113

LIMITS
    The number of arguments is not to exceed 1024

LINK LIBRARY
    BOOL evalpoly(LIST *clist, LISTELEM *x, VALUE *result);

SEE ALSO
    list
```

pop - pop a value from front of a list

```
SYNOPSIS
    pop(lst)

TYPES
    lst           list, &list

    return  any

DESCRIPTION
     This function removes index 0 and returns it.

     This function is equivalent to calling delete(lst, 0).

EXAMPLE
    ; lst = list(2,"three")

    list (2 elements, 2 nonzero):
      [[0]] = 2
      [[1]] = "three"

    ; pop(lst)
         2
    ; print lst

    list (1 elements, 1 nonzero):
      [[0]] = "three"

    ; pop(lst)
         "three"
    ; print lst
    list (0 elements, 0 nonzero)
    ; pop(lst)
    ; print lst
    list (0 elements, 0 nonzero)

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, islist, push, remove, rsearch, search,
     select, size
```

popcnt - number of bit that match a given value

```
SYNOPSIS
    popcnt(x [,bitval])

TYPES
    x       number (real or integer)
    bitval  0 or 1

    return  number

DESCRIPTION
    Count the number of bits in abs(x) that match bitval.  The default
    bitval is 1 which counts the number of 1 bits.

    The popcnt function is equivalent to #x when x is an integer.

EXAMPLE
    ; print popcnt(32767), popcnt(3/2), popcnt(pi(),0), popcnt(pi(),1)
    15 3 69 65

    ; print popcnt(randombit(128), 0), popcnt(randombit(128), 1)
    61 64

LIMITS
    none

LINK LIBRARY
    long zpopcnt(ZVALUE z, int bitval)

SEE ALSO
    none
```

# #

```
SYNOPSIS
    #!/usr/local/src/cmd/calc/calc -q -f
    # x
    x # y
    ## comment

TYPES
    x, y          integer or real

    return        integer (uniary operator case)
                  integer or real (binary operator case)

DESCRIPTION
    The pound sign or sharp sign "#" has special meaning in calc.

    As a uniary operator:

      # value

    returns the number of 1 bits, or pop-count of the absolute value of
    the numerator (abs(num(value))).  Therefore when x is a non-negative
    integer , # x is the pop-count of x.  And thus when x is a negative
    integer, # x returns the pop-count of abs(x).  And in the general
    case when x is a real, # x returns the pop-count of abs(num(x)).

    As a binary operator:

      x # y

    returns abs(x-y), the absolute value of the difference.

    When two or more pound signs in a row start a comment:

      ## this is a comment
      ### this is also a comment
      print "this will print";  ## but this will not because it is a comment

    A pound sign followed by a bang also starts a comment:

      #! strictly speaking, this is a comment
      print "this is correct but not recommended" #! acts like ##

    On POSIX / Un*X / Linux / BSD conforming systems, when an executable
    starts with the two bytes # and !, the remainder of the 1st line
    (up to an operating system imposed limit) is taken to be the path
    to the shell (plus shell arguments) that is to be executed.  The
    kernel appends the filename of the executable as a final argument
    to the shell.

    For example, of an executable file contains:

      #!/usr/local/src/cmd/calc/calc -q -f
      /* NOTE: The #! above must start in column 1 of the 1st line */
      /*     The 1st line must end with -f */
      ## Single # shell comments don't work, use two or more
```

```
  print "2+2 =", 2+2;
```

When the above file it is executed by the kernel, it will print:

```
  2+2 = 4
```

Such files are known to calc as cscripts. See "help cscript" for examples.

It is suggested that the -q be added to the first line to disable the reading of the startup scripts. It is not mandatory.

The last argument of the first line must be -f without the filename because the kernel will supply the cscript filename as a final argument. The final -f also implies -s. See "help usage" for more information.

EXAMPLE

```
; #3
  2
; #3.5
  3
; 4 # 5
  1
; 5 # 4
  1

; pi() # exp(1)
  0.4233108251307480031
; exp(1) # pi()
  0.4233108251307480031

; ## this is a comment
```

LIMITS

none

LINK LIBRARY

none

SEE ALSO

cscript, unexpected, usage

power - evaluate a numerical power to specified accuracy

```
SYNOPSIS
    power(x, y, [, eps])

TYPES
    x       number
    x       number
    eps             nonzero real, defaults to epsilon()

    return  number

DESCRIPTION
    For real or complex x and y, power(x,y,eps) returns a real or
    complex number for which the real and imaginary parts are multiples
    of epsilon differing from the true real and imaginary parts of the
    principal y-th power of x by less than 0.75 * abs(eps), usually by
    less than 0.5 * abs(eps).  If the principal y-th power of x is a
    multiple of eps, it will be returned exactly.

    If y is a large integer but x^y is not large, and accuracy
    represented by eps is all that is required, power(x,y,eps) may be
    considerably faster than appr(x^y, eps, 24), the difference between
    the two results being probably at most abs(eps).

EXAMPLE
    ; print power(1.2345, 10, 1e-5), power(1.2345, 10, 1e-10)
    8.22074 8.2207405646

    ; print power(1+3i, 3, 1e-5), power(1 + 3i, 2+ 1i, 1e-5)
    -26-18i -2.50593-1.39445i

    ; print power(1+ 1e-30, 1e30, 1e-20)
    2.71828182845904523536

    ; print power(1i, 1i, 1e-20)
    .20787957635076190855

    ; print power(exp(1, 1e-20), pi(1e-20) * 1i/2, 1e-20)
    1i

LIMITS
    If x = 0, y in power(x,y,eps) has to have positive real part,
    except in the case of y = 0; power(0, 0, eps) is the multiple of
    eps nearest 1.

    eps > 0

LINK LIBRARY
    void powervalue(VALUE *x, VALUE *y, VALUE *eps, VALUE *result)
    NUMBER *qpower(NUMBER *x, NUMBER *y, NUMBER *eps)
    COMPLEX *c_power(COMPLEX *x, COMPLEX *y, NUMBER *eps)

SEE ALSO
    root
```

prevcand - previous candidate for primeness

```
SYNOPSIS
    prevcand(n [,count [, skip [, residue [, modulus]]]])

TYPES
    n       integer
    count   integer with absolute value less than 2^24, defaults to 1
    skip    integer, defaults to 1
    residue integer, defaults to 0
    modulus integer, defaults to 1

    return  integer

DESCRIPTION
    The sign of n is ignored; in the following it is assumed that n >= 0.

    prevcand(n, count, skip, residue, modulus) returns the greatest
    positive integer i less than abs(n) expressible as
    residue + k * modulus, where k is an integer, for which
    ptest(i,count,skip) == 1, or if there is no such integer i, zero.

    If n < 2^32, count >= 0, and the returned value i is not zero, i is
    definitely prime.  If n > 2^32, count != 0, and i is not zero,
    i is probably prime, particularly if abs(count) is greater than 1.

    With the default argument values, if n > 2, prevcand(n) returns the a
    probably prime integer i less than n such that every integer
    between i and n is composite.

    If skip == 0, the bases used in the probabilistic test are random
    and then the probability that the returned value is composite is
    less than 1/4^abs(count).

    If skip == 1 (the default value) the bases used in the probabilistic
    test are the first abs(count) primes 2, 3, 5, ...

    For other values of skip, the bases used are the abs(count) consecutive
    integer skip, skip + 1, ...

    If modulus = 0, the only values that may be returned are zero and the
    value of residue.  The latter is returned if it is positive, less
    than n, and is such that ptest(residue, count, skip) = 1.

RUNTIME
    The runtime for v = prevcand(n, ...) will depend strongly on the
    number and nature of the integers between n and v.      If this number
    is reasonably large the size of count is largely irrelevant as the
    compositeness of the numbers between n and v will usually be
    determined by the test for small prime factors or one pseudoprime
    test with some base b.  If N > 1, count should be positive so that
    candidates divisible by small primes will be passed over quickly.

    On the average for random n with large word-count N, the runtime
    seems to be between roughly K/N^3 some constant K.

EXAMPLE
```

```
    ; print prevcand(50), prevcand(2), prevcand(125,-1), prevcand(125,-2)
    47 1 113 113

    ; print prevcand(100,1,1,1,6), prevcand(100,1,1,-1,6)
    97 89

    ; print prevcand(100,1,1,2,6), prevcand(100,1,1,4,6),
    2 0

    ; print prevcand(100,1,1,53,0), prevcand(100,1,1,53,106)
    53 53

    ; print prevcand(125,1,3), prevcand(125,-1,3), prevcand(125,-2,3)
    113 121 113

    ; print prevcand(2e60, 1, 1, 31, 1e30)
    1999999999999999999999999999914000000000000000000000000000031
```

```
LIMITS
    none

LINK LIBRARY
    int zprevcand(ZVALUE n, long count, long skip, ZVALUE res, ZVALUE mod,
      ZVALUE *cand)

SEE ALSO
    factor, isprime, lfactor, nextcand, nextprime, prevprime,
    pfact, pix, ptest
```

prevprime - nearest prime less than specified number

```
SYNOPSIS
    prevprime(n [,err])

TYPES
    n       real
    err           integer

    return  positive integer or err

DESCRIPTION
    If n is an integer and 2 < n < 2^32, prevprime(n) returns the
    nearest prime less than n.

    If n <= 2 or >= 2^32 or n is fractional, prevprime(n, err)
    returns the value of err.

    Other cases cause a runtime error.

EXAMPLE
    ; print prevprime(10), prevprime(100), prevprime(1e6)
    7 97 999983

    ; print prevprime(2,-99), prevprime(2^32,-99)
    -99 -99

    ; print prevprime(2)
    pprime arg 1 is <= 2

LIMITS
    2 < n < 2^32

LINK LIBRARY
    FULL zpprime(ZVALUE z)

SEE ALSO
    factor, isprime, lfactor, nextcand, nextprime, prevcand,
    pfact, pix, ptest
```

printf - formatted print to standard output

```
SYNOPSIS
    printf(fmt, x_1, x_2, ...)

TYPES
    fmt                 string
    x_1, x_2, ... any

    return        null

DESCRIPTION
    The function printf() is similar to the C function with the same name.
    The most significant difference is that there is no requirement
    that the types of values of the arguments x_i match the
    corresponding format specifier in fmt.  Thus, whatver the
    format specifier, a number is printed as a number, a string as
    a string, a list as a list, a matrix as a matrix, an xx-object
    as an xx-object, etc.

    Except when a '%' is encountered, characters of the string fmt are
    printed in succession to the standard output.  Occurrence of
    a '%' indicates the intention to build a format specifier.
    This is completed by a succession of characters as follows:

          an optional '-'
          zero or more decimal digits
          an optional '. followed by zero or more decimal deigits
          an optional 'l'
          one of the letters: d, s, c, f, e, r, o, x, b,

    If any other character is read, the '%' and any characters
    between '%' and the character are ignored and no specifier is
    formed.  E.g. "%+f" prints as if only "f" were read; "% 10s"
    prints as "10s", "%X" prints as "X", "%%" prints as "%".

    The characters in a format specifier are interpreted as follows:

          a minus sign sets the right-pad flag;
          the first group of digits determines the width w;
                w = 0 if there are no digits
          a dot indicates the precision is to be read from the
                following digits; if there is no dot,
                precision = config("display").
          any digits following the . determines the precision p;
                p = 0 if there are no digits
          any 'l' before the final letter is ignored
          the final letter determines the mode as follows:

          d, s, c       current config("mode")
          f       real (decimal, floating point)
          e       exponential
          r       fractional
          o       octal
          x       hexadecimal
          b       binary
```

```
    If the number of arguments after fmt is less than the
    number of format specifiers in fmt, the "missing" arguments
    may be taken to be null values - these contribute nothing to the
    output; if a positive width w has been specified, the effect is
    to produce w spaces, e.g. printf("abc%6dxyz") prints "abc      xyz".

    If i <= the number of specifiers in fmt, the value of argument x_i
    is printed in the format specified by the i-th specifier.
    If a positive width w has been specified and normal printing of x_i
    does not include a '\n' character, what is printed will if necessary
    be padded with spaces so that the length of the printed output
    is at least the w.  Note that control
    characters like '\t', '\b' each count as one character.  If
    the 'right-pad' flag has been set, the padding is on the right;
    otherwise it is on the left.

    If i > the number of specifiers in fmt, the value of argument x_i
    does not contribute to the printing.  However, as all arguments
    are evaluated before printing occurs, side-effects of the
    evaluation of x_i might affect the result.

    If the i-th specifier is of numerical type, any numbers in the
    printing of x_i will be printed in the specified format, unless
    this is modified by the printing procedure for x_i's type.    Any
    specified precision will be ignored except for floating-point
    mode.

    In the case of floating-point (f) format the precision determines
    the maximum number of decimal places to be
    displayed.    Other aspects of this printing may be affected by the
    configuration parameters "outround", "tilde", "fullzero", "leadzero".

EXAMPLE
    ; c = config("epsilon", 1e-6); c = config("display", 6);
    ; c = config("tilde", 1); c = config("outround", 0);
    ; c = config("fullzero", 0);
    ; fmt = "%f,%10f,%-10f,%10.4f,%.4f,%.f.\n";
    ; a = sqrt(3);
    ; printf(fmt,a,a,a,a,a,a);
    1.732051,  1.732051,1.732051  ,   ~1.7320,~1.7320,~1.

    ; c = config("tilde", 0); c = config("outround",24);
    ; c = config("fullzero", 1);
    ; printf(fmt,a,a,a,a,a,a);
    1.732051,  1.732051,1.732051  ,    1.7321,1.7321,2.

    ; mat A[4] = {sqrt(2), 3/7, "undefined", null()};
    ; printf("%f%r",A,A);
    mat [4] (4 elements, 4 nonzero):
      [0] = 1.414214
      [1] = .428571
      [2] = "undefined"
      [3] = NULL

    mat [4] (4 elements, 4 nonzero):
      [0] = 707107/500000
      [1] = 3/7
      [2] = "undefined"
      [3] = NULL
```

```
LIMITS
    The number of arguments of printf() is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    fprintf, strprintf, print
```

protect - read or adjust protect status for a variable or named block

```
SYNOPSIS
    protect(var [, N [, depth])
    protect(nblk [, N [, depth]])

TYPES
    var           lvalue
    nblk    named block
    N       integer, abs(N) < 65536
    depth   nonnegative integer

    return  null value

DESCRIPTION
    The protection status of the association of an lvalue A with
    its value is represented by a nonnegative integer less than 2^16.
    The current value sts of this status is returned by protect(A).
    Each nonzero bit of the low eight bits of sts corresponds to a
    builtin kind of protection as follows:

      bit value   protection

            1     no assign to A
            2     no change of A by assignment
            4     no change of type value of A
            8     no error value for A
           16     no copy to A
           32     no relocation for A or its elements
           64     no assign from A
          128     no copy from A

    For example, A having protection status 65 = 1 + 64 prevents
    execution of assignments of the forms A = expression and V = A
    where V is an lvalue. Attempting either of these assignments
    will return an error value and leave the value of A or V unchanged.

    Initally, when created, any lvalue A has zero status corresponding
    to "no protection". This may be restored at any time by protect(A, 0).

    If N is positive and A does not already have the protection
    corresponding to a nonzero bit of N, protect(A, N) adds that
    protection to the status of A. For example, if protect(A) is 65,
    protect(A, 17) will add the no-copy-to protection so that the
    new protection status of A will be 81 = 1 + 16 + 64.

    Similarly, if N is negative and A has the protection corresponding
    to a nonzero bit of abs(N), protect(A,N) will remove that kind of
    protection. For example if protect(A) = 65, then protect(A, -17)
    will remove the no-assign-to protection and the new value of
    protect(A) will be 64. Note that protect(A, -65535) has the same
    effect as protect(A, 0).

    For the purposes of this function, the depth of a global or local
    variable is zero; if A has depth d and the value of A is a list,
    matrix, object or association, its elements have depth d + 1.
```

For example, after:

```
; obj point {x,y}
; X = mat[3] = {1, list(2,3), mat[2] = {obj point, obj point} }
```

X has depth 0; X[0], X[1] and X[2] have depth 1; X[1][0], X[1][1], X[2][0] and X[2][1] have depth 2; X[2][0].x, X[2][0].y, X[2][1].x and X[2][1].y have depth 3. For any lvalue A, protect(A, N, depth) applies protect(A, N) to A and to all elements, elements of elements, etc., up tothe stated depth. In the above example, protect(X, 20, 2) gives no-type-change and no-copy-to protection to 8 of the listed lvalues, but not to the components of the objects X[2][0] and X[2][1]; With any d >= 3, protect(X, 20, d) would give that protection the 12 listed lvalues.

If B is a variable with positive status and assignment of B to A is permitted, execution of the assignment A = B adds to the protections of A all protections of B that A does not already have. Except when the value returned is the result of the evqluation of an lvalue with positive status, calc's builtin operators and functions return values with zero protection status. For example, whatever the protection statuses of X and Y, X + sqrt(Y) will have zero status, but t ? X : Y may have nonzero status. The list, matrix, object or association returned by the use of list, mat, obj or assoc will have zero status, but any element specified by an lvalue will receive its status; e.g. after L = list(X, X^2) , protect(L[0]) equals protect(X) and both protect(L) and protect(L[1]) are zero.

Users may define functions that return values with positive status, e.g.

```
; define noassigntovalue(x) {protect(x,1); return x};
; S = noassigntovalue(42);
```

will result in S having the value 42 and no-assign-to protection. By using a backquote with a variable as argument, an even simpler function:

```
; define noassignto(x) = protect(x, 1);
```

gives no-assign-to protection to the variable; i.e. noassignto(`A) achieves the same as protect(A,1).

In the brief descriptions above of builtin kinds of protectiopn, "assign" refers to use of '=' as in A = expr to assign the value of expr to A, and in A = {..., expr, ...} to assign the value of expr to some component of A, and to the assignments implicit in quomod(x, y, A, B), and pre or post ++ or --. Swapping of lvalues is prevented if either value has no-assign-to or no-assign-from protection. (Swapping of octets is prevented if at least one of them is in a string or block with no-copy-to or no-copy-from protection.)

"Copying" refers to initialization using {...} and to the operations copy and blkcpy as applied to strings, blocks, lists and matrices. Although A = {..., expr, ...) assigns the value of expr to an elment of A, it is also regarded as copying to A. Thus, initialization of A may be prevented either by giving no-copy-to protection to A or no-assignment=to protection to the elements of A. Assignments to and from characters or octets in strings or blocks are also regarded as

copying to or from the string or block. For example, after
A = "abc", protect(A,16) would prevent the assignment A[0] = 'x'.
(Note that in this example, A[0] is not an lvalue in the sense
normally understood - the only values it can have are nonnegative
integers less than 256. The only kinds of protection relevant to an
octet are the no-copy-to, no-copy-from and no-change protections of
the string or block in which the octet resides.)

The no-relocate protection applies to lists and blocks. For lists,
it refers to the operations push, pop, append, remove, insert and
delete. For example, if A = list(2,3,5,7), protect(A, 32) will
prevent any change in the content or size of the list.
No-relocation protection of a block prevents reallocation of the
memory used by a block and the freeing of a named block, For example,
if a block B has maxsize 256, then after:

```
  ; protect(B, 32);
```

copy(A, B) will fail if the copying would cause size(B) to equal or
exceed 256; if B is a named block, blkfree(B) will not be permitted.

The elements of the list returned by list(...) will initially have zero
protection status except when an argument is an lvalue with positive
status, in which case the corresponding element will receive that
status. E.g., L = list(A,B) will result in L[0] having status
protect(A) and L[1] having status protect(B). L itself will have
the status L had before the assignment. There is a similar copying
of protection status when "= { ... }" initialization is used for
matrices, lists or objects. For example, except when A or B has
no-assign-from protection, M = mat [2] = {A,B} or mat M[2] = {A,B}
will result in M[0] and M[1] having statuses protect(A) and
protect(B) respectively. (If A or B has no-assign-from protection,
mat[2] = {A,B} returns an error value.)

Although M = mat[2] = {...} and mat M[2] = {...} do the same thing,
these are different from (M = mat[2]) = {...} and (mat M[3]) = {...}.
In the former pair of statements, the result of mat[2] = {...} is being
assigned to M. In the latter statments, a matrix with zero elements is
being assigned to M and then that matrix is being "reinitialized". Both
will fail if M has no-asssign-to protection, but only the latter
would be prevented by M having no-copy-to protection.

When the functions which mave move elements like of sort, reverse,
swap, insert, pop, remove, push and append. are evaluated, the
protection statuses move with the values, e.g. if among the values
and elements involved, there is just one with value 42, then the
lvalue to which the value 42 is moved will get the status the lvalue
with value 42 had before the evaluation of the function. This is
relevant to evaluation of expressions like A = sort(A),
append(A, pop(A)), insert(A,2,B,C). Note that when pop(A) is first
evaluated it is located on the stack calc uses for calculating
expressions rather than being the value of an lvalue. With an
explicit assignment like X = pop(A) or the implied assignment in
append(A, pop(A)), it becomes the value of an lvalue.

Users may use higher bits values for other kinds of protection or
simply to store information about an lvalue and its current value.
For example 1024 might be used to indicate that the lvalue is always
to have positive value. Then code for evaluating a function might

include lines like

```
; if (protect(A) & 1024 && B <= 0) {
;; return newerror("Prohibited assignment");
;; }
; A = B;
```

When an operation forbidden by a particular bit in the protection status of A is attempted, an error value is created but unless this causes errcount to exceed errmax, the only immediate evidence for the error might be the incrementing of errcount. Sometimes the failure causes the return of the error value; e.g. swap(A,B) if not permitted returns an appropriate error value rather than the null value. If the value of A is a number and A has no-type-change protection, A = "abc" returns an error value. The error-number of the most recent error value is returned by errno(), a string describing it by strerror().

A named block may be referred to by using the blocks() or blk() functions, or by assigning it to a variable A and then using either A or *A. In the latter cases, protect(A, sts) sets the status for the variable A; protect(*A, sts) assigns the status for the named block. For example, protect(*A,16) will prevent any copying to the named block; protect(A,16) will prevent any copying to the named block only when it is referred to by A.

EXAMPLE

```
; A = 27
; protect(A,1)
; A = 45
; A
      27
; strerror()
      "No-assign-to destination for assign"
; protect(A,64)
; protect(A)
      65
; X = A
; X
      0
; strerror()
      "No-assign-from source for assign"
; protect(A,-1)
; protect(A)
      64
; protect(A,4)
; protect(A)
      68
; A = "abc"
; A
      27
; strerror()
      "No-type-change destination for assign"
; B = 45
; swap(A,B)
      Error 10372
; strerror()
      "No-assign-to-or-from argument for swap"
```

```
    ; protect(A,-64)
    ; protect(A)
            4
    ; swap(A,B)
    ; A
            45
    ; B
            27

    ; A = mat[4] = {1,2,3,4}
    ; B = list(5,6,7,8)
    ; protect(A,16)
    ; copy(B,A)
            Error 10226
    ; strerror()
            "No-copy-to destination variable"

    ; A = list(1,2,3)
    ; protect(A,32)
    ; append(A,4)
            Error 10402
    ; strerror()
            "No-relocate for list append"

    ; A = blk(0,5)
    ; copy("abc", A)
    ; copy("de",A)
            Error 10229
    ; strerror()
            "No-relocate destination variable"

    ; A = blk("alpha") = {1,2,3,4,5}
    ; protect(A,0)
    ; protect(*A, 16)
    ; copy("abc", A)
            Error 10228
    ; strerror()
            "No-copy-to destination named block"
```

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    assign, copy, blk, error, errno, strerror

ptest - probabilistic test of primality

```
SYNOPSIS
    ptest(n [,count [,skip]])

TYPES
    n       integer
    count   integer with absolute value less than 2^24, defaults to 1
    skip    integer, defaults to 1

    return  0 or 1

DESCRIPTION
    In ptest(n, ...) the sign of n is ignored; here we assume n >= 0.

    ptest(n, count, skip) always returns 1 if n is prime; equivalently,
    if 0 is returned then n is not prime.

    If n is even, 1 is returned only if n = 2.

    If count >= 0 and n < 2^32, ptest(n,...) essentially calls isprime(n)
    and returns 1 only if n is prime.

    If count >= 0, n > 2^32, and n is divisible by a prime <= 101, then
    ptest(n,...) returns 0.

    If count is zero, and none of the above cases have resulted in 0 being
    returned, 1 is returned.

    In other cases (which includes all cases with count < 0), tests are
    made for abs(count) bases b:  if n - 1 = 2^s * m where m is odd,
    the test for base b of possible primality is passed if b is a
    multiple of n or b^m = 1 (mod n) or b^(2^j * m) = n - 1 (mod n) for
    some j where 0 <= j < s; integers that pass the test are called
    strong probable primes for the base b; composite integers that pass
    the test are called strong pseudoprimes for the base b; Since
    the test for base b depends on b % n, and bases 0, 1 and n - 1 are
    trivial (n is always a strong probable prime for these bases), it
    is sufficient to consider 1 < b < n - 1.

    The bases for ptest(n, count, skip) are selected as follows:

      skip = 0:   random in [2, n-2]
      skip = 1:   successive primes 2, 3, 5, ...
                  not exceeding min(n, 65536)
      otherwise:  successive integers skip, skip + 1, ...,
                  skip+abs(count)-1

    In particular, if m > 0, ptest(n, -m, 2) == 1 shows that n is either
    prime or a strong pseudoprime for all positive integer bases <= m + 1.
    If 1 < b < n - 1, ptest(n, -1, b) == 1 if and only if n is
    a strong pseudoprime for the base b.

    For the random case (skip = 0), the probability that any one test
    with random base b will return 1 if n is composite is always
    less than 1/4, so with count = k, the probability is less
    than 1/4^k.    For most values of n the probability is much
```

smaller, possible zero.

RUNTIME

If n is composite, ptest(n, 1, skip) is usually faster than ptest(n, -1, skip), much faster if n is divisible by a small prime. If n is prime, ptest(n, -1, skip) is usually faster than ptest(n, 1, skip), possibly much faster if n < 2^32, only slightly faster if n > 2^32.

If n is a large prime (say 50 or more decimal digits), the runtime for ptest(n, count, skip) will usually be roughly K * abs(count) * ln(n)^3 for some constant K. For composite n with highbit(n) = N, the compositeness is detected quickly if n is divisible by a small prime and count >= 0; otherwise, if count is not zero, usually only one test is required to establish compositeness, so the runtime will probably be about K * N^3. For some rare values of composite n, high values of count may be required to establish the compositeness.

If the word-count for n is less than conf("redc2"), REDC algorithms are used in evaluating ptest(n, count, skip) when small-factor cases have been eliminated. For small word-counts (say < 10) this may more than double the speed of evaluation compared with the standard algorithms.

EXAMPLE

```
; print ptest(103^3 * 3931, 0), ptest(4294967291,0)
1 1
```

In the first example, the first argument > 2^32; in the second the first argument is the largest prime less than 2^32.

```
; print ptest(121,-1,2), ptest(121,-1,3), ptest(121,-2,2)
0 1 0
```

121 is the smallest strong pseudoprime to the base 3.

```
; x = 151 * 751 * 28351
; print x, ptest(x,-4,1), ptest(x,-5,1)
3215031751 1 0
```

The integer x in this example is the smallest positive integer that is a strong pseudoprime to each of the first four primes 2, 3, 5, 7, but not to base 11. The probability that ptest(x,-1,0) will return 1 is about .23.

```
; for (i = 0; i < 11; i++) print ptest(91,-1,0),:; print;
0 0 0 1 0 0 0 0 0 0 1
```

The results for this example depend on the state of the random number generator; the expectation is that 1 will occur twice.

```
; a = 24444516448431392447461 * 48889032896862784894921;
; print ptest(a,11,1), ptest(a,12,1), ptest(a,20,2), ptest(a,21,2)
1 0 1 0
```

These results show that a is a strong pseudoprime for all 11 prime bases less than or equal to 31, and for all positive integer bases less than or equal to 21, but not for the bases 37 and 22. The

```
    probability that ptest(a,-1,0) (or ptest(a,1,0)) will return 1 is
    about 0.19.

LIMITS
    none

LINK LIBRARY
    BOOL qprimetest(NUMBER *n, NUMBER *count, NUMBER *skip)
    BOOL zprimetest(ZVALUE n, long count, long skip)

SEE ALSO
    factor, isprime, lfactor, nextcand, nextprime, prevcand, prevprime,
    pfact, pix
```

push - push one or more values into the front of a list

```
SYNOPSIS
    push(x, y_0, y_1, ...)

TYPES
    x       lvalue whose value is a list
    y_0, ...      any

    return  null value

DESCRIPTION
     If after evaluation of y_0, y_1, ..., x is a list with
     contents (x_0, x_1, ...), then after push(x, y_0, y_1, ..., y_n-1),
     x has contents (y_n-1, ..., y_1, y_0, x_0, x_1, ...), i.e. the
     values of y_0, y_1, ... are inserted in succession at the beginning
     of the list.

     This function is equivalent to insert(x, 0, y_n-1, ..., y_1, y_0).

EXAMPLE
    ; A = list(2,"three")
    ; print A

    list (2 elements, 2 nonzero):
      [[0]] = 2
      [[1]] = "three"

    ; push(A, 4i, 7^2)
    ; print A

    list (4 elements, 4 nonzero):
      [[0]] = 49
      [[1]] = 4i
      [[2]] = 2
      [[3]] = "three"

    ; push (A, pop(A), pop(A))
    ; print A

    list (4 elements, 4 nonzero):
      [[0]] = 4i
      [[1]] = 49
      [[2]] = 2
      [[3]] = "three"

LIMITS
    push() can have at most 100 arguments

LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, islist, pop, remove, rsearch, search,
     select, size
```

quo - compute integer quotient of a value by a real number

```
SYNOPSIS
    quo(x, y, rnd) or x // y

TYPES
    If x is a matrix or list, the returned value is a matrix or list v of
    the same structure for which each element v[[i]] = quo(x[[i]], y, rnd).

    If x is an xx-object or x is not an object and y is an xx-object,
    this function calls the user-defined function xx_quo(x, y, rnd);
    the types of arguments and returned value are as required by the
    definition of xx_quo().

    If neither x nor y is an object, and x is not a matrix or list:

    x        number (real or complex)
    y        real
    rnd            integer, defaults to config("quo")

    return   number

DESCRIPTION
    If x is real or complex and y is zero, quo(x, y, rnd) returns zero.

    If x is complex, quo(x, y, rnd) returns
             quo(re(x), y, rnd) + quo(im(x), y, rnd) * 1i.

    In the following it is assumed that x is real and y is nonzero.

    If x/y is an integer quo(x, y, rnd) returns x/y.

    If x is real, y nonzero and x/y is not an integer, x // y returns
   one of the two integers v for which abs(x/y - v) < 1.  Which
   integer is returned is controlled by rnd as follows:

            rnd   sign of x/y - v         Description of rounding

            0           +                 down, towards minus infinity
            1           -                 up, towards infinity
            2         sgn(x/y)            towards zero
            3        -sgn(x/y)            from zero
            4         sgn(y)
            5        -sgn(y)
            6         sgn(x)
            7        -sgn(x)
            8                       to nearest even integer
            9                       to nearest odd integer
             10                     even if x/y > 0, otherwise odd
             11                     odd if x/y > 0, otherwise even
             12                     even if y > 0, otherwise odd
             13                     odd if y > 0, otherwise even
             14                     even if x > 0, otherwise odd
             15                     odd if x > 0, otherwise even

           16-31                    to nearest integer when this
                                    is uniquely determined;
```

```
                                  otherwise, when x/y is a
                                  half-integer, as if
                                  rnd replaced by rnd & 15

            NOTE: Blank entries in the table above indicate that the
                  description would be complicated and probably not of
                  much interest.

    The C language method of modulus and integer division is:

          config("quomod", 2)
          config("quo", 2)
          config("mod", 2)
```

EXAMPLE

```
    print quo(11,5,0), quo(11,5,1), quo(-11,5,2), quo(-11,-5,3)
    2 3 -2 3

    print quo(12.5,5,16), quo(12.5,5,17), quo(12.5,5,24), quo(-7.5,-5,24)
    2 3 2 2
```

LIMITS

    none

LINK LIBRARY

```
    void quovalue(VALUE *x, VALUE *y, VALUE *rnd, VALUE *result)
    NUMBER *qquo(NUMBER *x, NUMBER *y, long rnd)
```

SEE ALSO

    mod, quomod, //, %

quomod - assign quotient and remainder to two lvalues

```
SYNOPSIS
    quomod(x, y, Q, R [, rnd])

TYPES
    x       real
    y       real
    Q       null-or-real-valued lvalue with assign-to permission
    R       null-or-real-valued lvalue with assign-to permission
    rnd           nonnegative integer, defaults to config("quomod")

    return  0 or 1

DESCRIPTION
    If y is nonzero and x/y is an integer q, this function assigns
    q to Q and zero to R, and returns zero.

    If y is zero, zero is assigned to Q, x to R and 0 or 1 returned
    according as x is zero or nonzero.

    In the remaining case, y nonzero and x/y not an intger, there
    are two pairs (q,r) for which x = q * y + r, q is an integer, and
    abs(r) < abs(y).  Depending on the low 5 bits of rnd, the q and r
    of one of these pairs will be assigned to Q and R respectively,
    and the number 1 returned. The effects of rnd can be described in
    terms of the way q is related to x/y, e.g. by rounding down,
    rounding towards zero, rounding to a nearest integeri, etc. or
    by properties of the remainder r, e.g. positive, negative,
    smallest, etc. The effects of the most commonly used values of
    rnd are described in the following table:

       rnd         q                     r

      0   round down. q = floor(x/y)         same sign as y
      1   round up, q = ceil(x/y)      opposite sign to y
      2   round to zero, q = int(x/y)        same sign as x, r = y * frac(x/y)
      3   round from zero              oppsite sign to x
      4                          positive
      5                          negative
      6                          same sign as x/y
      7                          opposite sigh to x/y

      8   to nearest even
      9   to nearest odd

    For 16 <= rnd < 32, the rounding is to the nearest integer and r
    is the smallest (in absolute value) remainder except when x/y is
    halfway between consecutive integers, in which case the rounding
    is as given by the 4 low bits of rnd. Using rnd = 24 gives the
    cpmmonly used principle of rounding: round to the nearest integer,
    but take the even integer when there are two equally close integers.

    For more detail on the effects of rnd for values other than those
    listed above, see "help quo" and "help mod".

    In all cases, the values assigned to Q and R by quomod(x, y, Q, R, rnd)
```

```
    are the same as those given by Q = quo(x,y,rnd), R = mod(x,y,rnd).
    If config("quo") == rnd, Q is also given by quo(x,y) or x // y.
    If config("mod") == rnd, R is also given by mod(x,y) or x % y.

    The rounding used by the C language for x / y  and x % y corresponds
    to rnd = 2.

    An error values is returned and the values of Q and R are not changed
    if Q and R are not both lvalues, or if the current value of any
    argument is not as specified above, or if Q or R has no-assign-to
    prptection, e.g. after protect(Q,1).


EXAMPLE
    ; global u, v;
    ; global mat M[2];
    ; print quomod(13,5,u,v), u, v, quomod(15.6,5.2,M[0],M[1]), M[0], M[1];
    1 2 3 0 3 0

    ; A = assoc();
    ; print quomod(13, 5, A[1], A[2]), A[1], A[2]
    ; 1 2 3

    ; print quomod(10, -3, u, v), u, v;
    1 -4 -2
    ; print quomod(10, -3, u, v, 0), u, v;
    1 -4 -2
    ; print quomod(10, -3, u, v, 1), u, v;
    1 -3 1
    ; print quomod(10, -3, u, v, 2), u, v;
    1 -3 1
    ; print quomod(-10, -3, u, v, 2), u, v;
    1 3 -1

LIMITS
    rnd < 2^31

LINK LIBRARY
    BOOL qquomod(NUMBER *q1, NUMBER *q2, NUMBER **quo, NUMBER **mod)

SEE ALSO
    //,      %, quo, mod, floor. ceil, int. frac
```

rand - subtractive 100 shuffle pseudo-random number generator

```
SYNOPSIS
    rand([[min, ] beyond])

TYPES
    min           integer
    beyond  integer

    return  integer

DESCRIPTION
    Generate a pseudo-random number using an subtractive 100 shuffle generator.
    We return a pseudo-random number over the half closed interval:

      [min,beyond)      ((min <= return < beyond))

    By default, min is 0 and beyond is 2^64.

    The shuffle method is fast and serves as a fairly good standard
    pseudo-random generator.   If you need a fast generator and do not
    need a cryptographically strong one, this generator is likely to do
    the job.  Casual direct use of the shuffle generator may be
    acceptable.    For a much higher quality cryptographically strong
    (but slower) generator use the Blum-Blum-Shub generator (see the
    random help page).

    Other arg forms:

      rand()            Same as rand(0, 2^64)
      rand(beyond)      Same as rand(0, beyond)

    The rand generator generates the highest order bit first.  Thus:

      rand(256)

    will produce the save value as:

      (rand(8) << 5) + rand(32)

    when seeded with the same seed.

    The rand generator has two distinct parts, the subtractive 100 method
    and the shuffle method.  The subtractive 100 method is described in:

      "The Art of Computer Programming - Seminumerical Algorithms",
      Vol 2, 3rd edition (1998), Section 3.6, page 186, formula (2).

    The "use only the first 100 our of every 1009" is described in
    Knuth's "The Art of Computer Programming - Seminumerical Algorithms",
    Vol 2, 3rd edition (1998), Section 3.6, page 188".

    The period and other properties of the subtractive 100 method
    make it very useful to 'seed' other generators.

    The shuffle method is feed values by the subtractive 100 method.
    The shuffle method is described in:
```

```
  "The Art of Computer Programming - Seminumerical Algorithms",
  Vol 2, 3rd edition (1998), Section 3.2.2, page 34, Algorithm B.

The rand generator has a good period, and is fast.      It is reasonable as
generators go, though there are better ones available.  The shuffle
method has a very good period, and is fast.  It is fairly good as
generators go, particularly when it is feed reasonably random
numbers.  Because of this, we use feed values from the subtractive 100
method into the shuffle method.

The rand generator uses two internal tables:

  additive table - 100 entries of 64 bits used by the subtractive
                   100 method

  shuffle table - 256 entries of 64 bits used by the shuffle method
                  feed by the subtractive 100 method from the
                  subtractive table

The goals of this generator are:

  * all magic numbers are explained

      I (Landon Curt Noll) distrust systems with constants (magic
      numbers) and tables that have no justification (e.g.,
      DES).  I believe that I have done my best to justify all of
      the magic numbers used.

  * full documentation

      You have this source file, plus background publications,
      what more could you ask?

  * large selection of seeds

       Seeds are not limited to a small number of bits.  A seed
       may be of any size.

Most of the magic constants used by this generator ultimately are
based on the Rand book of random numbers.  The Rand book contains
10^6 decimal digits, generated by a physical process.  This book,
produced by the Rand corporation in the 1950's is considered
a standard against which other generators may be measured.

The Rand book of numbers was groups into groups of 20 digits.  The
first 100 groups < 2^64 were used to initialize the default additive
table.  The size of 20 digits was used because 2^64 is 20 digits
long.  The restriction of < 2^64 was used to prevent modulus biasing.

The shuffle table size is longer than the 100 entries recommended
by Knuth.  We use a power of 2 shuffle table length so that the
shuffle process can select a table entry from a new subtractive 100
value by extracting its low order bits.  The value 256 is convenient
in that it is the size of a byte which allows for easy extraction.

We use the upper byte of the subtractive 100 value to select the
shuffle table entry because it allows all of 64 bits to play a part
in the entry selection.  If we were to select a lower 8 bits in the
```

```
64 bit value, carries that propagate above our 8 bits would not
impact the subtractive 100 generator output.

It is 'nice' when a seed of "n" produces a 'significantly different'
sequence than a seed of "n+1".  Generators, by convention, assign
special significance to the seed of '0'.  It is an unfortunate that
people often pick small seed values, particularly when large seed
are of significance to the generators found in this file.  An internal
process called randreseed64 will effectively eliminate the human
perceptions that are noted above.

It should be noted that the purpose of randreseed64 is to scramble a
seed ONLY.    We do not care if these generators produce good random
numbers.  We only want to help eliminate the human factors & perceptions
noted above.

The randreseed64 process scrambles all 64 bit chunks of a seed, by
mapping [0,2^64) into [0,2^64).  This map is one-to-one and onto.
Mapping is performed using a linear congruence generator of the form:

  X1 <-- (a*X0 + c) % m

with the exception that:

  0 ==> 0                 (so that srand(0) acts as default)

while maintaining a 1-to-1 and onto map.

The randreseed64 constants 'a' and 'c' based on the linear
congruential generators found in:

  "The Art of Computer Programming - Seminumerical Algorithms"
  by Knuth, Vol 2, 2nd edition (1981), Section 3.6, pages 170-171.

We will select the randreseed64 multiplier 'a' such that:

  a mod 8 == 5                  (based on note iii)
  0.01*m < a < 0.99*m           (based on note iv)
  0.01*2^64 < a < 0.99*2^64
  a is prime              (help keep the generators independent)

The choice of the randreseed64 adder 'c' is considered immaterial
according (based in note v).  Knuth suggests 'c==1' or 'c==a'.  We
elect to select 'c' using the same process as we used to select
'a'.  The choice is 'immaterial' after all, and as long as:

  gcd(c, m) == 1          (based on note v)
  gcd(c, 2^64) == 1
  gcd(a, c) == 1          (adders & multipliers will be more independent)

The values 'a' and 'c for randreseed64 are taken from the Rand book
of numbers.    Because m=2^64 is 20 decimal digits long, we will
search the Rand book of numbers 20 at a time.  We will skip any of
the 100 values that were used to initialize the subtractive 100
generators.    The values obtained from the Rand book are:

  a = 6316878969928993981
  c = 1363042948800878693
```

```
    As we stated before, we must map 0 ==> 0 so that srand(0) does the
    default thing.  The randreseed64 would normally map as follows:

      0 ==> 1363042948800878693     (0 ==> c)

    To overcome this, and preserve the 1-to-1 and onto map, we force:

      0 ==> 0
      10239951819489363767 ==> 1363042948800878693

    One might object to the complexity of the seed scramble/mapping via
    the randreseed64 process.  But Calling srand(0) with the randreseed64
    process would be the same as calling srand(10239951819489363767)
    without it.    No extra security is gained or reduced by using the
    randreseed64 process.  The meaning of seeds are exchanged, but not
    lost or favored (used by more than one input seed).

    The randreseed64 process does not reduce the security of the rand
    generator.    Every seed is converted into a different unique seed.
    No seed is ignored or favored.

    The truly paranoid might suggest that my claims in the MAGIC NUMBERS
    section are a lie intended to entrap people.  Well they are not, but
    if you that paranoid why would you use a non-cryprographically strong
    pseudo-random number generator in the first place?  You would be
    better off using the random() builtin function.

    The two constants that were picked from the Rand Book of Random Numbers
    The random numbers from the Rand Book of Random Numbers can be
    verified by anyone who obtains the book.  As these numbers were
    created before I (Landon Curt Noll) was born (you can look up
    my birth record if you want), I claim to have no possible influence
    on their generation.

    There is a very slight chance that the electronic copy of the
    Rand Book of Random Numbers that I was given access to differs
    from the printed text.  I am willing to provide access to this
    electronic copy should anyone wants to compare it to the printed text.

    When using the s100 generator, one may select your own 100 subtractive
    values by calling:

         srand(mat100)

    and avoid using my magic numbers.  The randreseed64 process is NOT
    applied to the matrix values. Of course, you must pick good subtractive
    100 values yourself!

EXAMPLE
    ; print srand(0), rand(), rand(), rand()
    RAND state 2298441576805697181 3498508396312845423 5031615567549397476

    ; print rand(123), rand(123), rand(123), rand(123), rand(123), rand(123)
    106 59 99 99 25 88

    ; print rand(2,12), rand(2^50,3^50), rand(0,2), rand(-400000, 120000)
    2 658186291252503497642116 1 -324097

LIMITS
```

```
    min < beyond

LINK LIBRARY
    void zrand(long cnt, ZVALUE *res)
    void zrandrange(ZVALUE low, ZVALUE beyond, ZVALUE *res)
    long irand(long beyond)

SEE ALSO
    seed, srand, randbit, isrand, random, srandom, israndom
```

randbit - additive 55 shuffle pseudo-random number generator

```
SYNOPSIS
    randbit([x])

TYPES
    x       integer

    return  integer

DESCRIPTION
    If x > 0, randbit(x) returns a pseudo-random integer in [0, 2^x),
    i.e. the same as rand(2^x).      If the integer returned is

          b_1 * 2^(x-1) + b_2 * 2^(x-2) + ... + b_n,

    where each b_i is 0 or 1, then b_1, b_2, ..., b_n may be
    considered as a sequence of x random bits.

    If x <= 0, randbit(x) causes the random-number generator to skip
    abs(x) bits, and returns abs(x).

    If x is omitted, it is assumed to have the value of 1.

    See the rand() help page for details on the additive 55 shuffle
    pseudo-random number generator.

EXAMPLE
    ; print srand(0), randbit(20), randbit(20), randbit(20), randbit(20)
    RAND state 817647 476130 944201 822573

    ; print srand(0), randbit(-20), randbit(20), randbit(-20), randbit(20)
    RAND state 20 476130 20 822573

LIMITS
    x != 0

LINK LIBRARY
    void zrand(long cnt, ZVALUE *res)

SEE ALSO
    seed, srand, randbit, isrand, random, srandom, israndom
```

random - Blum-Blum-Shub pseudo-random number generator

```
SYNOPSIS
    random([[min, ] beyond])

TYPES
    min           integer
    beyond  integer

    return  integer

DESCRIPTION
    Generate a pseudo-random number using a Blum-Blum-Shub generator.
    We return a pseudo-random number over the half closed interval:

      [min,beyond)      ((min <= return < beyond))

    By default, min is 0 and beyond is 2^64.

    While the Blum-Blum-Shub generator is not painfully slow, it is not
    a fast generator.  For a faster, but lesser quality generator
    (non-cryptographically strong) see the additive 55 generator
    (see the rand help page).

    Other arg forms:

      random()    Same as random(0, 2^64)
      random(beyond)    Same as random(0, beyond)

    The random generator generates the highest order bit first.    Thus:

      random(256)

    will produce the save value as:

      (random(8) << 5) + random(32)

    when seeded with the same seed.

    The basic idea behind the Blum-Blum-Shub generator is to use
    the low bit bits of quadratic residues modulo a product of
    two 3 mod 4 primes.  The lowest int(log2(log2(p*q))) bits are used
    where log2() is log base 2 and p,q are two primes 3 mod 4.

    The Blum-Blum-Shub generator is described in the papers:

      Blum, Blum, and Shub, "Comparison of Two Pseudorandom Number
      Generators", in Chaum, D. et. al., "Advances in Cryptology:
      Proceedings Crypto 82", pp. 61-79, Plenum Press, 1983.

      Blum, Blum, and Shub, "A Simple Unpredictable Pseudo-Random
      Number Generator", SIAM Journal of Computing, v. 15, n. 2,
      1986, pp. 364-383.

      U. V. Vazirani and V. V. Vazirani, "Trapdoor Pseudo-Random
      Number Generators with Applications to Protocol Design",
      Proceedings of the 24th  IEEE Symposium on the Foundations
```

of Computer Science, 1983, pp. 23-30.

U. V. Vazirani and V. V. Vazirani, "Efficient and Secure Pseudo-Random Number Generation", Proceedings of the 24th IEEE Symposium on the Foundations of Computer Science, 1984, pp. 458-463.

U. V. Vazirani and V. V. Vazirani, "Efficient and Secure Pseudo-Random Number Generation", Advances in Cryptology - Proceedings of CRYPTO '84, Berlin: Springer-Verlag, 1985, pp. 193-202.

Sciences 28, pp. 270-299.

Bruce Schneier, "Applied Cryptography", John Wiley & Sons, 1st edition (1994), pp 365-366.

This generator is considered 'strong' in that it passes all polynomial-time statistical tests. The sequences produced are random in an absolutely precise way. There is absolutely no better way to predict the sequence than by tossing a coin (as with TRULY random numbers) EVEN IF YOU KNOW THE MODULUS! Furthermore, having a large chunk of output from the sequence does not help. The BITS THAT FOLLOW OR PRECEDE A SEQUENCE ARE UNPREDICTABLE!

Of course the Blum modulus should have a long period. The default Blum modulus as well as the compiled in Blum moduli have very long periods. When using your own Blum modulus, a little care is needed to avoid generators with very short periods. See the srandom() help page for information for more details.

To compromise the generator, an adversary must either factor the modulus or perform an exhaustive search just to determine the next (or previous) bit. If we make the modulus hard to factor (such as the product of two large well chosen primes) breaking the sequence could be intractable for todays computers and methods.

The Blum generator is the best generator in this package. It produces a cryptographically strong pseudo-random bit sequence. Internally, a fixed number of bits are generated after each generator iteration. Any unused bits are saved for the next call to the generator. The Blum generator is not too slow, though seeding the generator via srandom(seed,plen,qlen) can be slow. Shortcuts and pre-defined generators have been provided for this reason. Use of Blum should be more than acceptable for many applications.

The goals of this package are:

all magic numbers are explained

I distrust systems with constants (magic numbers) and tables that have no justification (e.g., DES). I believe that I have done my best to justify all of the magic numbers used.

full documentation

You have this source file, plus background publications, what more could you ask?

```
      large selection of seeds

          Seeds are not limited to a small number of bits.  A seed
          may be of any size.

      the strength of the generators may be tuned to meet the need

          By using the appropriate seed and other arguments, one may
          increase the strength of the generator to suit the need of
          the application.  One does not have just a few levels.

    For a detailed discussion on seeds, see the srandom help page.

    It should be noted that the factors of the default Blum modulus
    is given in the source.  While this does not reduce the quality
    of the generator, knowing the factors of the Blum modulus would
    help someone determine the next or previous bit when they did
    not know the seed.  If this bothers you, feel free to use one
    of the other compiled in Blum moduli or provide your own. See
    the srandom help page for details.


EXAMPLE
    ; print srandom(0), random(), random(), random()
    RANDOM state 9203168135432720454 13391974640168007611 13954330032848846793

    ; print random(123), random(123), random(123), random(123), random(123)
    22 83 66 88 67

    ; print random(2,12), random(2^50,3^50), random(0,2), random(-400000,120000)
    10 483381144668580304003305 0 -70235

LIMITS
    min < beyond

LINK LIBRARY
    void zrandom(long cnt, ZVALUE *res)
    void zrandomrange(ZVALUE low, ZVALUE beyond, ZVALUE *res)
    long irandom(long beyond)

SEE ALSO
    seed, srand, randbit, isrand, rand, srandom, israndom
```

randbit - Blum-Blum-Shub pseudo-random number generator

```
SYNOPSIS
    randombit([x])

TYPES
    x       integer

    return  integer

DESCRIPTION
    If x > 0, randombit(x) returns a pseudo-random integer in [0, 2^x),
    i.e. the same as rand(2^x).      If the integer returned is

          b_1 * 2^(x-1) + b_2 * 2^(x-2) + ... + b_n,

    where each b_i is 0 or 1, then b_1, b_2, ..., b_n may be
    considered as a sequence of x random bits.

    If x <= 0, randombit(x) causes the random-number generator to skip
    abs(x) bits, and returns abs(x).

    If x is omitted, it is assumed to have the value of 1.

    See the random() help page for details on the additive 55 shuffle
    pseudo-random number generator.

EXAMPLE
    ; print srandom(0), randombit(20), randombit(20), randombit(20)
    RANDOM state 523139 567456 693508
    ; print srandom(0), randombit(-20), randombit(20), randombit(-20)
    RANDOM state 20 567456 20

LIMITS
    x != 0

LINK LIBRARY
    void zrandom(long cnt, ZVALUE *res)

SEE ALSO
    seed, srand, randbit, israndom, rand, srandom, israndom
```

randperm - randomly permute a list or matrix

```
SYNOPSIS
    randperm(x)

TYPES
    x       list or matrix

    return  same as x

DESCRIPTION
    For a list or matrix x, randperm(x) returns a copy of x in which
    the elements have been randomly permuted.  The value of x is not
    changed.

    This function uses the rand() subtractive 100 shuffle pseudo-random
    number generator.

EXAMPLE
    ; A = list(1,2,2,3,4)
    ; randperm(A)

    list (5 elements, 5 nonzero):
        [[0]] = 4
        [[1]] = 1
        [[2]] = 2
        [[3]] = 3
        [[4]] = 2

    ; randperm(A)

    list (5 elements, 5 nonzero):
        [[0]] = 2
        [[1]] = 1
        [[2]] = 4
        [[3]] = 2
        [[4]] = 3

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    comp, fact, rand, perm
```

rcin - encode for REDC algorithms

```
SYNOPSIS
    rcin(x, m)

TYPES
    x       integer
    m       odd positive integer

    return  integer v, 0 <= v < m.

DESCRIPTION
    Let B be the base calc uses for representing integers internally
    (B = 2^16 for 32-bit machines, 2^32 for 64-bit machines) and N the
    number of words (base-B digits) in the representation of m.    Then
    rcin(x,m) returns the value of B^N * x % m, where the modulus
    operator % here gives the least nonnegative residue.

    If y = rcin(x,m), x % m may be evaluated by x % m = rcout(y, m).

    The "encoding" method of using rcmul(), rcsq(), and rcpow() for
    evaluating products, squares and powers modulo m correspond to the
    formulae:

          rcin(x * y, m) = rcmul(rcin(x,m), rcin(y,m), m);

          rcin(x^2, m) = rcsq(rcin(x,m), m);

          rcin(x^k, m) = rcpow(rcin(x,m), k, m).

    Here k is any nonnegative integer.    Using these formulae may be
    faster than direct evaluation of x * y % m, x^2 % m, x^k % m.
    Some encoding and decoding may be bypassed by formulae like:

          x * y % m = rcin(rcmul(x, y, m), m).

    If m is a divisor of B^N - h for some integer h, rcin(x,m) may be
    computed by using rcin(x,m) = h * x % m.  In particular, if
    m is a divisor of B^N - 1 and 0 <= x < m, then rcin(x,m) = x.
    For example if B = 2^16 or 2^32, this is so for m = (B^N - 1)/d
    for the divisors d = 3, 5, 15, 17, ...

RUNTIME
    The first time a particular value for m is used in rcin(x, m),
    the information required for the REDC algorithms is
    calculated and stored for future use in a table covering up to
    5 (i.e. MAXREDC) values of m.  The runtime required for this is about
    two that required for multiplying two N-word integers.

    Two algorithms are available for evaluating rcin(x, m), the one
    which is usually faster for small N is used when N <
    config("pow2"); the other is usually faster for larger N. If
    config("pow2") is set at about 200 and x has both been reduced
    modulo m, the runtime required for rcin(x, m) is at most about f
    times the runtime required for an N-word by N-word multiplication,
    where f increases from about 1.3 for N = 1 to near 2 for N > 200.
    More runtime may be required if x has to be reduced modulo m.
```

```
EXAMPLE
    Using a 64-bit machine with B = 2^32:

    ; for (i = 0; i < 9; i++) print rcin(x, 9),:; print;
    0 4 8 3 7 2 6 1 5

LIMITS
    none

LINK LIBRARY
    void zredcencode(REDC *rp, ZVALUE z1, ZVALUE *res)

SEE ALSO
   rcout, rcmul, rcsq, rcpow
```

rcmul - REDC multiplication

```
SYNOPSIS
    rcmul(x, y, m)

TYPES
    x       integer
    y       integer
    m       odd positive integer

    return  integer v, 0 <= v < m.

DESCRIPTION
    Let B be the base calc uses for representing integers internally
    (B = 2^16 for 32-bit machines, 2^32 for 64-bit machines)
    and N the number of words (base-B digits) in the representation
    of m.  Then rcmul(x,y,m) returns the value of B^-N * x * y % m,
    where the inverse implicit in B^-N is modulo m
    and the modulus operator % gives the least non-negative residue.

    The normal use of rcmul() may be said to be that of multiplying modulo m
    values encoded by rcin() and REDC functions, as in:

          rcin(x * y, m) = rcmul(rcin(x,m), rcin(y,m), m),

    or with only one factor encoded:

          x * y % m = rcmul(rcin(x,m), y, m).

RUNTIME
    If the value of m in rcmul(x,y,m) is being used for the first time
    in a REDC function, the information required for the REDC
    algorithms is calculated and stored for future use, possibly
    replacing an already stored valued, in a table covering up to 5
    (i.e. MAXREDC) values of m.      The runtime required for this is about
    two times that required for multiplying two N-word integers.

    Two algorithms are available for evaluating rcmul(x,y,m), the one
    which is usually faster for small N is used when N <
    config("redc2"); the other is usually faster for larger N. If
    config("redc2") is set at about 90 and x and y have both been
    reduced modulo m, the runtime required for rcmul(x,y,m) is at most
    about f times the runtime required for an N-word by N-word
    multiplication, where f increases from about 1.3 for N = 1 to near
    3 for N > 90.  More runtime may be required if x and y have to be
    reduced modulo m.

EXAMPLE
    Using a 64-bit machine with B = 2^32:

    ; print rcin(4 * 5, 9), rcmul(rcin(4,9), rcin(5,9), 9), rcout(8, 9);
    8 8 2

LIMITS
    none

LINK LIBRARY
```

```
    void zredcmul(REDC *rp, ZVALUE z1, ZVALUE z2, ZVALUE *res)
```

SEE ALSO

```
    rcin, rcout, rcsq, rcpow
```

rcout - decode for REDC algorithms

```
SYNOPSIS
    rcout(x, m)

TYPES
    x       integer
    m       odd positive integer

    return  integer v, 0 <= v < m.

DESCRIPTION
    Let B be the base calc uses for representing integers internally
    (B = 2^16 for 32-bit machines, 2^32 for 64-bit machines) and N the
    number of words (base-B digits) in the representation of m.    Then
    rcout(x,m) returns the value of B^-N * x % m, where the inverse
    implicit in B^-N is modulo m and the modulus operator % gives the
    least non-negative residue.      The functions rcin() and rcout() are
    inverses of each other for all x:

          rcout(rcin(x,m), m) = rcin(rcout(x,m),m) = x % m.

    The normal use of rcout() may be said to be that of decoding
    values encoded by rcin() and REDC functions, as in:

          x * y % m = rcout(rcmul(rcin(x,m),rcin(y,m),m),m),

          x^2 % m = rcout(rcsq(rcin(x,m),m),m),

          x ^ k % m = rcout(rcpow(rcin(x,m),k,m),m).

RUNTIME
    If the value of m in rcout(x,m) is being used for the first time in
    a REDC function, the information required for the REDC algorithms
    is calculated and stored for future use, possibly replacing an
    already stored valued, in a table covering up to 5 (i.e. MAXREDC)
    values of m.  The runtime required for this is about two times that
    required for multiplying two N-word integers.

    Two algorithms are available for evaluating rcout(x, m), the one
    which is usually faster for small N is used when N <
    config("pow2"); the other is usually faster for larger N. If
    config("pow2") is set at about 200, and x has been reduced modulo
    m, the runtime required for rcout(x, m) is at most about f times
    the runtime required for an N-word by N-word multiplication, where
    f increases from about 1 for N = 1 to near 2 for N >
    config("pow2").  More runtime may be required if x has to be
    reduced modulo m.

EXAMPLE
    Using a 64-bit machine with B = 2^32:

    ; for (i = 0; i < 9; i++) print rcout(i,9),:; print;
    0 7 5 3 1 8 6 4 2

LIMITS
    none
```

```
LINK LIBRARY
    void zredcdecode(REDC *rp, ZVALUE z1, ZVALUE *res)

SEE ALSO
    rcout, rcmul, rcsq, rcpow
```

rcpow - REDC powers

```
SYNOPSIS
    rcpow(x, k, m)

TYPES
    x       integer
    k       nonnegative integer
    m       odd positive integer

    return  integer v, 0 <= v < m.

DESCRIPTION
    Let B be the base calc uses for representing integers internally
    (B = 2^16 for 32-bit machines, 2^32 for 64-bit machines) and N the
    number of words (base-B digits) in the representation of m.    Then
    rcpow(x,k,m) returns the value of B^-N * (B^N * x)^k % m, w here
    the inverse implicit in B^-N is modulo m and the modulus operator %
    gives the least nonnegative residue.  Note that rcpow(x,0,m) =
    rcin(1,m), rcpow(x,1,m) = x % m; rcpow(x,2,m) = rcsq(x,m).

    The normal use of rcpow() may be said to be that of finding the
    encoded value of the k-th power of an integer modulo m:

          rcin(x^k, m) = rcpow(rcin(x,m), k, m),

    from which one gets:

          x^k % m  = rcout(rcpow(rcin(x,m), k, m), m).

    If x^k % m is to be evaluated for the same k and m and several
    values of x, it may be worth while to first evaluate:

          a = minv(rcpow(1, k, m), m);

    and use:

      x^k % m = a * rcpow(x, k, m) % m.

RUNTIME
    If the value of m in rcpow(x,k,m) is being used for the first time
    in a REDC function, the information required for the REDC
    algorithms is calculated and stored for future use, possibly
    replacing an already stored valued, in a table covering up to 5
    (i.e. MAXREDC) values of m.      The runtime required for this is about
    two times that required for multiplying two N-word integers.

    Two algorithms are available for evaluating rcpow(x,k,m), the one
    which is usually faster for small N is used when N <
    config("redc2"); the other is usually faster for larger N. If
    config("redc2") is set at about 90 and 0 <= x < m, the runtime
    required for rcpow(x,k,m) is at most about f times the runtime
    required for ilog2(k) N-word by N-word multiplications, where f
    increases from about 1.3 for N = 1 to near 4 for N > 90.  More
    runtime may be required if x has to be reduced modulo m.

EXAMPLE
```

```
    Using a 64-bit machine with B = 2^32:

    ; m = 1234567;
    ; x = 15;
    ; print rcout(rcpow((rcin(x,m), m - 1, m), m), pmod(x, m-1, m)
    783084 783084
```

```
LIMITS
    none

LINK LIBRARY
    void zredcpower(REDC *rp, ZVALUE z1, ZVALUE z2, ZVALUE *res)

SEE ALSO
    rcin, rcout, rcmul, rcsq
```

rcsq - REDC squaring

```
SYNOPSIS
    rcsq(x, m)

TYPES
    x       integer
    m       odd positive integer

    return  integer v, 0 <= v < m.

DESCRIPTION
    Let B be the base calc uses for representing integers internally
    (B = 2^16 for 32-bit machines, 2^32 for 64-bit machines)
    and N the number of words (base-B digits) in the representation
    of m.  Then rcsq(x,m) returns the value of B^-N * x^2 % m,
    where the inverse implicit in B^-N is modulo m
    and the modulus operator % gives the least non-negative residue.

    The normal use of rcsq() may be said to be that of squaring modulo m a
    value encoded by rcin() and REDC functions, as in:

          rcin(x^2, m) = rcsq(rcin(x,m), m)

    from which we get:

          x^2 % m = rcout(rcsq(rcin(x,m), m), m)

    Alternatively, x^2 % m may be evaluated usually more quickly by:

          x^2 % m = rcin(rcsq(x,m), m).

RUNTIME
    If the value of m in rcsq(x,m) is being used for the first time in
    a REDC function, the information required for the REDC algorithms
    is calculated and stored for future use, possibly replacing an
    already stored valued, in a table covering up to 5 (i.e. MAXREDC)
    values of m.  The runtime required for this is about two times that
    required for multiplying two N-word integers.

    Two algorithms are available for evaluating rcsq(x, m), the one
    which is usually faster for small N is used when N <
    config("redc2"); the other is usually faster for larger N. If
    config("redc2") is set at about 90 and 0 <= x < m, the runtime
    required for rcsq(x, m)i is at most about f times the runtime
    required for an N-word by N-word multiplication, where f increases
    from about 1.1 for N = 1 to near 2.8 for N > 90.  More runtime may
    be required if x has to be reduced modulo m.

EXAMPLE
    Using a 64-bit machine with B = 2^32:

    ; for (i = 0; i < 9; i++) print rcsq(i,9),:; print;
    0 7 1 0 4 4 0 1 7

    ; for (i = 0; i < 9; i++) print rcin((rcsq(i,9),:; print;
    0 1 4 0 7 7 0 4 1
```

```
LIMITS
    none

LINK LIBRARY
    void zredcsquare(REDC *rp, ZVALUE z1, ZVALUE *res)

SEE ALSO
    rcin, rcout, rcmul, rcpow
```

re - real part of a real or complex number

```
SYNOPSIS
    re(x)

TYPES
    x        real or complex

    return   real

DESCRIPTION
    If x = u + v * 1i where u and v are real, re(x) returns u.

EXAMPLE
    ; print re(2), re(2 + 3i), re(-4.25 - 7i)
    2 2 -4.25

LIMITS
    none

LINK LIBRARY
    COMPLEX *c_imag(COMPLEX *x)

SEE ALSO
    im
```

remove - remove the last member of a list

```
SYNOPSIS
    remove(lst)

TYPES
    lst           lvalue whose current value is a list

    return  any

DESCRIPTION
    If lst has no members, remove(lst) returns the null value and does
    not change lst.

    If lst has n members where n > 0, remove(lst) returns the value of
    lst[[n-1]] and deletes this value from the end of the lst, so that
    lst now has n - 1 members and for 0 <= i < n - 1, lst[[i]] returns
    what it would have returned before the remove operation.

EXAMPLE
    ; lst = list(2,"three")

    list (2 elements, 2 nonzero):
      [[0]] = 2
      [[1]] = "three"

    ; remove(lst)
          "three"
    ; print lst

    list (1 elements, 1 nonzero):
      [[0]] = 2

    ; remove(lst)
          2
    ; print lst
    list (0 elements, 0 nonzero)
    ; remove(lst)
    ; print lst
    list (0 elements, 0 nonzero)

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, islist, pop, push, rsearch, search,
     select, size
```

reverse - reverse a copy of a list or matrix

```
SYNOPSIS
    reverse(x)

TYPES
    x        list or matrix

    return   same type as x

DESCRIPTION
    For a list or matrix x, reverse(x) returns a list or matrix in
    which the order of the elements has been reversed.      The original
    list or matrix x is unchanged.

    In the case of matrix x, the returned value is a matrix with
    the same dimension and index limits, but the reversing is performed
    as if the matrix were a linear array.

EXAMPLE
    ; A = list(1, 7, 2, 4, 2)
    ; print reverse(A)

    list (5 elements, 5 nonzero):
        [[0]] = 2
        [[1]] = 4
        [[2]] = 2
        [[3]] = 7
        [[4]] = 1

    ; mat B[2,3] = {1,2,3,4,5,6}
    ; print reverse(B)

    mat [2,3] (6 elements, 6 nonzero):
      [0,0] = 6
      [0,1] = 5
      [0,2] = 4
      [1,0] = 3
      [1,1] = 2
      [1,2] = 1

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    join, sort
```

root - root of a number

```
SYNOPSIS
    root(x, n, [, eps])

TYPES
    x       number
    n       positive integer
    eps            nonzero real, defaults to epsilon()

    return  real number

DESCRIPTION
    For real x and positive integer n, n being odd if x is negative,
    root(x,n,eps) returns a multiple of eps differing from the
    real n-th root of x (nonnegative if x is positive) by less than
    0.75 eps, usually by less than 0.5 eps.  If the n-th root of
    x is a multiple of eps, it will be returned exactly.

    For complex x and positive integer n, or negative x with positive even
    n, root(x, n, eps) returns a real or complex numbers whose real
    and imaginary parts are multiples of eps differing from the real
    and imaginary parts of the principal n-th root of x by less than
    0.75 eps, usually by less than 0.5 eps.

    For negative x and odd n, the principal n-th root of x may be
    obtained by using power(x, 1/n, eps).

EXAMPLE
    ; print root(7, 4, 1e-5), root(7, 4, 1e-10), root(7, 4, 1e-15)
    1.62658 1.6265765617 1.626576561697786

    ; print root(1+3i, 3, 1e-5), root(1 + 3i, 3, 1e-10)
    1.34241+.59361i 1.3424077452+.5936127825i

    ; print root(-8, 3, 1e-5), root(-8, 34, 1e-5)
    -2 ~1.05853505050032399594+~.09807874962631613016i

    ; print root(1i, 100, 1e-20)
    .99987663248166059864+.01570731731182067575i

LIMITS
    n >= 0
    eps > 0

LINK LIBRARY
    void rootvalue(VALUE *x, VALUE *n, VALUE *eps, VALUE *result)
    NUMBER *qroot(NUMBER *x, NUMBER *n, NUMBER *eps)
    COMPLEX *c_root(COMPLEX *x, NUMBER *n, NUMBER *eps)

SEE ALSO
    power
```

round - round numbers to a specified number of decimal places

```
SYNOPSIS
    round(x [,plcs [, rnd]])

TYPES
    If x is a matrix or a list, round(x[[i]], ...) is to return
    a value for each element x[[i]] of x; the value returned will be
    a matrix or list with the same structure as x.

    Otherwise, if x is an object of type tt, or if x is not an object or
    number but y is an object of type tt, and the function tt_round has
    to be defined; the types for x, plcs, rnd, and the returned value, if
    any, are as required or specified in the definition of tt_round.
    In this object case, plcs and rnd default to the null value.

    For other cases:

    x       number (real or complex)
    plcs    integer, defaults to zero
    rnd           integer, defaults to config("round")

    return  number

DESCRIPTION
    For real x, round(x, plcs, rnd) returns x rounded to either
    plcs significant figures (if rnd & 32 is nonzero) or to plcs
    decimal places (if rnd & 32 is zero).  In the significant-figure
    case the rounding is to plcs - ilog10(x) - 1 decimal places.
    If the number of decimal places is n and eps = 10^-n, the
    result is the same as for appr(x, eps, rnd).  This will be
    exactly x if x is a multiple of eps; otherwise rounding occurs
    to one of the nearest multiples of eps on either side of x.   Which
    of these multiples is returned is determined by z = rnd & 31, i.e.
    the five low order bits of rnd, as follows:

          z = 0 or 4:         round down, i.e. towards minus infinity
          z = 1 or 5:         round up, i.e. towards plus infinity
          z = 2 or 6:         round towards zero
          z = 3 or 7:         round away from zero
          z = 8 or 12:  round to the nearest even multiple of eps
          z = 9 or 13:  round to the nearest odd multiple of eps
          z = 10 or 14: round to nearest even or odd multiple of eps
                            according as x > or < 0
          z = 11 or 15: round to nearest odd or even multiple of eps
                            according as x > or < 0
          z = 16 to 31: round to the nearest multiple of eps when
                            this is uniquely determined.  Otherwise
                            rounding is as if z is replaced by z - 16

    For complex x:

      The real and imaginary parts are rounded as for real x; if the
      imaginary part rounds to zero, the result is real.

    For matrix or list x:
```

```
      The returned values has element round(x[[i]], plcs, rnd) in
      the same position as x[[i]] in x.

    For object x or plcs:

      When round(x, plcs, rnd) is called, x is passed by address so may be
      changed by assignments; plcs and rnd are copied to temporary
      variables, so their values are not changed by the call.

EXAMPLES
    ; a = 7/32, b = -7/32

    ; print a, b
    .21875 -.21875

    ; print round(a,3,0), round(a,3,1), round(a,3,2), print round(a,3,3)
    .218, .219, .218, .219

    ; print round(b,3,0), round(b,3,1), round(b,3,2), print round(b,3,3)
    -.219, -.218, -.218, -.219

    ; print round(a,3,16), round(a,3,17), round(a,3,18), print round(a,3,19)
    .2188 .2188 .2188 .2188

    ; print round(a,4,16), round(a,4,17), round(a,4,18), print round(a,4,19)
    .2187 .2188 .2187 .2188

    ; print round(a,2,8), round(a,3,8), round(a,4,8), round(a,5,8)
    .22 .218 .2188 .21875

    ; print round(a,2,24), round(a,3,24), round(a,4,24), round(a,5,24)
    .22 .219 .2188 .21875

    ; c = 21875
    ; print round(c,-2,0), round(c,-2,1), round(c,-3,0), round(c,-3,16)
    21800 21900 21000 22000

    ; print round(c,2,32), round(c,2,33), round(c,2,56), round(c,4,56)
    21000 22000 22000 21880

    ; A = list(1/8, 2/8, 3/8, 4/8, 5/8, 6/8, 7/8)
    ; print round(A,2,24)

    list(7 elements, 7 nonzero):
      [[0]] = .12
      [[1]] = .25
      [[3]] = .38
      [[4]] = .5
      [[5]] = .62
      [[6]] = .75
      [[7]] = .88

LIMITS
    For non-object case:
      0 <= abs(plcs) < 2^31
      0 <= abs(rnd) < 2^31

LINK LIBRARY
    void roundvalue(VALUE *x, VALUE *plcs, VALUE *rnd, VALUE *result)
```

```
    MATRIX *matround(MATRIX *m, VALUE *plcs, VALUE *rnd);
    LIST *listround(LIST *m, VALUE *plcs, VALUE *rnd);
    NUMBER *qround(NUMBER *m, long plcs, long rnd);
```

SEE ALSO

bround, btrunc, trunc, int, appr

rsearch - reverse search for an element satisfying a specified condition

```
SYNOPSIS
    rsearch(a, b [, [c] [, [d] ] ])

TYPES
    a       matrix, list, association, or file open for reading
    b       string if a is a file, otherwise any
    c       integer, defaults to zero, size(a) or the current file-position
    d       integer, defaults to size(a) or current file-position

    return  nonnegative integer or null

DESCRIPTION

    Negative values of c and nonpositive values of d are treated as
    offsets from size(a), i.e. as if c were replaced by size(a) + c
    and d by size(a) + d.  Any such adjustment is assumed to have been
    made.

    The nature of the search depends on whether the rsearch() is called
    with or without the fourth argument d.

Four argument case:

    The search interval is as for search(a,b,c,d), i.e. the indices i
    to be examined are to satisfy c <= i < d and 0 <= i < size(a)
    for non-file a, and c <= i <= d - strlen(b), 0 <= i <= size(a) - strlen(b)
    if a is a file stream.  The difference from search(a,b,c,d) is that
    the indices i, if any, are examined in decreasing order, so that
    if a match is found, the returned integer i is the largest in the
    search interval.  The null value is returned if no match is found.

    The default value for d is size(a) for non-file cases, and the current
    file-position if a is a file.  The default for c is zero except if a
    is a file and d is an integer.

    For non-file a, the search is for a[[i]] == b, except that if
    the function accept() as been defined, it is for i such that
    accept(a[[i]], b) tests as nonzero.    Since the addresses (rather than
    values) of a[[i]] and b are passed to accept(), the values of one or
    both of a[[i]] and b may be changed during a call to rsearch().

    In the file-stream case, if strlen(b) = n, a match at file-position i
    corresponds to the n characters in the file starting at position i
    matching those of the string b.  The null value is returned if no
    match is found.   The final file position will correspond to the
    last character if a match is found, or the start (lowest) position
    of the search interval if no match is found, except that if no
    reading of characters is required (e.g. if start > end), the original
    file-position is not changed.


    Two- or Three-argument case:

    If a is not a file, the default value for c is size(a).  If a is a
```

```
    file, rsearch(a,b) = rsearch(a, b, ftell(a)), and
    rsearch(a,b,) = rsearch(a, b, size(a)).

    If a is not a file, the search starts, if at all, at the largest
    non-negative index i for which i <= c and i < size(a), and continues
    until a match a[[i]] == b is found, or if accept() has been defined,
    accept(a[[i]], b) tests as nonzero; if no such i is found and returned,
    the null value is returned.

    If a is a file, the first, if any, file-position tested has the greatest
    nonnegative position i such that i <= c and i <= size(a) - strlen(b).
    The returned value is either the first i at which a match is found or
    the null value if no match with the string b is found.  The final
    file-position will correspond to the last character of b, or the zero
    position, according as a match is found or not found.

EXAMPLE
    ; L = list(2,"three",4i)
    ; rsearch(L,"three")
         1
    ; rsearch(L,"threes")
    ; rsearch(L, 4i, 4)
    ; rsearch(L, 4i, 1)
         2

    ; f = fopen("foo", "w+")
    ; fputs(f, "This file has 28 characters.")
    ; fflush(f)
    ; rsearch(f, "ha")
      18
    ; ftell(f)
      19
    ; rsearch(f, "ha", 17)
      10
    ; rsearch(f, "ha", 9)
    ; ftell(f)
      0
    ; rsearch(f, "ha")
      18
    ; rsearch(f, "ha", 5, 500)
      18

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, islist, pop, push, remove, search,
     select, size,

     assoc, list, mat
```

runtime - CPU time used by the current process in both user and kernel modes

```
SYNOPSIS
    runtime()

TYPES
    return  nonnegative real

DESCRIPTION
    In POSIX based systems, this function will return the CPU seconds
    used by the current process in both user and kernel mode.  Time
    spent in the kernel executing system calls and time spend executing
    user code (such as performing computation on behalf of calc) are
    both included.

    On non-POSIX based systems, this function will always return 0.
    In particular, most MS windows based systems do not have the required
    POSIX system call and so this function will always return 0.

EXAMPLE
    The result for this example will depend on the speed of the CPU
    and precision of the operating CPU time accounting sub-system:

    ; t = runtime();
    ; x = ptest(2^4253-1);
    ; runtime() - t;
      1.288804

LIMITS
    On non-POSIX based systems, this function always returns 0.

LINK LIBRARY
    none

SEE ALSO
    config, ctime, systime, time, usertime
```

saveval - enable or disable saving of values

```
SYNOPSIS
    saveval(arg)

TYPES
    arg           any

    return   null value

DESCRIPTION
    When evaluation of a line of statements at top level starts, a
    "saved value" for that line is assigned the null value.  When saving
    is enabled (the initial state) and a statement being evaluated is an
    expression or a return statement, the value returned by that expression
    or statement replaces the current saved value; on completion of
    evaluation of the line, the saved value, if it is not null, updates
    the "oldvalue".

    This saving of values is enabled or disabled by calling saveval(arg)
    with an argument arg that tests as nonzero or zero,

    Whether saving is enabled or disabled does not affect the operation of
    eval(str).

EXAMPLE
    ; saveval(1);
    ; a = 27;
    .
          27
    ; saveval(0);
    ; a = 45
    .
          27
    ; saveval(1);
    ; a = 63
    .
          63

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    oldvalue, eval
```

scale - scale a number or numbers in a value by a power of 2

```
SYNOPSIS
    scale(x, n)

TYPES
    If x is an xx-object, scale(x, n) requires xx_scale() to have been
    defined; conditions on x and n and the type of value returned are
    determined by the definition of xx_scale().

    For other x:

    x       number (real or complex) or matrix
    n       integer

    return  same type as x

DESCRIPTION
    Returns the value of 2^n * x.

    scale(x,n) returns the same as x << n and x >> -n if x is an integer
    for which 2^n * x is an integer.

EXAMPLE
    ; print scale(3, 2), scale(3,1), scale(3,0), scale(3,-1), scale(3,-2)
    12 6 3 1.5 .75

LIMITS
    For non-object x, abs(n) < 2^31

LINK LIBRARY
    NUMBER *qscale(NUMBER *x, long n)
    COMPLEX *c_scale(COMPLEX *x, long n)
    MATRIX *matscale(MATRIX *x, long n)

SEE ALSO
    obj
```

search - search for an element satisfying a specified condition

```
SYNOPSIS
    search(a, b [, [c] [, [d] ] ])

TYPES
    a       matrix, list, association or file
    b       string if a is a file, otherwise any
    c       integer, defaults to zero or current file-position
    d       integer, defaults to size(a) or current file-position

    return  nonnegative integer or null value

DESCRIPTION

    Negative values of c and nonpositive values for d are treated as
    offsets from size(a), i.e. as if c were replaced by size(a) + c,
    and d by size(a) + d.  Any such adjustment is assumed in the following
    description.


    For Non-file a:

    For a matrix, list, or association a,
    search(a, b, c, d) returns, if it exists, the least index i for which
    c <= i < d,    0 <= i < size(a), and, if accept() has not been defined,
    a[[i]] == b, or if accept() has been defined, accept(a[[i]], b)
    tests as nonzero.  The null value is returned if there is no such i.

    For example, to search for the first a[[i]] > b an appropriate
    accept() function is given by:

            define accept(v,b) = (v > b);

    To restore the original behavior of search(), one may then use

            define accept(v, b) = (v == b).

    Since the addresses (rather than values) of a and b are passed,
    the values of v = x[[i]] and b may be changed during execution
    of search(a, b, c, d), e.g. if accept(v,b) has been defined by

            define accept(v,b) = (v > b ? v-- : b++);


    For a is a file-stream:

    c defaults to the current file-position if there are just two
    arguments (a,b) or if there are four arguments as in (a,b, ,d)
    where d is an integer.  Otherwise c defaults to zero.

    d defaults to the current file-position or size(a) according as
    the number of arguments (indicated by commas) is four or less
    than four.

    If a is a file, a string formed by n successive characters in a
    is considered to occur at the file position
```

```
    of the first character.  E.g. if a has the characters "123456",
    the string "345" is said to occur at position 2.

    The file is searched forwards from file-position pos = c for
    a match with b (not including the terminating '\0').
    Only characters with file-positions less than d are considered,
    so the effective interval for the first-character position pos
    for a matching string is limited by both c <= pos <= d - strlen(b)
    and 0 <= pos < size(a) - strlen(b).

    The function returns pos if a match is found, and the reading position
    for the stream after the search will then correspond to the position of
    the terminating '\0' for the string b.

    The null value is returned if no match is found.  If c, d, size(a)
    and strlen(b) are such that no match is possible, no reading of the
    file occurs and the current file-position is not changed.  In a case
    where characters are read, the final file-position will be
    min(d, size(a)) - strlen(b) + 1,
    i.e. the file will be at the first position where a match is impossible
    because the specified search region has insufficient remaining characters.

EXAMPLE
    ; L = list(2,"three",4i)
    ; search(L,"three")
          1
    ; search(L,"threes")
    ; search(L, 4i, 4)
    ; search(L, 4i, 1)
          2

    ; f = fopen("foo", "w+")
    ; fputs(f, "This file has 28 characters.")
    ; rewind(f)
    ; search(f, "ha")
      10
    ; ftell(f)
      12
    ; search(f, "ha")
      18
    ; search(f, "ha")
    ; search(f, "ha",)
      10
    ; search(f, "ha", 12)
      18
    ; search(f, "ha", -10)
      18
    ; search(f, "ha", ,)
      10
    ; search(f, "ha", 11, 19)
    ; ftell(f)
      18
    ; search(f, "ha", 11, 20)
      18
    ; search(f, "ha", 5, 500)
      10

LIMITS
    none
```

```
LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, islist, pop, push, remove, rsearch,
     select, size,

     assoc, list, mat
```

sec - trigonometric secant function

```
SYNOPSIS
    sec(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the secant of x to a multiple of eps, with error less
    in absolute value than .75 * eps.

EXAMPLE
    ; print sec(1, 1e-5), sec(1, 1e-10), sec(1, 1e-15), sec(1, 1e-20)
    1.85082 1.8508157177 1.850815717680926 1.85081571768092561791

LIMITS
    unlike sin and cos, x must be real
    eps > 0

LINK LIBRARY
    NUMBER *qsec(NUMBER *x, NUMBER *eps)

SEE ALSO
    sin, cos, tan, csc, cot, epsilon
```

sech - hyperbolic secant

```
SYNOPSIS
    sech(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the sech of x to the nearest or next to nearest multiple of
    epsilon, with absolute error less than .75 * abs(eps).

    sech(x) = 2/(exp(x) + exp(-x))

EXAMPLE
    ; print sech(1, 1e-5), sech(1, 1e-10), sech(1, 1e-15), sech(1, 1e-20)
    .64805 .6480542737 .648054273663885 .64805427366388539958

LIMITS
    unlike sin and cos, x must be real
    eps > 0

LINK LIBRARY
    NUMBER *qsech(NUMBER *x, NUMBER *eps)

SEE ALSO
    sinh, cosh, tanh, csch, coth, epsilon
```

seed - return a value that may be used to seed a pseudo-random generator

```
SYNOPSIS
    seed()

TYPES
    return  integer

DESCRIPTION
    Generate a pseudo-random seed based on a collection of system and process
    information.  The seed() builtin function returns a value:

            0 <= seed < 2^64

    IMPORTANT WARNING:

      It should be pointed out that the information collected by seed
      is almost certainly non-chaotic.  This function is likely not
      suitable for applications (such as cryptographic applications)
      where the unpredictability of seeds is critical.  For such critical
      applications, LavaRnd should be used.     See the URL:

            http://www.LavaRnd.org/

      for information about seeding a pseudo-random number generator
      (such as rand() or random()) with the cryptographic hash of the
      digitization of chaotic system.

    Given the above warning, this builtin function produces a seed that is
    suitable for most applications that desire a different pseudo-random
    sequence each time they are run.

    The return value of this builtin function should NOT be considered
    a random or pseudo-random value.  The return value should be used
    as an argument to a seed function such as srand() or srandom().

EXAMPLE
    ; print srand(seed())
    RAND state

    ; print srandom(seed())
    RAND state

LIMITS
    none

LINK LIBRARY
    NUMBER *pseudo_seed(void)

SEE ALSO
    seed, srand, randbit, isrand, rand, random, srandom, israndom
```

segment - segment from and to specified elements of a list

```
SYNOPSIS
    segment(x, y, z)

TYPES
    x       list
    y, z    int

    return  list

DESCRIPTION
    For 0 <= y < size(x) and 0 <= z < size(x), segment(x, y, z)
    returns a list for which the values of the elements are those
    of the segment of x from x[[y]] to x[[z]].  If y < z, the
    new list is in the same order as x; if y > z, the order is
    reversed.

    If y < z, x == join(head(x,y), segment(x,y,z), tail(x, size(x) - z - 1)).

EXAMPLE
    ; A = list(2, 3, 5, 7, 11)
    ; segment(A, 1, 3)

    list (3 members, 3 nonzero):
        [[0]] = 3
        [[1]] = 5
        [[2]] = 7

    ; segment(A, 3, 1)

    list (3 members, 3 nonzero):
        [[0]] = 7
        [[1]] = 5
        [[2]] = 3

LIMITS
    0 <= y < size(x)
    0 <= z < size(x)

LINK LIBRARY
    none

SEE ALSO
    head, tail
```

select - form a list by selecting element-values from a given list

```
SYNOPSIS
    select(x, y)

TYPES
    x       list
    y       string

    return  list

DESCRIPTION
    If y is to be the name of a user-defined function, select(x, y)
    returns a list whose members are the values z of elements of x
    for which the function at z tests as nonzero.
    The list x is not changed.      The order of the returned list is
    the same as in x.

EXAMPLE
    ; define f(x) = x > 5
    ; A = list(2,4,6,8,2,7)
    ; print select(A, "f")

    list (3 elements, 3 nonzero):
        [[0]] = 6
        [[1]] = 8
        [[2]] = 7

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     append, delete, insert, islist, pop, push, remove, rsearch, search,
     size
```

sgn - indicator of sign of a real or complex number

```
SYNOPSIS
    sgn(x)

TYPES
    x       real or complex

    return  -1, 0, 1                            (real)
            -1, 0, 1, -1+1i, 1i, 1+1i, -1-1i, -1i or 1-1i   (complex)

DESCRIPTION
    Return the value of cmp(a,0).

    For real x, sgn(x) returns:
      -1 if x < 0
      0  if x == 9
      1  if x > 0

    For complex, sgn(x) returns:

      sgn(re(x)) + sgn(im(x))*1i


EXAMPLE
    ; print sgn(27), sgn(1e-20), sgn(0), sgn(-45)
    1 1 0 -1

    ; print sgn(2+3i), sgn(6i), sgn(-7+4i), sgn(-6), sgn(-6-3i), sgn(-2i)
    1+1i 1i -1+1i -1 -1-1i -1i

LIMITS
    none

LINK LIBRARY
    NUMBER *qsign(NUMBER *x)

SEE ALSO
    abs
```

sha1 - Secure Hash Algorithm (SHS-1 FIPS Pub 180-1)

```
SYNOPSIS
    sha1([arg1 [, val ...]])

TYPES
    arg1    any
    val           any

    return  HASH or number

DESCRIPTION
    The sha1() builtin implements the old Secure Hash Algorithm
    (SHA).  The SHA is sometimes referenced as SHS.  The SHA
    is a 160 bit hash.

    With no args, sha1() returns the default initial SHA-1 HASH state.

    If arg1 is a HASH state and no other val args are given, then the
    HASH state is finalized and the numeric value of the hash is given.

    If arg1 is a HASH state and one or more val args are given,
    then the val args are used to modify the arg1 HASH state.
    The new arg1 HASH state is returned.

    If arg1 is not a a HASH state, then the initial HASH is
    used and modifed by arg1 and any val args supplied.      The
    return value is the new HASH state.

    The following table gives a summary of actions and return values.
    Here, assume that 'h' is a HASH state:

      sha1()                  HASH  returns initial HASH state

      sha1(h)                 number      h is put into final form and the
                              numeric value of the hash state

      sha1(x)                 HASH  modify the initial state by hashing 'x'

      sha1(sha1(), x)         HASH  the same as sha1(x)

      sha1(x, y)        HASH  the same as sha1(sha1(x), y)

      sha1(h, x, y)           HASH  modify state 'h' by 'x' and then 'y'

      sha1(sha1(h,x,y)) number      numeric value of the above call

EXAMPLE
    ; base(16)
          0xa

    ; sha1()
          sha1 hash state
    ; sha1(sha1())
          0xda39a3ee5e6b4b0d3255bfef95601890afd80709

    ; sha1("x", "y", "z") == sha1("xyz")
```

```
            1
    ; sha1("x", "y", "z") == sha1("xy")
            0

    ; sha1(sha1("this is", 7^19-8, "a composit", 3i+4.5, "hash"))
            0xc3e1b562bf45b3bcfc055ac65b5b39cdeb6a6c55

    ; x = sha1(list(1,2,3), "curds and whey", 2^21701-1, pi())
    ; x
            sha1 hash state
    ; sha1(x)
            0x988d2de4584b7536aa9a50a5749707a37affa1b5

    ; y = sha1()
    ; y = sha1(y, list(1,2,3), "curds and whey")
    ; y = sha1(y, 2^21701-1)
    ; y = sha1(y, pi())
    ; y
            sha1 hash state
    ; sha1(y)
            0x988d2de4584b7536aa9a50a5749707a37affa1b5

LIMITS
    none

LINK LIBRARY
    HASH* hash_init(int, HASH*);
    void hash_free(HASH*);
    HASH* hash_copy(HASH*);
    int hash_cmp(HASH*, HASH*);
    void hash_print(HASH*);
    ZVALUE hash_final(HASH*);
    HASH* hash_long(int, long, HASH*);
    HASH* hash_zvalue(int, ZVALUE, HASH*);
    HASH* hash_number(int, void*, HASH*);
    HASH* hash_complex(int, void*, HASH*);
    HASH* hash_str(int, char*, HASH*);
    HASH* hash_usb8(int, USB8*, int, HASH*);
    HASH* hash_value(int, void*, HASH*);

SEE ALSO
    ishash, hash
```

sin - trigonometric sine

```
SYNOPSIS
    sin(x [,eps])

TYPES
    x       number (real or complex)
    eps           nonzero real, defaults to epsilon()

    return  number

DESCRIPTION
    Calculate the sine of x to a multiple of eps with error less in
    absolute value than .75 * eps.

EXAMPLE
    ; print sin(1, 1e-5), sin(1, 1e-10), sin(1, 1e-15), sin(1, 1e-20)
    .84147 .8414709848 .841470984807896 .84147098480789650665

    ; print sin(2 + 3i, 1e-5), sin(2 + 3i, 1e-10)
    9.1545-4.16891i 9.1544991469-4.16890696i

    ; pi = pi(1e-20)
    ; print sin(pi/6, 1e-10), sin(pi/2, 1e-10), sin(pi, 1e-10)
    .5 1 0

LIMITS
    eps > 0

LINK LIBRARY
    NUMBER *qsin(NUMBER *x, NUMBER *eps)
    COMPLEX *c_sin(COMPLEX *x, NUMBER *eps)

SEE ALSO
    cos, tan, sec, csc, cot, epsilon
```

sinh - hyperbolic sine

```
SYNOPSIS
    sinh(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the sinh of x to the nearest or next to nearest multiple of
    epsilon, with absolute error less than .75 * abs(eps).

    sinh(x) = (exp(x) - exp(-x))/2

EXAMPLE
    ; print sinh(1, 1e-5), sinh(1, 1e-10), sinh(1, 1e-15), sinh(1, 1e-20)
    1.1752 1.1752011936 1.175201193643801 1.17520119364380145688

LIMITS
    unlike sin and cos, x must be real
    eps > 0

LINK LIBRARY
    NUMBER *qsinh(NUMBER *x, NUMBER *eps)

SEE ALSO
    cosh, tanh, sech, csch, coth, epsilon
```

size - number of elements in value

```
SYNOPSIS
    size(x)

TYPES
    x       any

    return  integer

DESCRIPTION
    For the different types of value x may have, size(x) is defined as follows:

      null        0
      real number 1
      complex number    1
      string            length of string (not counding the trailing \0)
      matrix            number of elements
      list        number of members
      association number of (elements, value) pairs
      object            value returned by xx_size(x) if x of type xx
      file        length of the file in octets
      rand state  1
      random state      1
      config state      1
      hash state  1
      block       numer of octets of data it currently holds
      octet       1
      named block numer of octets of data it currently holds

EXAMPLE
    ; print size(null()), size(3), size(2 - 7i), size("abc")
    0 1 1 1

    ; mat M[2,3]
    ; print size(M), size(list()), size(list(2,3,4))
    6 0 3

    ; A = assoc()
    ; A[1] = 3, A[1,2] = 6, A["three"] = 5
    ; print size(A)
    3

    ; obj point {x,y}
    ; obj point P = {4,-5}
    ; define point_size(a) = abs(a.x) + abs(a.y)
    ; print size(P)
    9

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
     list, mat, assoc, obj, sizeof, memsize
```

sizeof - number of bytes required for value

```
SYNOPSIS
    sizeof(x)

TYPES
    x       any

    return  integer

DESCRIPTION
    This is analogous to the C operator sizeof for the value only.
    It attempts to assess the number of bytes in memory used to store
    a value and all of its components.    Unlike memsize(x), this
    builtin does not include the size of the overhead.

    Unlike size(x), this builtin incldues the trailing \0 byte on the
    end of strings.

    For numeric values, sizeof(x) ignores the demoninator if 'x' is
    an integer.    For complex values, sizeof(x) ignores the imaginary
    part if 'x' is real.  Because the 0, 1 and -1 numeric values are
    shared static values, sizeof(x) reports such values as having
    0 bytes of storage.

    The number returned by sizeof(x) may be less than the actual number
    used because, for example, more memory may have been allocated for
    a string than is used: only the characters up to and including the
    first '\0' are counted in calculating the contribution of the
    string to sizeof(x).

    The number returned by sizeof(x) may be greater (and indeed
    substantially greater) than the number of bytes actually used.
    For example, after:

            a = sqrt(2);
            mat A[3] = {a, a, a};

    the numerical information for a, A[0], A[1], A[2] are stored in the
    same memory, so the memory used for A is the same as if
    its 3 elements were null values.  The value returned by
    sizeof(A) is calculated as A were defined by:

            mat A[3] = {sqrt(2), sqrt(2), sqrt(2)}.

    Similar sharing of memory occurs with literal strings.

    For associative arrays, only the value part of the name/value pair
    is counted.

    The minimum value for sizeof(x) occurs for the null and error values.

EXAMPLES
    The results for examples like these will depend to some extent on
    the system being used.  The following were for an SGI R4k machine
    in 32-bit mode:
```

```
    ; print sizeof(null()), sizeof(0), sizeof(3), sizeof(2^32 - 1), sizeof(2^32)
    8 68 68 68 72

    ; x = sqrt(2, 1e-100); print sizeof(x), sizeof(num(x)), sizeof(den(x))
    148 108 108

    ; print sizeof(list()), sizeof(list(1)), sizeof(list(1,2))
    28 104 180

    ; print sizeof(list()),sizeof(list(1)),sizeof(list(1,2)),sizeof(list(1,2,3))
    28 104 180 256

    ; mat A[] = {1}; mat B[] = {1,2}; mat C[] = {1,2,3}; mat D[100,100];
    ; print sizeof(A), sizeof(B), sizeof(C), sizeof(D)
    124 192 260 680056

    ; obj point {x,y,z}
    ; obj point P = {1,2,3}; print sizeof(P)
    274

LIMITS
    It is assumed sizeof(x) will fit into a system long integer.

LINK LIBRARY
    none

SEE ALSO
    size, fsize, strlen, digits
```

sort - sort a copy of a list or matrix

```
SYNOPSIS
    sort(x)

TYPES
    x       list or matrix

    return  same type as x

DESCRIPTION
    For a list or matrix x, sort(x) returns a list or
    matrix y of the same size as x in which the elements
    have been sorted into order completely or partly determined by
    a user-defined function precedes(a,b), or if this has not been
    defined, by a default "precedes" function which for numbers or
    strings is as equivalent to (a < b).   More detail on this default
    is given below.  For most of the following discussion
    it is assumed that calling the function precedes(a,b) does not
    change the value of either a or b.

    If x is a matrix, the matrix returned by sort(x) has the same
    dimension and index limits as x, but for the sorting, x is treated
    as a one-dimensional array indexed only by the double- bracket
    notation.     Then for both lists and matrices, if x has size n, it
    may be identified with the array:

            (x[[0]], x[[1]], ..., x[[n-1]])

    which we will here display as:

            (x_0, x_1, ..., x_n-1).

    The value y = sort(x) will similarly be identified with:

            (y_0, y_1, ..., x_n-1),

    where, for some permutation p() of the integers (0, 1, ..., n-1):

            y_p(i) = x_i.

    In the following i1 and i2 will be taken to refer to different
    indices for x, and j1 and j2 will denote p(i1) and p(i2).

    The algorithm for evaluating y = sort(x) first makes a copy of x;
    x remains unchanged, but the copy may be considered as a first
    version of y.  Successive values a in this y are read and compared
    with earlier values b using the integer-valued function precedes();
    if precedes(a,b) is nonzero, which we may consider as "true",
    a is "moved" to just before b; if precedes(a,b) is zero, i.e. "false",
    a remains after b.  Until the sorting is completed, other similar
    pairs (a,b) are compared and if and only if precedes(a,b) is true,
    a is moved to before b or b is moved to after a.  We may
    say that the intention of precedes(a,b) being nonzero is that a should
    precede b, while precedes(a,b) being zero intends that the order
    of a and b is to be as in the original x.  For any integer-valued
    precedes() function, the algorithm will return a result for sort(x),
```

```
but to guarantee fulfilment of the intentions just described,
precedes() should satisfy the conditions:

(1) For all a, b, c, precedes(a,b) implies precedes(a,c) || precedes (c,b),

(2) For all a, b, precedes(a,b) implies !precedes(b,a).

Condition (1) is equivalent to transitivity of !precedes():

(1)' For all a,b,c, !precedes(a,b) && !precedes(b,c) implies !precedes(a,c).

(1) and (2) together imply transitivity of precedes():

(3) For all a,b,c, precedes(a,b) && precedes(b,c) implies precedes(a,c).

Condition (2) expresses the obvious fact that if a and b are distinct
values in x, there is no permutation in which every occurrence of
a both precedes and follows every occurrence of b.

Condition (1) indicates that if a, b, c occur
in the order  b c a, moving a to before b or b to after a must change
the order of either a and c or c and b.

Conditions (2) and (3) together are not sufficient to ensure a
result satisfying the intentions of nonzero and zero values of
precedes() as described above.  For example, consider:

        precedes(a,b) = a is a proper divisor of b,

and x = list(4, 3, 2).  The only pair for which precedes(a,b) is
nonzero is (2,4),  but x cannot be rearranged so that 2 is before
4 without changing the order of one of the pairs (4,3) and (3,2).

If precedes() does not satisfy the antisymmetry condition (2),
i.e. there exist a, b for which both precedes(a, b)
and precedes(b, a), and if x_i1 = a, x_i2 = b, whether or
not y_j1 precedes or follows y_j2 will be determined by the
sorting algorithm by methods that are difficult to describe;
such a situation may be acceptable to a user not concerned with
the order of occurrences of a and b in the result.      To permit
this, we may now describe the role of precedes(a,b) by the rules:

  precedes(a,b) && !precedes(b,a):  a is to precede b;

  !precedes(a,b) && !precedes(b,a):  order of a and b not to be changed;

  precedes(a,b) && precedes(b,a):      order of a and b may be changed.

Under the condition (1), the result of sort(x) will accord with these rules.

Default precedes():

  If precedes(a,b) has not been defined by a define command,
  the effect is as if precedes(a,b) were determined by:

  If a and b are are not of the same type, they are ordered by

        null values < numbers < strings < objects.
```

```
  If a and b are of the same type, this type being
  null, numbers or strings, precedes(a,b) is given by (a < b).
  (If a and b are both null, they are considered to be equal, so
  a < b then returns zero.)  For null values, numbers and
  strings, this definition has the properties (1) and (2)
  discussed above.

  If a and b are both xx-objects, a < b is defined to mean
  xx_rel(a,b) < 0; such a definition does not
  necessarily give < the properties usually expected -
  transitivity and antisymmetry.      In such cases, sort(x)
  may not give the results expected by the "intentions" of
  the comparisons expressed by "a < b".

In many sorting applications, appropriate precedes() functions
have definitions equivalent to:

        define precedes(a,b) = (key(a) < key(b))

where key() maps possible values to a set totally ordered by <.
Such a precedes() function has the properties (1) and (2),
so the elements of the result returned by sort(x) will be in
nondecreasing order of their key-values, elements with equal keys
retaining the order they had in x.

For two-stage sorting where elements are first to be sorted by
key1() and elements with equal key1-values then sorted by key2(),
an appropriate precedes() function is given by:

  define precedes(a,b) = (key(a) < key(b)) ||
              (key(a) == key(b)) && (key2(a) < key2(b)).

When precedes(a.b) is called, the addresses of a and b rather
than their values are passed to the function.  This permits
a and b to be changed when they are being compared, as in:

  define precedes(a,b) = ((a = round(a)) < (b = round(b)));

(A more efficient way of achieving the same result would be to
use sort(round(x)).)

Examples of effects of various precedes functions for sorting
lists of integers:

  a > b             Sorts into nonincreasing order.

  abs(a) < abs(b)         Sorts into nondecreasing order of
                    absolute values, numbers with the
                    same absolute value retaining
                    their order.

  abs(a) <= abs(b)  Sorts into nondecreasing order of
                    absolute values, possibly
                    changing the order of numbers
                    with the same absolute value.

  abs(a) < abs(b) || abs(a) == abs(b) && a < b
                    Sorts into nondecreasing order of
                    absolute values, numbers with the
```

```
                     same absolute value being in
                     nondecreasing order.

      iseven(a)         Even numbers in possibly changed order
                        before odd numbers in unchanged order.

      iseven(a) && isoddd(b)  Even numbers in unchanged order before
                        odd numbers in unchanged order.

      iseven(a) ? iseven(b) ? a < b : 1 : 0
                        Even numbers in nondecreasing order
                        before odd numbers in unchanged order.

      a < b && a < 10         Numbers less than 10 in nondecreasing
                        order before numbers not less than 10
                        in unchanged order.

      !ismult(a,b)            Divisors d of any integer i for which
                        i is not also a divisor of d will
                        precede occurrences of i; the order of
                        integers which divide each other will
                        remain the same; the order of pairs of
                        integers neither of which divides the
                        other may be changed.  Thus occurrences
                        of 1 and -1 will precede all other
                        integers; 2 and -2 will precede all
                        even integers; the order of occurrences
                        of 2 and 3 may change; occurrences of 0
                        will follow all other integers.

      1                 The order of the elements is reversed
```

EXAMPLES

```
    ; A = list(1, 7, 2, 4, 2)
    ; print sort(A)

    list (5 elements, 5 nonzero):
        [[0]] = 1
        [[1]] = 2
        [[2]] = 2
        [[3]] = 4
        [[4]] = 7

    ; B = list("pear", 2, null(), -3, "orange", null(), "apple", 0)
    ; print sort(B)

    list (8 elements, 7 nonzero):
        [[0]] = NULL
        [[1]] = NULL
        [[2]] = -3
        [[3]] = 0
        [[4]] = 2
        [[5]] = "apple"
        [[6]] = "orange"
        [[7]] = "pear"

    ; define precedes(a,b) = (iseven(a) && isodd(b))
    ; print sort(A)
```

```
    list (5 elements, 5 nonzero):
        [[0]] = 2
        [[1]] = 4
        [[2]] = 2
        [[3]] = 1
        [[4]] = 7

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    join, reverse
```

sqrt - evaluate exactly or approximate a square root

```
SYNOPSIS
    sqrt(x [, eps[, z]])

TYPES
    If x is an object of type tt, or if x is not an object but y
    is an object of type tt, and the user-defined function
    tt_round has been defined, the types for x, y, z are as
    required for tt_round, the value returned, if any, is as
    specified in tt_round.  For object x or y, z defaults to a
    null value.

    For other argument types:

    x       real or complex
    eps           nonzero real
    z       integer

    return  real or complex

DESCRIPTION
    For real or complex x, sqrt(x, y, z) returns either the exact
    value of a square root of x (which is possible only if this
    square root is rational) or a number for which the real and
    imaginary parts are either exact or the nearest below or nearest
    above to the exact values.

    The argument, eps, specifies the epsilon/error value to be
    used during calculations.  By default, this value is epsilon().

    The seven lowest bits of z are used to control the signs of the
    result and the type of any rounding:

    z bit 6    ((z & 64) > 0)

      0:    principal square root

      1:    negative principal square root

    z bit 5    ((z & 32) > 0)

      0:    return aprox square root

      1:    return exact square root when real & imaginary are rational

    z bits 5-0    (z & 31)

      0:    round down or up according as y is positive or negative,
            sgn(r) = sgn(y)

      1:    round up or down according as y is positive or negative,
            sgn(r) = -sgn(y)

      2:    round towards zero, sgn(r) = sgn(x)

      3:    round away from zero, sgn(r) = -sgn(x)
```

```
    4:    round down

    5:    round up

    6:    round towards or from zero according as y is positive or
          negative, sgn(r) = sgn(x/y)

    7:    round from or towards zero according as y is positive or
          negative, sgn(r) = -sgn(x/y)

    8:    a/y is even

    9:    a/y is odd

    10:   a/y is even or odd according as x/y is positive or negative

    11:   a/y is odd or even according as x/y is positive or negative

    12:   a/y is even or odd according as y is positive or negative

    13:   a/y is odd or even according as y is positive or negative

    14:   a/y is even or odd according as x is positive or negative

    15:   a/y is odd or even according as x is positive or negative

The value of y and lowest 5 bits of z are used in the same way as
y and z in appr(x, y, z): for either the real or imaginary part
of the square root, if this is a multiple of y, it is returned
exactly; otherwise the value returned for the part is the
multiple of y nearest below or nearest above the true value.
For z = 0, the remainder has the sign of y;  changing bit 0
changes to the other possibility; for z = 2, the remainder has the
sign of the true value, i.e. the rounding is towards zero; for
z = 4, the remainder is always positive, i.e. the rounding is down;
for z = 8, the rounding is to the nearest even multiple of y;
if 16 <= z < 32, the rounding is to the nearest multiple of y when
this is uniquely determined and otherwise is as if z were replaced
by z - 16.

With the initial default values, 1e-20 for epsilon() and 24 for
config("sqrt"), sqrt(x) returns the principal square root with
real and imaginary parts rounded to 20 decimal places, the 20th
decimal digit being even when the part differs from a multiple
of 1e-20 by 1/2 * 1e-20.
```

EXAMPLE

```
; eps = 1e-4
; print sqrt(4,eps,0), sqrt(4,eps,64), sqrt(8i,eps,0), sqrt(8i, eps, 64)
2 -2 2+2i -2-2i

; print sqrt(2,eps,0), sqrt(2,eps,1), sqrt(2,eps,24)
1.4142 1.4143 1.4142

; x = 1.2345678^2
; print sqrt(x,eps,24), sqrt(x,eps,32), sqrt(x,eps,96)
1.2346 1.2345678 -1.2345678
```

```
    ; print sqrt(.00005^2, eps, 24), sqrt(.00015^2, eps, 24)
    0 .0002

LIMITS
    none

LINK LIBRARY
    COMPLEX *c_sqrt(COMPLEX *x, NUMBER *ep, long z)
    NUMBER *qisqrt(NUMBER *q)
    NUMBER *qsqrt(NUMBER *x, NUMBER *ep, long z)
    FLAG zsqrt(ZVALUE x, ZVALUE *result, long z)

SEE ALSO
   appr, epsilon
```

srand - seed the subtractive 100 shuffle pseudo-random number generator

```
SYNOPSIS
    srand([seed])

TYPES
    seed    integer, matrix of integers or rand state

    return  rand state

DESCRIPTION
    Seed the pseudo-random number using an subtractive 100 shuffle generator.

    For integer seed != 0:

      Any buffered rand generator bits are flushed.  The subtractive table
      for the rand generator is loaded with the default subtractive table.
      The low order 64 bits of seed is xor-ed against each table value.
      The subtractive table is shuffled according to seed/2^64.

      The following calc code produces the same effect on the internal
      subtractive table:

          /* reload default subtractive table xor-ed with low 64 seed bits */
          seed_xor = seed & ((1<<64)-1);
          for (i=0; i < 100; ++i) {
            subtractive[i] = xor(default_subtractive[i], seed_xor);
          }

          /* shuffle the subtractive table */
          seed >>= 64;
          for (i=100; seed > 0 && i > 0; --i) {
            quomod(seed, i+1, seed, j);
            swap(subtractive[i], subtractive[j]);
          }

      Seed must be >= 0.  All seed values < 0 are reserved for future use.

      The subtractive table pointers are reset to subtractive[36]
      and subtractive[99].  Last the shuffle table is loaded with
      successive values from the subtractive 100 generator.

      There is no limit on the size of a seed.  On the other hand,
      extremely large seeds require large tables and long seed times.
      Using a seed in the range of [2^64, 2^64 * 100!) should be
      sufficient for most purposes.  An easy way to stay within this
      range to to use seeds that are between 21 and 178 digits, or
      64 to 588 bits long.

      To help make the generator produced by seed S, significantly
      different from S+1, seeds are scrambled prior to use.  The
      internal function randreseed64 maps [0,2^64) into [0,2^64) in a
      1-to-1 and onto fashion for every 64 bits of S.

      The purpose of the randreseed64() is not to add security.  It
      simply helps remove the human perception of the relationship
```

```
  between the seed and the production of the generator.

  The randreseed64 process does not reduce the security of the
  rand generator.    Every seed is converted into a different
  unique seed.  No seed is ignored or favored.  See the rand
  help file for details.

For integer seed == 0:

  Restore the initial state and modulus of the rand generator.
  After this call, the rand generator is restored to its initial
  state after calc started.

  The subtractive 100 pointers are reset to subtractive[36]
  and subtractive[99].  Last the shuffle table is loaded with
  successive values from the subtractive 100 generator.

  The call:

      srand(0)

  restores the rand generator to the initial conditions at calc startup.

For matrix arg:

  Any buffered random bits are flushed.  The subtractive table with the
  first 100 entries of the matrix mod 2^64.

  The subtractive 100 pointers are reset to subtractive[36]
  and subtractive[99].  Last the shuffle table is loaded with
  successive values from the subtractive 100 generator.

  This form allows one to load the internal subtractive 100 generator
  with user supplied values.

  The randreseed64 process is NOT applied to the matrix values.

For rand state arg:

  Restore the rand state and return the previous state.  Note that
  the argument state is a rand state value (isrand(state) is true).
  Any internally buffered random bits are restored.

  All calls to srand(seed) return the previous state or current
  state in case of srand().  Their return value can be supplied
  to srand in restore the generator to that previous state:

       state = srand(123456789);
       newstate = srand();      /* save state */

       x = rand();
       ...
       srand(newstate);   /* restore state to after srand(123456789) */
       x1 = rand(); /* produces the same value as x */
       ...
       srand(state);      /* restore original state */

For no arg given:
```

```
        Return current s100 generator state.  This call does not alter
        the generator state.

        This call allows one to take a snapshot of the current generator state.

    See the rand help file for details on the generator.

EXAMPLE
    ; srand(0x8d2dcb2bed3212844f4ad31)
            RAND state
    ; state = srand();
    ; print rand(123), rand(123), rand(123), rand(123), rand(123), rand(123);
    80 95 41 78 100 27
    ; print rand(123), rand(123), rand(123), rand(123), rand(123), rand(123);
    122 109 12 95 80 32
    ; state2 = srand(state);
    ; print rand(123), rand(123), rand(123), rand(123), rand(123), rand(123);
    80 95 41 78 100 27
    ; print rand(123), rand(123), rand(123), rand(123), rand(123), rand(123);
    122 109 12 95 80 32
    ; state3 = srand();
    ; print state3 == state2;
    1
    ; print rand();
    10710588361472584495

LIMITS
    for matrix arg, the matrix must have at least 100 integers

LINK LIBRARY
    RAND *zsrand(ZVALUE *pseed, MATRIX *pmat100)
    RAND *zsetrand(RAND *state)

SEE ALSO
    seed, srandom, randbit, isrand, random, srandom, israndom
```

srandom - seed the Blum-Blum-Shub pseudo-random number generator

```
SYNOPSIS
    srandom([state])
    srandom(seed)
    srandom(seed, newn)
    srandom(seed, ip, iq, trials)

TYPES
    state   random state
    seed    integer
    newn    integer
    ip            integer
    iq            integer
    trails  integer

    return  random state

DESCRIPTION
    Seed the pseudo-random number using the Blum-Blum-Shub generator.

    There are two primary values contained inside generator state:

      Blum modulus:

            A product of two primes.  Each prime is 3 mod 4.

      Quadratic residue:

            Some integer squared modulo the Blum modulus.

    Seeding the generator involves changing the Quadratic residue
    and in most cases the Blum modulus as well.

    In addition to the two primary values values, an internal buffer of
    unused random output is kept.  When the generator is seeded, any
    buffered random output is tossed.

    In each of the following cases, srandom returns the previous state
    of the generator.  Depending on what args are supplied, a new
    generator state is established.  The exception is the no-arg state.

    0 args:                   srandom()

      Returns the current generator state.  Unlike all of the other
      srandom calls, this call does not modify the generator, nor
      does it flush the internal bits.

    1 arg (state arg):              srandom(state)

      sets the generator to 'state', where 'state' is a previous
      return of srandom().

    1 arg (0 seed):                 srandom(0)

      Sets the generator to the initial startup state.  This a
```

```
  call of srandom(0) will restore the generator to the state
  found when calc starts.

1 arg (seed >= 2^32):           srandom(21609139158123209^9+17)

  The seed value is used to compute the new quadratic residue.
  The seed passed will be successively squared mod the Blum
  modulus until we get a smaller value (modulus wrap).  The
  calc resource file produces an equivalent effect:

        /* assume n is the current Blum modulus */
        r = seed;
        do {
              last_r = r;
              r = pmod(r, 2, n);
        } while (r > last_r);
        /* r is the new Quadratic residue */

  In this form of srandom, the Blum modulus is not changed.

  NOTE: [1,2^32) seed values and seed<0 values
        are reserved for future use.

2 args (seed, newn>=2^32):            srandom(seed, newn)

  The newn value is used as the new Blum modulus.  This modulus
  is assumed to be a product of two primes that are both 3 mod
  4.  The newn value is not factored, it is only checked to see
  if it is 1 mod 4.

  In this call form, newn value must be >= 2^32.

  The seed arg is used to establish the initial quadratic value
  once newn has been made the Blum moduli.  The seed must
  be either 0 or >= 2^32.  If seed == 0, the initial quadratic
  residue used with srandom(0) is used with the new Blum moduli.
  If seed >= 2^32, then srandom(seed, newn) has the same effect as:

        srandom(0, newn);    /* set Blum modulus & def quad res */
        srandom(seed);         /* set quadratic residue */

  Use of newn values that are not the product of two 3 mod 4
  primes will result in a non-cryptographically strong generator.
  While the generator will produce values, their quality will
  be suspect.

  The period of the generator determines how many bits will
  be produced before it repeats.      The period is determined
  by the Blum modulus.  Some newn values (that are a product
  of two 3 mod 4 primes) can produce a generator with a
  very short period making is useless for most applications.

  When Blum modulus is p*q, the period of a generator is:

      lcm(factors of p-1 and q-1)

  One can construct a generator with a maximal period when 'p'
  and 'q' have the fewest possible factors in common.  The
  quickest way to select such primes is only use 'p' and 'q' when
```

'(p-1)/2' and '(q-1)/2' are both primes. Assuming that fp=(p-1)/2, fq=(q-1)/2, p and q are all primes 3 mod 4, the period of the generator is the longest possible:

```
    lcm(factors of p-1 and q-1) == lcm(2,fp,2,fq) = 2*fp*fq = ~n/2
```

The following calc resource file:

```
    /* find first Blum prime: p */
    fp = int((ip-1)/2);
    do {
      do {
          fp = nextcand(fp+2, 1, 0, 3, 4);
          p = 2*fp+1;
      } while (ptest(p, 1, 0) == 0);
    } while (ptest(p, trials) == 0 || ptest(fp, trials));

    /* find second Blum prime: q */
    fq = int((iq-1)/2);
    do {
      do {
          fq = nextcand(fq+2, 1, 0, 3, 4);
          q = 2*fq+1;
      } while (ptest(q, 1, 0) == 0);
    } while (ptest(q, trials) == 0 || ptest(fq, trials));

    /* seed the generator */
    srandom(ir, p*q);
```

Where:

```
    ip
      initial search location for the Blum prime 'p'
    iq
      initial search location for the Blum prime 'q'
    ir
      initial Blum quadratic residue generator.  The 'ir'
      must be 0 or >= 2^32, preferably large some random
      value < p*q.  The following may be useful to set ir:

            srand(p+q);
            ir = randbit(highbit(p)+highbit(q))
    trials
      number of pseudo prime tests that a candidate must pass
      before being considered a probable prime (must be >0, try 25)
```

The calc standard resource file seedrandom.cal will produce a seed a generator. If the config value custom("resource_debug") is 0 or 1, then the selected Blum modulus and quadratic residue will be printed. If the global value is 1, then p and q are also printed. The resource file defines the function:

```
      seedrandom(seed1, seed2, size [, trials])
```

Where:

```
    seed1
      A random number >= 10^20 and perhaps < 10^93.
    seed2
      A random number >= 10^20 and perhaps < 10^93.
    size
```

```
      Minimal Blum modulus size in bits,  This must be >= 32.
      A value of 512 might be a good choice.
    trials
      number of pseudo prime tests that a candidate must pass
      before being considered a probable prime (must be >0, try 25).
      Using the default value of 25 might be a good choice.

Unfortunately finding optimal values can be very slow for large
values of 'p' and 'q'.  On a 200Mhz r4k, it can take as long as
1 minute at 512 bits, and 5 minutes at 1024 bits.

For the sake of speed, you may want to use to use one of the
pre-compiled in Blum moduli via the [1
If you don't want to use a pre-compiled in Blum moduli you can
compute your own values ahead of time.    This can be done by a
method of your own choosing, or by using the seedrandom.cal
resource file in the following way:

    1) calc                # run calc
    2) read seedrandom           # load seedrandom
    3) config("resource_debug",0)  # we want the modulus & quad res only
    4) seedrandom( ~pound out 20-93 random digits on the keyboard~,
               ~pound out 20-93 random digits on the keyboard~,
               512 )
    5) save the seed and newn values for later use

NOTE: [1,2^32) seed values, seed<0 values, [21,2^32) newn values
      and newn<=0 values are reserved for future use.

2 args (seed, 1>=newn>=20):           srandom(seed, newn)

The newn is used to select one of 20 pre-computed Blum moduli.

The seed arg is used to establish the initial quadratic value
once newn has been made the Blum moduli.  The seed must be
either 0 or >= 2^32.  If seed == 0, the pre-compiled quadratic
residue for the given newn is selected.    If seed >= 2^32, then
srandom(seed, newn) has the same effect as:

      srandom(0, newn);    /* set Blum modulus & def quad res */
      srandom(seed);         /* set quadratic residue */

Note that unlike the newn>=2^32 case, a seed if 0 uses the
pre-compiled quadratic residue for the selected pre-compiled
Blum moduli.

The pre-defined Blum moduli and quadratic residues were selected
by LavaRnd, a hardware random number generator.  See the URL:

    http://www.LavaRnd.org/

for an explanation of how the LavaRnd random number generator works.
For more information, see the comments at the top of the calc
source file, zrandom.c.

The purpose of these pre-defined Blum moduli is to provide users with
an easy way to use a generator where the individual Blum primes used
are not well known.  True, these values are in some way "MAGIC", on
the other hand that is their purpose!  If this bothers you, don't
```

```
  use them.

  The value 'newn' determines which pre-defined generator is used.

      newn ==  1: (Blum modulus bit length 130)
      newn ==  2: (Blum modulus bit length 137)
      newn ==  3: (Blum modulus bit length 147)
      newn ==  4: (Blum modulus bit length 157)
      newn ==  5: (Blum modulus bit length 257)
      newn ==  6: (Blum modulus bit length 259)
      newn ==  7: (Blum modulus bit length 286)
      newn ==  8: (Blum modulus bit length 294)
      newn ==  9: (Blum modulus bit length 533)
      newn == 10: (Blum modulus bit length 537)
      newn == 11: (Blum modulus bit length 542)
      newn == 12: (Blum modulus bit length 549)
      newn == 13: (Blum modulus bit length 1048)
      newn == 14: (Blum modulus bit length 1054)
      newn == 15: (Blum modulus bit length 1055)
      newn == 16: (Blum modulus bit length 1062)
      newn == 17: (Blum modulus bit length 2062)
      newn == 18: (Blum modulus bit length 2074)
      newn == 19: (Blum modulus bit length 2133)
      newn == 20: (Blum modulus bit length 2166)

  See the comments near the top of the source file, zrandom.c, for the
  actual pre-compiled values.

  The Blum moduli associated with 1 <= newn < 9 are subject
  to having their Blum moduli factored, depending in their size,
  by small PCs in a reasonable to large supercomputers/highly
  parallel processors over a long time.  Their value lies in their
  speed relative the the default Blum generator.  As of Feb 1997,
  the Blum moduli associated with 13 <= newn < 20 appear to
  be well beyond the scope of hardware and algorithms,
  and 9 <= newn < 12 might be factorable with extreme difficulty.

  The following table may be useful as a guide for how easy it
  is to factor the modulus:

       1 <= newn <= 4   PC using ECM in a short amount of time
       5 <= newn <= 8   Workstation using MPQS in a short amount of time
       8 <= newn <= 12  High end supercomputer or high parallel processor
                    using state of the art factoring over a long time
      12 <= newn <= 16  Beyond Feb 1997 systems and factoring methods
      17 <= newn <= 20  Well beyond Feb 1997 systems and factoring methods

  In other words, use of newn == 9, 10, 11 and 12 is likely to
  work just fine for all but the truly paranoid.

  NOTE: [1,2^32) seed values, seed<0 values, [21,2^32) newn values
        and newn<=0 values are reserved for future use.

4 args (seed, ip>=2^16, iq>=2^16, trials):  srandom(seed, ip, iq, 25)

  The 'ip' and 'iq' args are used to find simples prime 3 mod 4

  The call srandom(seed, ip, iq, trials) has the same effect as:
```

```
        srandom(seed,
              nextcand(ip, trials,0, 3,4)*nextcand(iq, trials,0, 3,4));

    Note that while the newn is very likely to be a product of
    two primes both 3 mod 4, there is no guarantee that the period
    of the generator will be long.      The likelihood is that the
    period will be long, however.  See one of the 2 arg srandom
    calls above for more information on this issue.

    NOTE: [1,2^32) seed values, seed<0 values, [21,2^32) newn values,
          newn<=0 values, ip<2^16 and iq<2^16 are reserved for future use.

  See the random help file for details on the generator.
```

EXAMPLE

```
    ; srandom(0x8d2dcb2bed3212844f4ad31)
            RANDOM state
    ; state = srandom();
    ; print random(123), random(123), random(123), random(123), random(123)
    42 58 57 82 15
    ; print random(123), random(123), random(123), random(123), random(123)
    90 121 109 114 80
    ; state2 = srandom(state);
    ; print random(123), random(123), random(123), random(123), random(123)
    42 58 57 82 15
    ; print random(123), random(123), random(123), random(123), random(123)
    90 121 109 114 80
    ; state3 = srandom();
    ; print state3 == state2;
    1
    ; print random();
    2101582493746841221
```

LIMITS

```
    integer seed == 0 or >= 2^32
    for newn >= 2^32: newn % 4 == 1
    for small newn: 1 <= newn <= 20
    ip >= 2^16
    iq >= 2^16
```

LINK LIBRARY

```
    RAND *zsrandom(ZVALUE *pseed, MATRIX *pmat55)
    RAND *zsetrandom(RAND *state)
```

SEE ALSO

    seed, srand, randbit, isrand, random, srandom, israndom

ssq - sum of squares

```
SYNOPSIS
    ssq(x1, x2, ...)

TYPES
    x1, x2, ... lists or values for which required operations are defined

    return  as determined by the operations on x1, x2, ...

DESCRIPTION
    Null values are ignored; ssq() returns the null value.
    If no argument is a list, returns x1^2 + x2^2 + ...
    If an argument = list(t1, t2, ...) it contributes ssq(t1, t2, ...)
      to the result.
EXAMPLE
    ; print ssq(1,2,3), ssq(1+2i, 3-4i, 5 +6i)
    14 -21+40i

    ; mat A[2,2] = {1,2,3,4}; mat B[2,2] = {5,6,7,8}
    ; print ssq(A, B, A + B)

    mat [2,2] (4 elements, 4 nonzero):
      [0,0] = 190
      [0,1] = 232
      [1,0] = 286
      [1,1] = 352

    ; ssq(list(2,3,5),7)
      87

    ; ssq(1,2,3,4,5,6,7,8)
      204
    ; ssq(1,2, list(3,4,list(5,6)), list(), 7, 8)
      204

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    sum, max, min
```

stoponerror - controls when / if calc stops calculations based on errors

```
SYNOPSIS
    stoponerror([n])

TYPES
    n       integer

    return  null value or error value

DESCRIPTION

    The stoponerror controls when or if calc stops based on the
    number of errors:

      n == -1           do not stop
      n == 0            stop on error unless calc was invoked with -c
      n > 0       stop when n errors are encountered

    When no arguments are given, stoponerror() returns the current
    stoponerror value.  When 1 argument is given, stoponerror() returns
    the previous stoponerror value.

EXAMPLE
    ; stoponerror()
      0

LIMITS
    -1 <= stoponerror < 2147483647

LINK LIBRARY
    none

SEE ALSO
    errcount, errmax, errorcodes, iserror, errno, strerror, newerror
```

str - convert some types of values to strings

```
SYNOPSIS
    str(x)

TYPES
    x       null, string, real or complex number

    return  string

DESCRIPTION
    Convert a value into a string.

    If x is null, str(x) returns the string "".

    If x is a string, str(x) returns x.

    For real or complex x, str(x) returns the string representing x
    in the current printing mode; configuration parameters affecting
    this are "mode", "mode2", "display", "outround", "tilde", "leadzero",

EXAMPLE
    ; str("")
          ""
    ; str(null())
          ""
    ; print str(123), str("+"), str(4i), str("is the same as"), str(123+4i)
    123 + 4i is the same as 3+4i

    ; base2(16),
    ; print str(23209)
    23209 /* 0x5aa9 */

LIMITS
    none

LINK LIBRARY
    void math_divertio();
    qprintnum(NUMBER *x, int outmode);
    char *math_getdivertedio();

    math_divertio();
    comprint(COMPLEX *x);
    char *math_getdivertedio();

SEE ALSO
    base, base2, config
```

strcat - concatenate null-terminated strings

```
SYNOPSIS
    strcat(x1, x2, ...)

TYPES
    x1, x2, ...         strings

    return        string

DESCRIPTION
    strcat(x1, x2, ...) forms a string starting with a copy of
    x1 before the first, if any, null character in x1, followed by the
    initial non-null characters of any later arguments x2, ... The
    length of the resulting string will be the sum of the lengths
    of the component strings considered as null-terminated strings (i.e.
    the lengths as returned by strlen()). The sum function may be used
    to concatenate strings where '\0' is to be considered as an ordinary
    character, either by sum(x1, x2, ...) or sum(list(x1, x2, ...));
    in this case, the size of the resulting string is the sum of the
    sizes of the component strings.


EXAMPLE
    ; A = "abc"; B = "XY"; C = "  ";
    ; print strcat(A, B, C, B, A)
    abcXY  XYabc

LIMITS
    The number of arguments may not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    strcmp, strcpy, strerror, strlen, strncmp, strncpy, strpos,
    strprintf, strscan, strscanf, substr
```

strcmp - compare two strings in the customary ordering of strings

```
SYNOPSIS
    strcmp(s1, s2)

TYPES
    s1              string
    s2              string

    return  integer (1, 0 or -1)

DESCRIPTION
    Let n1 = size(s1), n2 = size(s2) and m = min(n1, n2).
    This function compares up to m values of consecutive characters
    in the strings s1 and s2. If an inequality is encountered, the
    function returns 1 or -1 according as the greater character is
    in s1 or s2. If there has been no inequality, the function
    returns 1, 0, or -1 according as n1 is greater than, equal to,
    or less than n2.
    Note that null characters within the strings are included in the
    comparison.

EXAMPLE
    strcmp("ab", "abc") == -1
    strcmp("abc", "abb") == 1
    strcmp("abc", "abc") == 0
    strcmp("abc", "abd") == -1
    strcmp("abc\0", "abc") == 1
    strcmp("a\0b", "a\0c") == -1

LIMITS
    none

LINK LIBRARY
    FLAG stringrel(STRING *s1, STRING *s2)

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strncpy, strpos,
    strprintf, strscan, strscanf, substr
```

strcpy - copy head or all of a string to head or all of a string

```
SYNOPSIS
    strcpy(s1, s2)

TYPES
    s1              string
    s2              string

    return  string

DESCRIPTION
    Let n1 = size(s1), n2 = size(n2), and m = min(n1, n2).
    This function replaces the first m characters of s1 by the first
    m characters of s2, and if m < n1, replaces the next character of
    s1 by '\0'. The size of s1 and any later characters of s1 are unchanged.
    s1, with its new value, is returned.
    Unlike the C Library function with the same name, this function does
    not require n1 to be greater than or equal to n2, but if this is so,
    normal printing of the returned value will give the same result as
    normal printing of s2.

EXAMPLE
    strcpy("", "xyz") == ""
    strcpy("a", "xyz") == "x"
    strcpy("ab", "xyz") == "xy"
    strcpy("abc", "xyz") == "xyz"
    strcpy("abcd", "xyz") == "xyz\0"
    strcpy("abcde", "xyz") == "xyz\0e"
    strcpy("abcdef", "xyz") == "xyz\0ef"
    strcpy("abc", "") == "\0bc"

LIMITS
    none

LINK LIBRARY
    STRING* stringcpy(STRING *s1, STRING *s2)

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strncpy, strpos,
    strprintf, strscan, strscanf, substr
```

strerror - returns a string describing an error value

```
SYNOPSIS
    strerror([x])

TYPES
    x       error-value or integer in [0, 32767], defaults to errno()

    return  string

DESCRIPTION
    If x is the error-value with index n, strerror(x) and strerror(n)
    return one of:

            a system-generated message,
            a calc-generated description,
            a user-defined description created by newerror(str),
            the string "Error n",

    where, in the last form, n is represented decimally.

EXAMPLE
    System error messages may be different for different systems.
    ; errmax(errcount()+3)
          0
    ; strerror(2)
      "No such file or directory"

    ; x = 3 * ("a" + "b")
    ; print strerror(x)
    Bad arguments for +

    ; a = newerror("alpha")
    ; print strerror(a)
    alpha

    ; print strerror(999)
    Error 999

    ; a = 1/0
    ; print strerror()
    Division by zero

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    strcat, strcpy, strlen, strncmp, strncpy, strpos,
    strprintf, strscan, strscanf, substr,

    errcount, errmax, error, iserror, errno, newerror, errorcodes,
    stoponerror
```

strlen - number of characters in a string

```
SYNOPSIS
    strlen(x)

TYPES
    x       string

    return  integer

DESCRIPTION
    strlen(x) returns the number of characters in x

EXAMPLE
    ; print strlen(""), strlen("abc"), strlen("a b\tc\\d")
    0 3 7

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    strcat, strcpy, strerror, strncmp, strncpy, strpos,
    strprintf, strscan, strscanf, substr
```

strncmp - compare two strings up to a specified number of characters

```
SYNOPSIS
    strncmp(s1, s2, n)

TYPES
    s1            string
    s2            string
    n       nonnegative integer

    return  integer (1, 0 or -1)

DESCRIPTION
    Let n1 = size(s1), n2 = size(s2) and m = min(n1, n2, n).
    This function compares up to m values of consecutive characters
    in the strings s1 and s2. If an inequality is encountered, the
    function returns 1 or -1 according as the greater character is
    in s1 or s2. If there has been no inequality, the function
    returns 1, 0, or -1 according as min(n1, n) is greater than, equal
    to, or less than min(n2, n); in particular, if n1 and n2 are
    both greater than equal to n, 0 is returned.


EXAMPLE
strncmp("abc", "xyz", 0) == 0
strncmp("abc", "xyz", 1) == -1
strncmp("abc", "", 1) == 1
strncmp("a", "b", 2) == -1
strncmp("ab", "ac", 2) == -1
strncmp("\0ac", "\0b", 2) == -1
strncmp("ab", "abc", 2) == 0
strncmp("abc", "abd", 2) == 0


LIMITS
    none

LINK LIBRARY
    This function uses FLAG stringrel(STRING *s1, STRING *s2),
    temporarily replacing the string sizes by min(n1,n) and min(n2,n).

SEE ALSO
    strcat, strcpy, strerror, strlen, strncpy, strpos,
    strprintf, strscan, strscanf, substr
```

strncpy - copy a number of chracters from head or all of a stringr

```
            to head or all of a string

SYNOPSIS
    strncpy(s1, s2, n)

TYPES
    s1              string
    s2              string
    n       nonnegative integer

    return  string

DESCRIPTION
    Let n1 = size(s1), n2 = size(n2), and m = min(n1, n2, n).
    This function replaces the first m characters of s1 by the first
    m characters of s2, and if min(n1, n) > n2, replaces the next
    min(n1,n) - n2 characters of s1 by '\0'. The size of s1 and any
    later characters of s1 are unchanged. The function returns s1, with
    new value.

EXAMPLE
    strncpy("abcdef", "xyz", 0) == "abcdef"
    strncpy("abcdef", "xyz", 1) == "xbcdef"
    strncpy("abcdef", "xyz", 2) == "xycdef"
    strncpy("abcdef", "xyz", 3) == "xyzdef"
    strncpy("abcdef", "xyz", 4) == "xyz\0ef"
    strncpy("abcdef", "xyz", 5) == "xyz\0\0f"
    strncpy("ab", "xyz", 3) == "xy"

LIMITS
    none

LINK LIBRARY
    STRING* stringncpy(STRING *s1, STRING *s2, long num)

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strpos,
    strprintf, strscan, strscanf, substr
```

strpos - print the first occurrence of a string in another string

```
SYNOPSIS
    strpos(s, t)

TYPES
    s       str
    t       str

    return  int

DESCRIPTION
    This function returns the location of the first occurance of the string t
    in the string s.  If t is not found within s, 0 is returned.  If t is
    found at the beginning of s, 1 is returned.

EXAMPLE
    ; strpos("abcdefg", "c")
          3
    ; strpos("abcdefg", "def")
          4
    ; strpos("abcdefg", "defg")
          4
    ; strpos("abcdefg", "defgh")
          0
    ; strpos("abcdefg", "abc")
          1
    ; strpos("abcdefg", "xyz")
          0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strncpy,
    strprintf, strscan, strscanf, substr
```

strprintf - formatted print to a string

```
SYNOPSIS
    strprintf(fmt, x_1, x_2, ...)

TYPES
    fmt                 string
    x_1, x_2, ... any

    return        string

DESCRIPTION
    This function returns the string formed from the characters that
    would be printed to standard output by printf(fmt, x_1, x_2, ...).

EXAMPLE
    ; strprintf("h=%d, i=%d", 2, 3);
          "h=2, i=3"

    ; c = config("epsilon", 1e-6); c = config("display", 6);
    ; c = config("tilde", 1); c = config("outround", 0);
    ; c = config("fullzero", 0);
    ; fmt = "%f,%10f,%-10f,%10.4f,%.4f,%.f.\n";
    ; a = sqrt(3);
    ; strprintf(fmt,a,a,a,a,a,a);
          "1.732051,    1.732051,1.732051  ,   ~1.7320,~1.7320,~1.
    "

LIMITS
    The number of arguments of strprintf() is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strncpy, strpos,
    strscan, strscanf, substr,

    printf, fprintf, print
```

strscan - scan a string for possible assignment to variables

```
SYNOPSIS
   strscan(str, x_1, x_2, ..., x_n)

TYPES
    str                 string
    x_1, x_2, ... any

    return        integer

DESCRIPTION
   Successive fields of str separated by white space are read and
   evaluated so long as values remain in the x_i arguments;  when the
   x_i corresponding to the field is an lvalue the value obtained for the
   i-th field is assigned to x_i.

   The function returns the number of fields evaluated.

EXAMPLE
    global a,b
    ; strscan(" 2+3  a^2  print(b)", a, b, 0);
    25
      3
    ; print a,b
    5 25

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    none

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strncpy, strpos,
    strprintf, strscanf, substr
```

strscanf - formatted scan of a string

```
SYNOPSIS
    strscanf(str, fmt, x_1, x_2, ...)

TYPES
    str                 string
    fmt                 string
    x_1, x_2, ... lvalues

    return  null, nonnegative integer, or error value

DESCRIPTION
    If the str is "", the null value is returned.

    Otherwise, until the terminating null character of either fmt or str
    is reached, characters other than '%' and whitespace are read from
    fmt and compared with the corresponding characters read from str.
    If the characters match, reading continues.  If they do not match
    an integer value is returned.  If whitespace is encountered in fmt,
    starting at the current positions in fmt and str, any whitespace
    characters are skipped and reading and comparison begins as before
    if neither fmt nor str has reached its end.

    When a '%' is encountered in fmt, if this is immediately followed by
    another '%', the pair formed is considered as if one '%' were read and
    reading from fmt and fs continues if and only if fs has a matching
    '%'.  A single '%' read from fmt is taken to indicate the beginning of
    a conversion specification field consisting in succession of:

            an optional '*',
            optional decimal digits,
            one of 'c', 's', 'n', 'f', 'e', 'i' or a scanset specifier.

    A scanset specifier starts with '[' and an optional '^', then an optional
    ']', then optional other characters, and ends with ']'.   If any
    other sequence of characters follows the '%', characters before the
    first exceptional character (which could be the terminating null
    character of the fmt string) are ignored, e.g. the sequence " %*3d " does
    the same as " d ".  If there is no '*' at the beginning of the specifier,
    and the list x_1, x_2, ... has not been exhausted,
    a value will be assigned to the next lvalue in the list; if no lvalue
    remains, the reading of fs stops and the function returns the number
    of assignments that have been made.

    Occurrence of '*' indicates that characters as specified are to be read
    but no assignment will be made.

    The digits, if any, read in the specifier are taken to be decimal digits
    of an integer which becomes the maximum "width" (number of characters
    to be read from str for string-type assignments); absence of digits or
    all zero digits in the 'c' case are taken to mean width = 1.  Zero width
    for the other cases are treated as if infinite.   Fewer characters than
    the specifier width may be read if end-of-file is reached or in the case
    of scanset specification, an exceptional character is encountered.

    If the ending character is 'c', characters are read from fs to
```

form a string, which will be ignored or in the non-'*' case, assigned to the next lvalue.

In the 's' case, reading to form the string starts at the first non-white character (if any) and ceases when end-of-file or further white space is encountered or the specified width has been attained.

The cases 'f', 'e', 'r', 'i' may be considered to indicate expectation of floating-point, exponential, ratio, or integer representation of the number to be read. For example, 'i' might be taken to suggest a number like +2345; 'r' might suggest a representation like -27/49; 'e' might suggest a representation like 1.24e-7; 'f' might suggest a representation like 27.145. However, there is no test that the the result conforms to the specifier. Whatever the specifier in these cases, the result depends on the characters read until a space or other exceptional character is read. The characters read may include one or more occurrences of +, -, * as well as /, interpreted in the usual way, with left-to-right associativity for + and -, and for * and /. Also acceptable is a trailing i to indicate an imaginary number. For example the expression

```
2+3/4*7i+3.15e7
```

would be interpreted as for an ordinary evaluation. A decimal fraction may have more than one dot: dots after the first, which is taken to be the decimal point, are ignored. Thus "12.3..45e6.7" is interpreted as if it were "12.345e67".

For the number specifiers 'f', 'e', 'r', 'i', any specified width is ignored.

For the specifier 'n', the index of the next character to b e read is assigned to the corresponding lvalue. (Any width or skip specification is ignored.)

EXAMPLE

```
; global a, b, c, d
; A = "abc xyz 234.6 alpha"
; strscanf(A, "%s%*[^0123456789]%f%n", a, b, c)
  3
; print a, b, c
; abc 234.6 13

; strscanf(A, "%*13c%s", d);
  1
; print d
; alpha
```

LIMITS

The number of arguments is not to exceed 1024.

LINK LIBRARY

```
int fscanfid(FILEID id, char *fmt, int count, VALUE **vals);
```

SEE ALSO

strcat, strcpy, strerror, strlen, strncmp, strncpy, strpos, strprintf, strscan, substr

substr - extract a substring of given string

```
SYNOPSIS
    substr(str, pos, len)

TYPES
    str           string
    pos           nonnegative integer
    len           nonnegative integer

    return  string

DESCRIPTION
    If pos > length of str or len is zero, the null string "" is returned.

    If 1 <= pos <= strlen(str), substr(str, pos, len) returns the
    string of length min(strlen(str) - pos + 1, len) formed by
    consecutive characters of str starting at position pos, i.e. the
    string has length len if this is possible, otherwise it ends with
    the last character of str.      (The first character has pos = 1, the
    second pos = 2, etc.)

    If pos = 0, the result is the same as for pos = 1.

EXAMPLE
    ; A = "abcde";
    ; print substr(A,0,2), substr(A,1,2), substr(A,4,1), substr(A,3,5)
    ab ab d cde

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    strcat, strcpy, strerror, strlen, strncmp, strncpy, strpos,
    strprintf, strscan, strscanf
```

sum - sum, or sum of defined sums

```
SYNOPSIS
    sum(x_1, x_2, ...)

TYPES
    x_1, x_2, ... any

    return        any

DESCRIPTION
    If an argument x_i is a list with elements e_1, e_2, ..., e_n, it
    is treated as if x_i were replaced by e_1, e_2, ..., e_n; this may
    continue recurively if any of the e_j is a list.

    If an argument x_i is an object of type xx, then x_i is replaced by
    xx_sum(x_i) if the function xx_sum() has been defined.  If the
    type xx has been defined by:

            obj xx = {x, y, z},

    an appropriate definition of xx_sum(a) is sometimes a.x + a.y + a.z.
    sum(a) then returns the sum of the elements of a.

    If x_i has the null value, it is ignored.  Thus, sum(a, , b, , c)
    will return the same as sum(a, b, c).

    Assuming the above replacements, and that the x_1, x_2, ..., are
    of types for which addition is defined, sum(x_1, x_2, ...) returns
    the sum of the arguments.

EXAMPLE
    ; print sum(2), sum(5, 3, 7, 2, 9), sum(3.2, -0.5, 8.7, -1.2, 2.5)
    2 26 12.7

    ; print sum(list(3,5), 7, list(6, list(7,8), 2))
    38


    ; obj point {x, y}
    ; define point_add(a,b) = obj point = {a.x + b.x, a.y + b.y}
    ; obj point A = {1, 5}
    ; obj point B = {1, 4}
    ; obj point C = {3, 3}
    ; print sum(A, B, C)
    obj point {5, 12}

    ; define point_sum(a) = a.x
    ; print sum(A, B, C)
    5

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY

SEE ALSO
```

```
max, min, obj, ssq
```

swap - swap values of two variables

```
SYNOPSIS
    swap(x,y)

TYPES
    x, y    lvalues, any type

    return  null value

DESCRIPTION
   swap(x,y) assigns the value of x to a temporary location, temp say,
   assigns the value of x to y, and then assigns the value at temp to y.

   swap(x,y) should not be used if the current value of one of the
   variables is a component of the value of the other; for example, after:

       A = list(1,2,3); swap(A, A[[1]]);

   A will have the value 2, but a three-member list remains in memory
   with no method within calc of recalling the list or freeing the
   memory used.

EXAMPLE
   ; x = 3/4; y = "abc"; print x, y, swap(x,y), x, y
   .75 abc  abc .75

   ; A = list(1,2,3); mat B[3] = {4,5,6}; swap(A[[1]], B[1]); print A[[1]], B[1]
   5 2

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    assign
```

systime - kernel CPU time used by the current process

```
SYNOPSIS
    systime()

TYPES
    return  nonnegative real

DESCRIPTION
    In POSIX based systems, this function will return the CPU seconds
    used by the current process while in kernel mode executing kernel
    code on behalf of the current process.  Time spent by the current
    process executing user code (such as performing computation on
    behalf of calc) is not counted.

    On non-POSIX based systems, this function will always return 0.
    In particular, most MS windows based systems do not have the required
    POSIX system call and so this function will always return 0.

EXAMPLE
    The result for this example will depend on the speed of the CPU
    and precision of the operating CPU time accounting sub-system:

    ; t = systime();
    ; system("true"),
    ; systime() - t;
      .001

LIMITS
    On non-POSIX based systems, this function always returns 0.

LINK LIBRARY
    none

SEE ALSO
    config, ctime, runtime, systime, time
```

tail - create a list of specified size from the tail of a list

```
SYNOPSIS
   tail(x, y)

TYPES
    x       list
    y       int

    return  list

DESCRIPTION
    If 0 <= y <= size(x) == n, tail(x,y) returns a list of size y whose
    elements in succession have values x[[n - y]]. x[[1]], ..., x[[n - 1]].

    If y > size(x), tail(x,y) is a copy of x.

    If -size(x) < y < 0, tail(x,y) returns a list of size (size(x) + y)
    whose elements in succession have values x[[-y]]. x[[-y + 1]], ...,
    x[[size(x) - 1]], i.e. a copy of x from which the first -y members
    have been deleted.

    If y <= -size(x), tail(x,y) returns a list with no members.

    For any integer y, x == join(head(x,-y), tail(x,y)).

EXAMPLE
    ; A = list(2, 3, 5, 7, 11)
    ; tail(A, 2)

    list (2 members, 2 nonzero):
        [[0]] = 7
        [[1]] = 11

    ; tail(A, -2)

    list (3 members, 3 nonzero):
        [[0]] = 5
        [[1]] = 7
        [[2]] = 11

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    head, segment
```

tan - trigonometric tangent

```
SYNOPSIS
    tan(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the tangent of x to a multiple of eps, with error less
    in absolute value than .75 * eps.

EXAMPLE
    ; print tan(1, 1e-5), tan(1, 1e-10), tan(1, 1e-15), tan(1, 1e-20)
    1.55741 1.5574077247 1.557407724654902 1.55740772465490223051

LIMITS
    unlike sin and cos, x must be real
    eps > 0

LINK LIBRARY
    NUMBER *qtan(NUMBER *x, NUMBER *eps)

SEE ALSO
    sin, cos, sec, csc, cot, epsilon
```

tanh - hyperbolic tangent

```
SYNOPSIS
    tanh(x [,eps])

TYPES
    x       real
    eps           nonzero real, defaults to epsilon()

    return  real

DESCRIPTION
    Calculate the tanh of x to the nearest or next to nearest multiple of
    epsilon, with absolute error less than .75 * abs(eps).

    tanh(x) = (exp(2*x) - 1)/(exp(2*x) + 1)

EXAMPLE
    ; print tanh(1, 1e-5), tanh(1, 1e-10), tanh(1, 1e-15), tanh(1, 1e-20)
    .76159 .761594156 .761594155955765 .76159415595576488812

LIMITS
    unlike sin and cos, x must be real
    eps > 0

LINK LIBRARY
    NUMBER *qtanh(NUMBER *x, NUMBER *eps)

SEE ALSO
    sinh, cosh, sech, csch, coth, epsilon
```

test - whether a value is deemed to be true or false

```
SYNOPSIS
    test(x)

TYPES
    x       any

    return  0 or 1

DESCRIPTION
    This function returns 1 or 0 according as x tests as "true" or "false".

    Conditions under which a value x is considered to be false are:

      Numbers (real or complex):  x is zero

      String: x == ""

      Matrix: every component of x tests as false

      List: every element of x tests as false

      Association: x has no element

      File: x is not open

      Null: always

      Object of type xx: if xx_test has been defined, xx_test(x)
            returns zero;  if xx_test has not been defined,
            every element of x tests as false.

      Error-value or other types: never

EXAMPLE
    ; print test(27), test(0), test("abc"), test("")
    1 0 1 0

    ; print test(mat[3] = {1,,2}), test(mat[2][2])
    1 0

    ; A = list(0, 2, 0)
    ; print test(A), test(pop(A)), test(A), test(pop(A)), test(A)
    1 0 1 1 0

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    isassoc, isfile, isident, isnum, isint, islist, ismat, isnull, isobj,
    isreal, isstr, issimple, istype
```

time - number of seconds since the Epoch

```
SYNOPSIS
    time()

TYPES
    return  int

DESCRIPTION
    The time() builtin returns the number of seconds since the Epoch,
    which according to Posix is:

      Thr Jan      1 00:00:00 UTC 1970

EXAMPLE
    ; print time();
    831081380

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    ctime, runtime
```

trunc - truncate a value to a number of decimal places

```
SYNOPSIS
    trunc(x [,j])

TYPES
    x       real
    j       int

    return  real

DESCRIPTION
    Truncate x to j decimal places.  If j is omitted, 0 places is assumed.
    Specifying zero places makes the result identical to int().

    Truncation of a non-integer prodcues values nearer to zero.

EXAMPLE
    ; print trunc(pi()), trunc(pi(), 5)
    3 3.14159

    ; print trunc(3.333), trunc(3.789), trunc(3.333, 2), trunc(3.789, 2)
    3 3 3.33 3.78

    ; print trunc(-3.333), trunc(-3.789), trunc(-3.333, 2), trunc(-3.789, 2)
    -3 -3 -3.33 -3.78

LIMITS
    0 <= j < 2^31

LINK LIBRARY
    NUMBER *qtrunc(NUMBER *x, *j)

SEE ALSO
    bround, btrunc, int, round
```

usertime - user CPU time used by the current process

```
SYNOPSIS
    usertime()

TYPES
    return  nonnegative real

DESCRIPTION
    In POSIX based systems, this function will return the CPU seconds
    used by the current process while in user mode.  Time spent in the
    kernel executing system calls is not included.

    On non-POSIX based systems, this function will always return 0.
    In particular, most MS windows based systems do not have the required
    POSIX system call and so this function will always return 0.

EXAMPLE
    The result for this example will depend on the speed of the CPU
    and precision of the operating CPU time accounting sub-system:

    ; t = usertime();
    ; x = ptest(2^4253-1);
    ; usertime() - t;
      1.287804

LIMITS
    On non-POSIX based systems, this function always returns 0.

LINK LIBRARY
    none

SEE ALSO
    config, ctime, usertime, systime, time
```

version - return the calc version string

```
SYNOPSIS
    version()

TYPES
    return  string

DESCRIPTION
    Returns the calc version string.

    Calc version strings can be of the form:

      x.y.z.w
      x.y.z
      x.y

    where x, y, z, w, v are integers (without leading 0's) and,
    t is the the literal character 't'.

EXAMPLE
    ; version()
          "2.11.5.4"

LIMITS
    none

LINK LIBRARY
    none

SEE ALSO
    n/a
```

xor - bitwise exclusive or of a set of integers

```
SYNOPSIS
    xor(x1, x2, ...)

TYPES
    x1, x2, ... integer

    return  integer

DESCRIPTION
    Compute the bitwise exclusive or of a set of integers.

    For one argument xor(x1) returns x1.  For two arguments,
    xor(x1,x2) returns the bitwise exclusive or of x1 and x2.
    For each bit pair:

      0 0 xor returns 0
      0 1 xor returns 1
      1 0 xor returns 1
      1 1 xor returns 0

    For more than two arguments, xor(x1,x2,x3, ..., xn) returns:

      xor(...xor(xor(x1,x2), x3), ... xn)

EXAMPLE
    ; print xor(2), xor(5, 3, -7, 2, 9)
    2 10

LIMITS
    The number of arguments is not to exceed 1024.

LINK LIBRARY
    NUMBER *qxor(NUMBER *x1, NUMBER *x2)

SEE ALSO
    operator
```

config - configuration parameters

```
SYNOPSIS
    config(parameter [,value])

TYPES
    parameter     string
    value   int, string, config state

    return  config state

DESCRIPTION
    The config() builtin affects how the calculator performs certain
    operations.    Among features that are controlled by these parameters
    are the accuracy of some calculations, the displayed format of results,
    the choice from possible alternative algorithms, and whether or not
    debugging information is displayed.    The parameters are
    read or set using the "config" built-in function; they remain in effect
    until their values are changed by a config or equivalent instruction.

    The following parameters can be specified:

          "all"         all configuration values listed below

          "trace"       turns tracing features on or off
          "display"           sets number of digits in prints.
          "epsilon"           sets error value for transcendentals.
          "maxprint"          sets maximum number of elements printed.
          "mode"        sets printout mode.
          "mode2"       sets 2nd base printout mode.
          "mul2"        sets size for alternative multiply.
          "sq2"         sets size for alternative squaring.
          "pow2"        sets size for alternate powering.
          "redc2"       sets size for alternate REDC.
          "tilde"       enable/disable printing of the roundoff '~'
          "tab"         enable/disable printing of leading tabs
          "quomod"            sets rounding mode for quomod
          "quo"         sets rounding mode for //, default for quo
          "mod"         sets "rounding" mode for %, default for mod
          "sqrt"        sets rounding mode for sqrt
          "appr"        sets rounding mode for appr
          "cfappr"            sets rounding mode for cfappr
          "cfsim"       sets rounding mode for cfsim
          "round"       sets rounding mode for round and bround
          "outround"          sets rounding mode for printing of numbers
          "leadzero"          enables/disables printing of 0 as in 0.5
          "fullzero"          enables/disables padding zeros as in .5000
          "maxscan"           maximum number of scan errors before abort
          "prompt"            default interactive prompt
          "more"        default interactive multi-line input prompt
          "blkmaxprint" number of block octets to print, 0 means all
          "blkverbose"  TRUE => print all lines, FALSE=>skip duplicates
          "blkbase"           block output base
          "blkfmt"            block output format
          "calc_debug"  controls internal calc debug information
          "resource_debug"    controls resource file debug information
          "user_debug"  for user defined debug information
```

```
        "verbose_quit"      TRUE => print message on empty quit or abort
        "ctrl_d"            The interactive meaning of ^D (Control D)
        "program"           Read-only calc program or shell script path
        "basename"          Read-only basename of the program value
        "windows"           Read-only indicator of MS windows
        "cygwin"            TRUE=>calc compiled with cygwin, Read-only
        "compile_custom"    TRUE=>calc was compiled with custom functions
        "allow_custom"      TRUE=>custom functions are enabled
        "version"           Read-only calc version
        "baseb"       bits in calculation base, a read-only value
        "redecl_warn" TRUE => warn when redeclaring
        "dupvar_warn" TRUE => warn when variable names collide
        "hz"          Read-only operating system tick rate or 0
```

The "all" config value allows one to save/restore the configuration set of values. The return of:

```
config("all")
```

is a CONFIG type which may be used as the 2rd arg in a later call. One may save, modify and restore the configuration state as follows:

```
oldstate = config("all")
...
config("tab", 0)
config("mod", 10)
...
config("all", oldstate)
```

This save/restore method is useful within functions. It allows functions to control their configuration without impacting the calling function.

There are two configuration state aliases that may be set. To set the backward compatible standard configuration:

```
config("all", "oldstd")
```

The "oldstd" will restore the configuration to the default at startup.

A new configuration that some people prefer may be set by:

```
config("all", "newstd")
```

The "newstd" is not backward compatible with the historic configuration. Even so, some people prefer this configuration and place the config("all", "newstd") command in their CALCRC startup files; newstd may also be established by invoking calc with the flag -n.

The following are synonyms for true:

```
"on"
"true"
"t"
"yes"
"y"
"set"
"1"
```

```
      any non-zero number
```

The following are synonyms for false:

```
      "off"
      "false"
      "f"
      "no"
      "n"
      "unset"
      "0"
      the number zero (0)
```

Examples of setting some parameters are:

```
      config("mode", "exp");          exponential output
      config("display", 50);          50 digits of output
      epsilon(epsilon() / 8);         3 bits more accuracy
      config("tilde", 0)              disable roundoff tilde printing
      config("tab", "off")            disable leading tab printing
```

=-=

config("trace", bitflag)

When nonzero, the "trace" parameter activates one or more features that may be useful for debugging. These features correspond to powers of 2 which contribute additively to config("trace"):

```
  1: opcodes are displayed as functions are evaluated

  2: disables the inclusion of debug lines in opcodes for functions
     whose definitions are introduced with a left-brace.

  4: the number of links for real and complex numbers are displayed
     when the numbers are printed; for real numbers "#" or for
     complex numbers "##", followed by the number of links, are
     printed immediately after the number.

  8: the opcodes for a new functions are displayed when the function
     is successfully defined.
```

See also resource_debug, calc_debug and user_debug below for more debug levels.

=-=

config("display", int)

The "display" parameter specifies the maximum number of digits after the decimal point to be printed in real or exponential mode in normal unformatted printing (print, strprint, fprint) or in formatted printing (printf, strprintf, fprintf) when precision is not specified.   The initial value for oldstd is 20, for newstd 10. The parameter may be changed to the value d by either config("display", d) or by display (d). This parameter does not change the stored value of a number. Where rounding is necessary to display up to d decimal places, the type of rounding to be used is controlled by config("outround").

```
=-=

config("epsilon", real)
epsilon(real)

The "epsilon" parameter specifies the default accuracy for the
calculation of functions for which exact values are not possible or
not desired.  For most functions, the

        remainder = exact value - calculated value

has absolute value less than epsilon, but, except when the sign of
the remainder is controlled by an appropriate parameter, the
absolute value of the remainder usually does not exceed epsilon/2.
Functions which require an epsilon value accept an
optional argument which overrides this default epsilon value for
that single call.  The value v can be assigned to the "epsilon"
parameter by either config("epsilon", v) or epsilon(v); each of
these functions return the current epsilon value; config("epsilon")
or epsilon() returns but does not change the epsilon value.
For the transcendental functions and the functions sqrt() and
appr(), the calculated value is always a multiple of epsilon.

=-=

config("mode", "mode_string")
config("mode2", "mode_string")

The "mode" parameter is a string specifying the mode for printing of
numbers by the unformatted print functions, and the default
("%d" specifier) for formatted print functions.  The initial mode
is "real".    The available modes are:

    config("mode")  meaning                       equivalent
        string                                    base() call

      "binary"            base 2 fractions        base(2)
      "bin"

      "octal"       base 8 fractions        base(8)
      "oct"

      "real"        base 10 floating point        base(10)
      "float"
      "default"

      "integer"           base 10 integer               base(-10)
      "int"

      "hexadecimal" base 16 fractions       base(16)
      "hex"

      "fraction"          base 10 fractions       base(1/3)
      "frac"

      "scientific"  base 10 scientific notation   base(1e20)
      "sci"
      "exp"
```

```
Where multiple strings are given, the first string listed is what
config("mode") will return.

The "mode2" controls the double base output.  When set to a value
other than "off", calc outputs files in both the "base" mode as
well as the "base2" mode.  The "mode2" value may be any of the
"mode" values with the addition of:

      "off"         disable 2nd base output mode  base2(0)

The base() builtin function sets and returns the "mode" value.
The base2() builtin function sets and returns the "mode2" value.

The default "mode" is "real".  The default "mode2" is "off".

=-=

config("maxprint", int)

The "maxprint" parameter specifies the maximum number of elements to
be displayed when a matrix or list is printed.  The initial value is 16.

=-=

config("mul2", int)
config("sq2", int)

Mul2 and sq2 specify the sizes of numbers at which calc switches
from its first to its second algorithm for multiplying and squaring.
The first algorithm is the usual method of cross multiplying, which
runs in a time of O(N^2).  The second method is a recursive and
complicated method which runs in a time of O(N^1.585).  The argument
for these parameters is the number of binary words at which the
second algorithm begins to be used.  The minimum value is 2, and
the maximum value is very large.  If 2 is used, then the recursive
algorithm is used all the way down to single digits, which becomes
slow since the recursion overhead is high.  If a number such as
1000000 is used, then the recursive algorithm is almost never used,
causing calculations for large numbers to slow down.

Units refer to internal calculation digits where each digit
is BASEB bits in length.  The value of BASEB is returned by
config("baseb").

The default value for config("sq2") is 3388.  This default was
established on a 1.8GHz AMD 32-bit CPU of ~3406 BogoMIPS when
the two algorithms are about equal in speed.  For that CPU test,
config("baseb") was 32.  This means that by default numbers up to
(3388*32)+31 = 108447 bits in length (< 32645 decimal digits) use
the 1st algorithm, for squaring.

The default value for config("mul2") is 1780.  This default was
established on a 1.8GHz AMD 32-bit CPU of ~3406 BogoMIPS when
the two algorithms are about equal in speed.  For that CPU test,
config("baseb") was 32.  This means that by default numbers up to
(1779*32)+31 = 56927 bits in length (< 17137 decimal digits) use
the 1st algorithm, for multiplication.
```

A value of zero resets the parameter back to their default values.

The value of 1 and values < 0 are reserved for future use.

Usually there is no need to change these parameters.

=-=

config("pow2", int)

Pow2 specifies the sizes of numbers at which calc switches from its first to its second algorithm for calculating powers modulo another number. The first algorithm for calculating modular powers is by repeated squaring and multiplying and dividing by the modulus. The second method uses the REDC algorithm given by Peter Montgomery which avoids divisions. The argument for pow2 is the size of the modulus at which the second algorithm begins to be used.

Units refer to internal calculation digits where each digit is BASEB bits in length. The value of BASEB is returned by config("baseb").

The default value for config("pow2") is 176. This default was established on a 1.8GHz AMD 32-bit CPU of ~3406 BogoMIPS when the two algorithms are about equal in speed. For that CPU test, config("baseb") was 32. This means that by default numbers up to (176*32)+31 = 5663 bits in length (< 1704 decimal digits) use the 1st algorithm, for calculating powers modulo another number.

A value of zero resets the parameter back to their default values.

The value of 1 and values < 0 are reserved for future use.

Usually there is no need to change these parameters.

=-=

config("redc2", int)

Redc2 specifies the sizes of numbers at which calc switches from its first to its second algorithm when using the REDC algorithm. The first algorithm performs a multiply and a modular reduction together in one loop which runs in O(N^2). The second algorithm does the REDC calculation using three multiplies, and runs in O(N^1.585). The argument for redc2 is the size of the modulus at which the second algorithm begins to be used.

Units refer to internal calculation digits where each digit is BASEB bits in length. The value of BASEB is returned by config("baseb").

The default value for config("redc2") is 220. This default was established as 5/4 (the historical ratio of config("pow2") to config("pow2")) of the config("pow2") value. This means that if config("baseb") is 32, then by default numbers up to (220*32)+31 = 7071 bits in length (< 2128 decimal digits) use the REDC algorithm, for calculating powers modulo another number.

A value of zero resets the parameter back to their default values.

```
The value of 1 and values < 0 are reserved for future use.

Usually there is no need to change these parameters.

=-=

config("tilde", boolean)

Config("tilde") controls whether or not a leading tilde ('~') is
printed to indicate that a number has not been printed exactly
because the number of decimal digits required would exceed the
specified maximum number.  The initial "tilde" value is 1.

=-=

config("tab", boolean)

Config ("tab") controls the printing of a tab before results
automatically displayed when working interactively.      It does not
affect the printing by the functions print, printf, etc.  The initial
"tab" value is 1.

=-=

config("quomod", bitflag)
config("quo", bitflag)
config("mod", bitflag)
config("sqrt", bitflag)
config("appr", bitflag)
config("cfappr", bitflag)
config("cfsim", bitflag)
config("outround", bitflag)
config("round", bitflag)

The "quomod", "quo", "mod", "sqrt", "appr", "cfappr", "cfsim", and
"round" control the way in which any necessary rounding occurs.
Rounding occurs when for some reason, a calculated or displayed
value (the "approximation") has to differ from the "true value",
e.g. for quomod and quo, the quotient is to be an integer, for sqrt
and appr, the approximation is to be a multiple of an explicit or
implicit "epsilon", for round and bround (both controlled by
config("round")) the number of decimal places or fractional bits
in the approximation is limited.  Zero value for any of these
parameters indicates that the true value is greater than the approximation,
i.e. the rounding is "down", or in the case of mod, that the
residue has the same sign as the divisor.  If bit 4 of the
parameter is set, the rounding of to the nearest acceptable candidate
when this is uniquely determined; in the remaining ambiguous cases,
the type of rounding is determined by the lower bits of the parameter
value.  If bit 3 is set, the rounding for quo, appr and sqrt,
is to the nearest even integer or the nearest even multiple of epsilon,
and for round to the nearest even "last decimal place".  The effects
of the 3 lowest bits of the parameter value are as follows:

  Bit 0: Unconditional reversal (down to up, even to odd, etc.)
  Bit 1: Reversal if the exact value is negative
  Bit 2: Reversal if the divisor or epsilon is negative
```

```
(Bit 2 is irrelevant for the functions round and bround since the
equivalent epsilon (a power of 1/10 or 1/2) is always positive.)

For quomod, the quotient is rounded to an integer value as if
evaluating quo with config("quo") == config("quomod").  Similarly,
quomod and mod give the same residues if config("mod") == config("quomod").

For the sqrt function, if bit 5 of config("sqrt") is set, the exact
square-root is returned when this is possible; otherwise the
result is rounded to a multiple of epsilon as determined by the
five lower order bits.  Bit 6 of config("sqrt") controls whether the
principal or non-principal square-root is returned.

For the functions cfappr and cfsim, whether the "rounding" is down
or up, etc. is controlled by the appropriate bits of config("cfappr")
and config("cfsim") as for quomod, quo, etc.

The "outround" parameter determines the type of rounding to be used
by the various kinds of printing to the output: bits 0, 1, 3 and 4
are used in the same way as for the functions round and bround.

The C language method of modulus and integer division is:

      config("quomod", 2)
      config("quo", 2)
      config("mod", 2)

=-=

config("leadzero", boolean)

The "leadzero" parameter controls whether or not a 0 is printed
before the decimal point in non-zero fractions with absolute value
less than 1, e.g. whether 1/2 is printed as 0.5 or .5.   The
initial value is 0, corresponding to the printing .5.

=-=

config("fullzero", boolean)

The "fullzero" parameter controls whether or not in decimal floating-
point printing, the digits are padded with zeros to reach the
number of digits specified by config("display") or by a precision
specification in formatted printing.  The initial value for this
parameter is 0, so that, for example, if config("display") >= 2,
5/4 will print in "real" mode as 1.25.

=-=

config("maxscan", int)

The maxscan value controls how many scan errors are allowed
before the compiling phase of a computation is aborted.  The initial
value of "maxscan" is 20.  Setting maxscan to 0 disables this feature.

=-=

config("prompt", str)
```

```
The default prompt when in interactive mode is "> ".  One may change
this prompt to a more cut-and-paste friendly prompt by:

      config("prompt", "; ")

On windowing systems that support cut/paste of a line, one may
cut/copy an input line and paste it directly into input.  The
leading ';' will be ignored.

=-=

config("more", str)

When inside multi-line input, the more prompt is used.  One may
change it by:

      config("more", ";; ")

=-=

config("blkmaxprint", int)

The "blkmaxprint" config value limits the number of octets to print
for a block.  A "blkmaxprint" of 0 means to print all octets of a
block, regardless of size.

The default is to print only the first 256 octets.

=-=

config("blkverbose", boolean)

The "blkverbose" determines if all lines, including duplicates
should be printed.  If TRUE, then all lines are printed.  If false,
duplicate lines are skipped and only a "*" is printed in a style
similar to od.  This config value has not meaning if "blkfmt" is "str".

The default value for "blkverbose" is FALSE: duplicate lines are
not printed.

=-=

config("blkbase", "blkbase_string")

The "blkbase" determines the base in which octets of a block
are printed.  Possible values are:

  "hexadecimal"           Octets printed in 2 digit hex
  "hex"
  "default"

  "octal"                 Octets printed in 3 digit octal
  "oct"

  "character"       Octets printed as chars with non-printing
  "char"                      chars as \123 or \n, \t, \r

  "binary"          Octets printed as 0 or 1 chars
  "bin"
```

```
  "raw"             Octets printed as is, i.e. raw binary
  "none"

Where multiple strings are given, the first string listed is what
config("blkbase") will return.

The default "blkbase" is "hexadecimal".

=-=

config("blkfmt", "blkfmt_string")

The "blkfmt" determines for format of how block are printed:

  "lines"           print in lines of up to 79 chars + newline
  "line"

  "strings"   print as one long string
  "string"
  "str"

  "od_style"  print in od-like format, with leading offset,
  "odstyle"       followed by octets in the given base
  "od"

  "hd_style"  print in hex dump format, with leading offset,
  "hdstyle"       followed by octets in the given base, followed
  "hd"            by chars or '.' if no-printable or blank
  "default"

Where multiple strings are given, the first string listed is what
config("blkfmt") will return.

The default "blkfmt" is "hd_style".

=-=

config("calc_debug", bitflag)

The "calc_debug" is intended for controlling internal calc routines
that test its operation, or collect or display information that
might be useful for debug purposes.    Much of the output from these
will make sense only to calc wizards.   Zero value (the default for
both oldstd and newstd) of config("resource_debug") corresponds to
switching off all these routines.  For nonzero value, particular
bits currently have the following meanings:

  n           Meaning of bit n of config("calc_debug")

  0     outputs shell commands prior to execution

  1     outputs currently active functions when a quit instruction
        is executed

  2     some details of hash states are included in the output
        when these are printed

  3     when a function constructs a block value, tests are
```

```
        made that the result has the properties required for use of
        that block, e.g. that the pointer to the start of the
        block is not NULL, and that its "length" is not negative.
        A failure will result in a runtime error.

  4     Report on changes to the state of stdin as well as changes
        to internal variables that control the setting and restoring
        of stdin.

  5     Report on changes to the run state of calc.

  6     Report on rand() subtractive 100 shuffle generator issues.

  7     Report on custom function issues.

Bits >= 8 are reserved for future use and should not be used at this time.

By default, "calc_debug" is 0.  The initial value may be overridden
by the -D command line option.

=-=

config("resource_debug", bitflag)
config("lib_debug", bitflag)

The "resource_debug" parameter is intended for controlling the possible
display of special information relating to functions, objects, and
other structures created by instructions in calc scripts.
Zero value of config("resource_debug") means that no such information
is displayed.  For other values, the non-zero bits which currently
have meanings are as follows:

  n           Meaning of bit n of config("resource_debug")

  0     When a function is defined, redefined or undefined at
        interactive level, a message saying what has been done
        is displayed.

  1     When a function is defined, redefined or undefined during
        the reading of a file, a message saying what has been done
        is displayed.

  2     Show func will display more information about a functions
        arguments and argument summary information.

  3     During execution, allow calc standard resource files
        to output additional debugging information.

The value for config("resource_debug") in both oldstd and newstd
is 3, but if calc is invoked with the -d flag, its initial value
is zero.  Thus, if calc is started without the -d flag, until
config("resource_debug") is changed, a message will be output when
a function is defined either interactively or during the reading of
a file.

The name config("lib_debug") is equivalent to config("resource_debug")
and is included for backward compatibility.

By default, "resource_debug" is 3.    The -d flag changes this default to 0.
```

```
The initial value may be overridden by the -D command line option.

=-=

config("user_debug", int)

The "user_debug" is provided for use by users.  Calc ignores this value
other than to set it to 0 by default (for both "oldstd" and "newstd").
No calc code or standard resource should change this value.    Users
should feel free to use it in any way.   In particular they may
use particular bits for special purposes as with "calc_debug", or
they may use it to indicate a debug level with larger values
indicating more stringent and more informative tests with presumably
slower operation or more memory usage, and a particular value (like
-1 or 0) corresponding to "no tests".

By default, "user_debug" is 0.  The initial value may be overridden
by the -D command line option.

=-=

config("verbose_quit", boolean)

The "verbose_quit" controls the print of the message:

  quit or abort executed

when a non-interactive quit or abort without an argument is encountered.
A quit of abort without an argument does not display a message when
invoked at the interactive level.

By default, "verbose_quit" is false.

=-=

config("ctrl_d", "ctrl_d_string")

For calc that is using the calc binding (not GNU-readline) facility:

  The "ctrl_d" controls the interactive meaning of ^D (Control D):

      "virgin_eof"  If ^D is the only character that has been typed
      "virgineof"     on a line, then calc will exit.  Otherwise ^D
      "virgin"        will act according to the calc binding, which
      "default"       by default is a Emacs-style delete-char.

      "never_eof"     The ^D never exits calc and only acts according
      "nevereof"      calc binding, which by default is a Emacs-style
      "never"   delete-char.

      "empty_eof"     The ^D always exits calc if typed on an empty line.
      "emptyeof"      This condition occurs when ^D either the first
      "empty"   character typed, or when all other characters on
                the line have been removed (say by deleting them).

  Where multiple strings are given, the first string listed is what
  config("ctrl_d") will return.

  Note that config("ctrl_d") actually controls each and every character
```

```
  that is bound to ``delete_char''.  By default, ``delete_char'' is
  Control D.  Any character(s) bound to ``delete_char'' will cause calc
  to exit (or not exit) as directed by config("ctrl_d").

  See the ``binding'' help for information on the default calc bindings.

  The default "ctrl_d", without GNU-readline is "virgin_eof".

For calc that was compiled with the GNU-readline facility:

  The "ctrl_d" controls the interactive meaning of ^D (Control D):

      "virgin_eof"  Same as "empty_eof"
      "virgineof"
      "virgin"
      "default"

      "never_eof"     The ^D never exits calc and only acts according
      "nevereof"      calc binding, which by default is a Emacs-style
      "never"   delete-char.

      "empty_eof"     The ^D always exits calc if typed on an empty line.
      "emptyeof"      This condition occurs when ^D either the first
      "empty"   character typed, or when all other characters on

  Where multiple strings are given, the first string listed is what
  config("ctrl_d") will return.

  The default "ctrl_d", with GNU-readline is effectively "empty_eof".

  Literally it is "virgin_eof", but since "virgin_eof" is the
  same as "empty_eof", the default is effectively "empty_eof".

Emacs users may find the default behavior objectionable, particularly
when using the GNU-readline facility.  Such users may want to add the line:

  config("ctrl_d", "never_eof"),;

to their ~/.calcrc startup file to prevent ^D from causing calc to exit.

=-=

config("program")         <== NOTE: This is a read-only config value

The full path to the calc program, or the calc shell script can be
obtained by:

  config("program")

This config parameter is read-only and cannot be set.

=-=

config("basename")        <== NOTE: This is a read-only config value

The calc program, or the calc shell script basename can be obtained by:

  config("basename")
```

```
The config("basename") is the config("program") without any leading
path.  If config("program") has a / in it, config("basename") is
everything after the last /, otherwise config("basename") is the
same as config("program").

This config parameter is read-only and cannot be set.

=-=

config("windows")         <== NOTE: This is a read-only config value

Returns TRUE if you are running on a MS windows system, false if you
are running on an operating system that does not hate you.

This config parameter is read-only and cannot be set.

=-=

config("cygwin")          <== NOTE: This is a read-only config value

Returns TRUE if you calc was compiled with cygwin, false otherwise.

This config parameter is read-only and cannot be set.

=-=

config("compile_custom")  <== NOTE: This is a read-only config value

Returns TRUE if you calc was compiled with -DCUSTOM.  By default,
the calc Makefile uses ALLOW_CUSTOM= -DCUSTOM so by default
config("compile_custom") is TRUE.  If, however, calc is compiled
without -DCUSTOM, then config("compile_custom") will be FALSE.

The config("compile_custom") value is only affected by compile
flags.   The calc -D runtime command line option does not change
the config("compile_custom") value.

See also config("allow_custom").

This config parameter is read-only and cannot be set.

=-=

config("allow_custom")    <== NOTE: This is a read-only config value

Returns TRUE if you custom functions are enabled.  To allow the use
of custom functions, calc must be compiled with -DCUSTOM (which it
is by default) AND calc run be run with the -D runtime command line
option (which it is not by default).

If config("allow_custom") is TRUE, then custom functions are allowed.
If config("allow_custom") is FALSE, then custom functions are not
allowed.

See also config("compile_custom").

This config parameter is read-only and cannot be set.

=-=
```

```
    config("version")         <== NOTE: This is a read-only config value

    The version string of the calc program can be obtained by:

      config("version")

    This config parameter is read-only and cannot be set.

    =-=

    config("baseb")           <== NOTE: This is a read-only config value

    Returns the number of bits in the fundamental base in which
    internal calculations are performed.  For example, a value of
    32 means that calc will perform many internal calculations in
    base 2^32 with digits that are 32 bits in length.

    For libcalc programmers, this is the value of BASEB as defined
    in the zmath.h header file.

    This config parameter is read-only and cannot be set.

    =-=

    config("redecl_warn", boolean)

    Config("redecl_warn") controls whether or not a warning is issued
    when redeclaring variables.

    The initial "redecl_warn" value is 1.

    =-=

    config("dupvar_warn", boolean)

    Config("dupvar_warn") controls whether or not a warning is issued
    when a variable name collides with an exist name of a higher scope.
    Examples of collisions are when:

      * both local and static variables have the same name
      * both local and global variables have the same name
      * both function parameter and local variables have the same name
      * both function parameter and global variables have the same name

    The initial "redecl_warn" value is 1.

    =-=

    config("hz")        <== NOTE: This is a read-only config value

    Returns the rate at which the operating system advances the clock
    on POSIX based systems.  Returns 0 on non-POSIX based systems.
    The non-zero value returned is in Hetrz.

    This config parameter is read-only and cannot be set.


EXAMPLE
```

```
; current_cfg = config("all");
; config("tilde", off),;
; config("calc_debug", 15),;
; config("all") == current_cfg
  0
; config("all", current_cfg),;
; config("all") == current_cfg
  1

; config("version")
        "2.12.0"

; config("all")
  mode             "real"
  mode2            "off"
  display          20
  epsilon          0.00000000000000000001
  trace            0
  maxprint         16
  mul2             20
  sq2              20
  pow2             40
  redc2            50
  tilde            1
  tab              1
  quomod           0
  quo              2
  mod              0
  sqrt             24
  appr             24
  cfappr           0
  cfsim            8
  outround         24
  round            24
  leadzero         1
  fullzero         0
  maxscan          20
  prompt           "; "
  more             ";; "
  blkmaxprint      256
  blkverbose       0
  blkbase          "hexadecimal"
  blkfmt           "hd_style"
  resource_debug   3
  lib_debug        3
  calc_debug       0
  user_debug       0
  verbose_quit     0
  ctrl_d           "virgin_eof"
  program          "calc"
  basename         "calc"
  windows          0
  cygwin           0
  compile_custom   1
  allow_custom     0
  version          "2.12.0"
  baseb       32
  redecl_warn 1
  dupvar_warn 1
```

```
      hz          100

    ; display()
      20
    ; config("display", 50),;
    ; display()
      50
```

LIMITS

    none

LINK LIBRARY

    n/a

SEE ALSO

    usage, custom, custom_cal, usage, epsilon, display

Calc generated error codes (see the error help file):

```
10001   Division by zero
10002   Indeterminate (0/0)
10003   Bad arguments for +
10004   Bad arguments for binary -
10005   Bad arguments for *
10006   Bad arguments for /
10007   Bad argument for unary -
10008   Bad argument for squaring
10009   Bad argument for inverse
10010   Bad argument for ++
10011   Bad argument for --
10012   Bad argument for int
10013   Bad argument for frac
10014   Bad argument for conj
10015   Bad first argument for appr
10016   Bad second argument for appr
10017   Bad third argument for appr
10018   Bad first argument for round
10019   Bad second argument for round
10020   Bad third argument for round
10021   Bad first argument for bround
10022   Bad second argument for bround
10023   Bad third argument for bround
10024   Bad first argument for sqrt
10025   Bad second argument for sqrt
10026   Bad third argument for sqrt
10027   Bad first argument for root
10028   Bad second argument for root
10029   Bad third argument for root
10030   Bad argument for norm
10031   Bad first argument for << or >>
10032   Bad second argument for << or >>
10033   Bad first argument for scale
10034   Bad second argument for scale
10035   Bad first argument for ^
10036   Bad second argument for ^
10037   Bad first argument for power
10038   Bad second argument for power
10039   Bad third argument for power
10040   Bad first argument for quo or //
10041   Bad second argument for quo or //
10042   Bad third argument for quo
10043   Bad first argument for mod or %
10044   Bad second argument for mod or %
10045   Bad third argument for mod
10046   Bad argument for sgn
10047   Bad first argument for abs
10048   Bad second argument for abs
10049   Scan error in argument for eval
10050   Non-simple type for str
10051   Non-real epsilon for exp
10052   Bad first argument for exp
10053   Non-file first argument for fputc
10054   Bad second argument for fputc
10055   File not open for writing for fputc
```

10056	Non-file first argument for fgetc
10057	File not open for reading for fgetc
10058	Non-string arguments for fopen
10059	Unrecognized mode for fopen
10060	Non-file first argument for freopen
10061	Non-string or unrecognized mode for freopen
10062	Non-string third argument for freopen
10063	Non-file argument for fclose
10064	Non-file argument for fflush
10065	Non-file first argument for fputs
10066	Non-string argument after first for fputs
10067	File not open for writing for fputs
10068	Non-file argument for fgets
10069	File not open for reading for fgets
10070	Non-file first argument for fputstr
10071	Non-string argument after first for fputstr
10072	File not open for writing for fputstr
10073	Non-file first argument for fgetstr
10074	File not open for reading for fgetstr
10075	Non-file argument for fgetline
10076	File not open for reading for fgetline
10077	Non-file argument for fgetfield
10078	File not open for reading for fgetfield
10080	Non-integer argument for files
10081	Non-string fmt argument for fprint
10082	Stdout not open for writing to ???
10083	Non-file first argument for fprintf
10084	Non-string second (fmt) argument for fprintf
10085	File not open for writing for fprintf
10086	Non-string first (fmt) argument for strprintf
10087	Error in attempting strprintf ???
10088	Non-file first argument for fscan
10089	File not open for reading for fscan
10090	Non-string first argument for strscan
10091	Non-file first argument for fscanf
10092	Non-string second (fmt) argument for fscanf
10093	Non-lvalue argument after second for fscanf
10094	File not open for reading or other error for fscanf
10095	Non-string first argument for strscanf
10096	Non-string second (fmt) argument for strscanf
10097	Non-lvalue argument after second for strscanf
10098	Some error in attempting strscanf ???
10099	Non-string first (fmt) argument for scanf
10100	Non-lvalue argument after first for scanf
10101	Some error in attempting scanf ???
10102	Non-file argument for ftell
10103	File not open or other error for ftell
10104	Non-file first argument for fseek
10105	Non-integer or negative second argument for fseek
10106	File not open or other error for fseek
10107	Non-file argument for fsize
10108	File not open or other error for fsize
10109	Non-file argument for feof
10110	File not open or other error for feof
10111	Non-file argument for ferror
10112	File not open or other error for ferror
10113	Non-file argument for ungetc
10114	File not open for reading for ungetc
10115	Bad second argument or other error for ungetc

```
10116   Exponent too big in scanning
10119   Non-string first argument for access
10120   Bad second argument for access
10121   Bad first argument for search
10122   Bad second argument for search
10123   Bad third argument for search
10124   Bad fourth argument for search
10125   Cannot find fsize or fpos for search
10126   File not readable for search
10127   Bad first argument for rsearch
10128   Bad second argument for rsearch
10129   Bad third argument for rsearch
10130   Bad fourth argument for rsearch
10131   Cannot find fsize or fpos for rsearch
10132   File not readable for rsearch
10133   Too many open files
10135   Bad argument type for strerror
10136   Index out of range for strerror
10137   Bad epsilon for cos
10138   Bad first argument for cos
10139   Bad epsilon for sin
10140   Bad first argument for sin
10141   Non-string argument for eval
10142   Bad epsilon for arg
10143   Bad first argument for arg
10144   Non-real argument for polar
10145   Bad epsilon for polar
10146   Non-integral argument for fcnt
10147   Non-variable first argument for matfill
10148   Non-matrix first argument-value for matfill
10149   Non-matrix argument for matdim
10150   Non-matrix argument for matsum
10151   E_ISIDENT is no longer used
10152   Non-matrix argument for mattrans
10153   Non-two-dimensional matrix for mattrans
10154   Non-matrix argument for det
10155   Matrix for det not of dimension 2
10156   Non-square matrix for det
10157   Non-matrix first argument for matmin
10158   Non-positive-integer second argument for matmin
10159   Second argument for matmin exceeds dimension
10160   Non-matrix first argument for matmin
10161   Second argument for matmax not positive integer
10162   Second argument for matmax exceeds dimension
10163   Non-matrix argument for cp
10164   Non-one-dimensional matrix for cp
10165   Matrix size not 3 for cp
10166   Non-matrix argument for dp
10167   Non-one-dimensional matrix for dp
10168   Different-size matrices for dp
10169   Non-string argument for strlen
10170   Non-string argument for strcat
10171   Non-string first argument for strcat
10172   Non-non-negative integer second argument for strcat
10173   Bad argument for char
10174   Non-string argument for ord
10175   Non-list-variable first argument for insert
10176   Non-integral second argument for insert
10177   Non-list-variable first argument for push
```

10178 Non-list-variable first argument for append
10179 Non-list-variable first argument for delete
10180 Non-integral second argument for delete
10181 Non-list-variable argument for pop
10182 Non-list-variable argument for remove
10183 Bad epsilon argument for ln
10184 Non-numeric first argument for ln
10185 Non-integer argument for error
10186 Argument outside range for error
10187 Attempt to eval at maximum input depth
10188 Unable to open string for reading
10191 Operation allowed because calc mode disallows read operations
10192 Operation allowed because calc mode disallows write operations
10193 Operation allowed because calc mode disallows exec operations
10194 Unordered arguments for min
10195 Unordered arguments for max
10196 Unordered items for minimum of list
10197 Unordered items for maximum of list
10198 Size undefined for argument type
10199 Calc must be run with a -C argument to use custom function
10200 Calc was built with custom functions disabled
10201 Custom function unknown, try: show custom
10202 Non-integral length for block
10203 Negative or too-large length for block
10204 Non-integral chunksize for block
10205 Negative or too-large chunksize for block
10206 Named block does not exist for blkfree
10207 Non-integral id specification for blkfree
10208 Block with specified id does not exist
10209 Block already freed
10210 No-realloc protection prevents blkfree
10211 Non-integer argument for blocks
10212 Non-allocated index number for blocks
10213 Non-integer or negative source index for copy
10214 Source index too large for copy
10215 E_COPY3 is no longer used
10216 Non-integer or negative number for copy
10217 Number too large for copy
10218 Non-integer or negative destination index for copy
10219 Destination index too large for copy
10220 Freed block source for copy
10221 Unsuitable source type for copy
10222 Freed block destinction for copy
10223 Unsuitable destination type for copy
10224 Incompatible source and destination for copy
10225 No-copy-from source variable
10226 No-copy-to destination variable
10227 No-copy-from source named block
10228 No-copy-to destination named block
10229 No-relocate destination for copy
10230 File not open for copy
10231 fseek or fsize failure for copy
10232 fwrite error for copy
10233 fread error for copy
10234 Non-variable first argument for protect
10235 Bad second argument for protect
10236 Bad third argument for protect
10237 No-copy-to destination for matfill
10238 No-assign-from source for matfill

10239	Non-matrix argument for mattrace
10240	Non-two-dimensional argument for mattrace
10241	Non-square argument for mattrace
10242	Bad epsilon for tan
10243	Bad argument for tan
10244	Bad epsilon for cot
10245	Bad argument for cot
10246	Bad epsilon for sec
10247	Bad argument for sec
10248	Bad epsilon for csc
10249	Bad argument for csc
10250	Bad epsilon for sinh
10251	Bad argument for sinh
10252	Bad epsilon for cosh
10253	Bad argument for cosh
10254	Bad epsilon for tanh
10255	Bad argument for tanh
10256	Bad epsilon for coth
10257	Bad argument for coth
10258	Bad epsilon for sech
10259	Bad argument for sech
10260	Bad epsilon for csch
10261	Bad argument for csch
10262	Bad epsilon for asin
10263	Bad argument for asin
10264	Bad epsilon for acos
10265	Bad argument for acos
10266	Bad epsilon for atan
10267	Bad argument for atan
10268	Bad epsilon for acot
10269	Bad argument for acot
10270	Bad epsilon for asec
10271	Bad argument for asec
10272	Bad epsilon for acsc
10273	Bad argument for acsc
10274	Bad epsilon for asin
10275	Bad argument for asinh
10276	Bad epsilon for acosh
10277	Bad argument for acosh
10278	Bad epsilon for atanh
10279	Bad argument for atanh
10280	Bad epsilon for acoth
10281	Bad argument for acoth
10282	Bad epsilon for asech
10283	Bad argument for asech
10284	Bad epsilon for acsch
10285	Bad argument for acsch
10286	Bad epsilon for gd
10287	Bad argument for gd
10288	Bad epsilon for agd
10289	Bad argument for agd
10290	Log of zero or infinity
10291	String addition failure
10292	String multiplication failure
10293	String reversal failure
10294	String subtraction failure
10295	Bad argument type for bit
10296	Index too large for bit
10297	Non-integer second argument for setbit

10298	Out-of-range index for setbit
10299	Non-string first argument for setbit
10300	Bad argument for or
10301	Bad argument for and
10302	Allocation failure for string or
10303	Allocation failure for string and
10304	Bad argument for xorvalue
10305	Bad argument for comp
10306	Allocation failure for string diff
10307	Allocation failure for string comp
10308	Bad first argument for segment
10309	Bad second argument for segment
10310	Bad third argument for segment
10311	Failure for string segment
10312	Bad argument type for highbit
10313	Non-integer argument for highbit
10314	Bad argument type for lowbit
10315	Non-integer argument for lowbit
10316	Bad argument type for unary hash op
10317	Bad argument type for binary hash op
10318	Bad first argument for head
10319	Bad second argument for head
10320	Failure for strhead
10321	Bad first argument for tail
10322	Bad second argument for tail
10323	Failure for strtail
10324	Failure for strshift
10325	Non-string argument for strcmp
10326	Bad argument type for strncmp
10327	Varying types of argument for xor
10328	Bad argument type for xor
10329	Bad argument type for strcpy
10330	Bad argument type for strncpy
10331	Bad argument type for unary backslash
10332	Bad argument type for setminus
10333	Bad first argument type for indices
10334	Bad second argument for indices
10335	Too-large re(argument) for exp
10336	Too-large re(argument) for sinh
10337	Too-large re(argument) for cosh
10338	Too-large im(argument) for sin
10339	Too-large im(argument) for cos
10340	Infinite or too-large result for gd
10341	Infinite or too-large result for agd
10342	Too-large value for power
10343	Too-large value for root
10344	Non-real first arg for digit
10345	Non-integral second arg for digit
10346	Bad third arg for digit
10347	Bad first argument for places
10348	Bad second argument for places
10349	Bad first argument for digits
10350	Bad second argument for digits
10351	Bad first argument for ilog
10352	Bad second argument for ilog
10353	Bad argument for ilog10
10354	Bad argument for ilog2
10355	Non-integer second arg for comb
10356	Too-large second arg for comb

```
10357   Bad argument for catalan
10358   Bad argument for bern
10359   Bad argument for euler
10360   Bad argument for sleep
10362   No-copy-to destination for octet assign
10363   No-copy-from source for octet assign
10364   No-change destination for octet assign
10365   Non-variable destination for assign
10366   No-assign-to destination for assign
10367   No-assign-from source for assign
10368   No-change destination for assign
10369   No-type-change destination for assign
10370   No-error-value destination for assign
10371   No-copy argument for octet swap
10372   No-assign-to-or-from argument for swap
10373   Non-lvalue argument for swap
10374   Non-lvalue argument 3 or 4 for quomod
10375   Non-real-number arg 1 or 2 or bad arg 5 for quomod
10376   No-assign-to argument 3 or 4 for quomod
10377   No-copy-to or no-change argument for octet preinc
10378   Non-variable argument for preinc
10379   No-assign-to or no-change argument for preinc
10380   No-copy-to or no-change argument for octet predec
10381   Non-variable argument for predec
10382   No-assign-to or no-change argument for predec
10383   No-copy-to or no-change argument for octet postinc
10384   Non-variable argument for postinc
10385   No-assign-to or no-change argument for postinc
10386   No-copy-to or no-change argument for octet postdec
10387   Non-variable argument for postdec
10388   No-assign-to or no-change argument for postdec
10389   Error-type structure for initialization
10390   No-copy-to structure for initialization
10391   Too many initializer values
10392   Attempt to initialize freed named block
10393   Bad structure type for initialization
10394   No-assign-to element for initialization
10395   No-change element for initialization
10396   No-type-change element for initialization
10397   No-error-value element for initialization
10398   No-assign-or-copy-from source for initialization
10399   No-relocate for list insert
10400   No-relocate for list delete
10401   No-relocate for list push
10402   No-relocate for list append
10403   No-relocate for list pop
10404   No-relocate for list remove
10405   Non-variable first argument for modify
10406   Non-string second argument for modify
10407   No-change first argument for modify
10408   Undefined function for modify
10409   Unacceptable type first argument for modify
10410   Non-string arguments for fpathopen
10411   Unrecognized mode for fpathopen
10412   Bad epsilon argument for log
10413   Non-numeric first argument for log
10414   Non-file argument for fgetfile
10415   File argument for fgetfile not open for reading
10416   Unable to set file position in fgetfile
```

```
10417   Non-representable type for estr
20000   base of user defined errors
```

calc - arbitrary precision calculator

```
SYNOPSIS
       calc [-c] [-C] [-d]
          [-D calc_debug[:resource_debug[:user_debug]]]
          [-e] [-h] [-i] [-m mode] [-O]
          [-p] [-q] [-s] [-u] [-v] [[--] calc_cmd ...]

       #!/usr/bin/calc [other_flags ...] -f

DESCRIPTION

       CALC OPTIONS



       -c      Continue reading command lines even after a scan/parse error has
            caused the abandonment of a line.    Note that  this  option  only
            deals  with  scanning and parsing of the calc language.  It does
            not deal with execution or run-time errors.

            For example:

               calc read many_errors.cal

            will cause calc to abort on the first syntax error, whereas:

               calc -c read many_errors.cal

            will cause calc to try to process each line being      read  despite
            the scan/parse errors that it encounters.

            By  default, calc startup resource files are silently ignored if
            not found.  This flag will report missing startup resource files
            unless -d is also given.


       -C      Permit  the execution of custom builtin functions.  Without this
            flag, calling the custom() builtin function will simply generate
            an error.

            Use  of  this  flag may cause calc to execute functions that are
            non-standard and that are not portable.   Custom      builtin
func-
            tions are disabled by default for this reason.


       -d      Disable  the  printing  of  the  opening title.  The printing of
            resource file debug and informational messages is also  disabled
            as if config("resource_debug", 0) had been executed.

            For example:

               calc "read qtime; qtime(2)"

            will output something like:
```

```
   qtime(utc_hr_offset) defined
   It's nearly ten past six.
```

whereas:

```
   calc -d "read qtime; qtime(2)"
```

will just say:

```
   It's nearly ten past six.
```

This flag disables the reporting of missing calc startup resource files.

-D calc_debug[:resource_debug[:user_debug]]
Force the initial value of config("calc_debug"), config("resource_debug") and config("user_debug").

The : separated strings are interpreted as signed 32 bit integers. After an optional leading sign a leading zero indicates octal conversion, and a leading ''0x'' or ''0X'' hexadecimal conversion. Otherwise, decimal conversion is assumed.

By default, calc_debug is 0, resource_debug is 3 and user_debug is 0.

For more information use the following calc command:

```
   help config
```

-e Ignore any environment variables on startup. The getenv() builtin will still return values, however.

-f This flag is required when using calc in shell script mode. It must be at the end of the initial #! line of the script.

This flag is normally only at the end of a calc shell script. If the first line of an executable file begins #! followed by the absolute pathname of the calc program and the flag -f as in:

```
   #!/usr/bin/calc [other_flags ...] -f
```

the rest of the file will be processed in shell script mode. See SHELL SCRIPT MODE section of this man page below for details.

The actual form of this flag is:

```
   -f filename
```

On systems that treat an executable that begins with #! as a script, the path of the executable is appended by the kernel as the final argument to the exec() system call. This is why the

-f flag at the very end of the #! line.

It is possible use -f filename on the command line:

```
calc [other_flags ...] -f filename
```

This will cause calc to process lines in filename in shell script mode.

Use of -f implies -s. In addition, -d and -p are implied if -i is not given.

-h Print a help message. This option implies -q. This is equivalent to the calc command help help. The help facility is disabled unless the mode is 5 or 7. See -m.

-i Become interactive if possible. This flag will cause calc to drop into interactive mode after the calc_cmd arguments on the command line are evaluated. Without this flag, calc will exit after they are evaluated.

For example:

```
calc 2+5
```

will print the value 7 and exit whereas:

```
calc -i 2+5
```

will print the value 7 and prompt the user for more calc commands.

-m mode

This flag sets the permission mode of calc. It controls the ability for calc to open files and execute programs. Mode may be a number from 0 to 7.

The mode value is interpreted in a way similar to that of the chmod(1) octal mode:

```
0  do not open any file, do not execute progs
1  do not open any file
2  do not open files for reading, do not execute progs
3  do not open files for reading
4  do not open files for writing, do not execute progs
5  do not open files for writing
6  do not execute any program
7  allow everything (default mode)
```

If one wished to run calc from a privileged user, one might want to use -m 0 in an effort to make calc somewhat more secure.

Mode bits for reading and writing apply only on an open. Files already open are not effected. Thus if one wanted to use the -m 0 in an effort to make calc somewhat more secure, but still wanted to read and write a specific file, one might want to do in sh(1), ksh(1), bash(1)-like shells:

```
calc -m 0 3<a.file
```

Files presented to calc in this way are opened in an unknown mode. Calc will attempt to read or write them if directed.

If the mode disables opening of files for reading, then the startup resource files are disabled as if -q was given. The reading of key bindings is also disabled when the mode disables opening of files for reading.

-O Use the old classic defaults instead of the default configuration. This flag as the same effect as executing config("all", "oldcfg") at startup time.

NOTE: Older versions of calc used -n to setup a modified form of the default calc configuration. The -n flag currently does nothing. Use of the -n flag is now deprecated and may be used for something else in the future.

-p Pipe processing is enabled by use of -p. For example:

```
calc -p "2^21701-1" | fizzbin
```

In pipe mode, calc does not prompt, does not print leading tabs and does not print the initial header. The -p flag overrides -i.

-q Disable the reading of the startup scripts.

-s By default, all calc_cmd args are evaluated and executed. This flag will disable their evaluation and instead make them available as strings for the argv() builtin function.

-u Disable buffering of stdin and stdout.

-v Print the calc version number and exit.

-- The double dash indicates to calc that no more option follow. Thus calc will ignore a later argument on the command line even if it starts with a dash. This is useful when entering negative values on the command line as in:

```
calc -p -- -1 - -7
```

With no calc_cmd arguments, calc operates interactively. If one or more arguments are given on the command line and -s is NOT given, then calc will read and execute them and either attempt to go interactive according as the -i flag was present or absent.

If -s is given, calc will not evaluate any calc_cmd arguments but instead make them available as strings to the argv() builtin function.

Sufficiently simple commands with no no characters like parentheses, brackets, semicolons, '*', which have special interpretations in UNIX shells may be entered, possibly with spaces, until the terminating new-line. For example:

```
calc 23 + 47
```

will print 70. However, command lines will have problems:

```
calc 23 * 47

calc -23 + 47
```

The first example above fails because the shell interprets the '*' as a file glob. The second example fails because '-23' is viewed as a calc option (which it is not) and do calc objects to that it thinks of as an unknown option. These cases can usually be made to work as expected by enclosing the command between quotes:

```
calc '23 * 47'

calc "print sqrt(2), exp(1)"
```

or in parentheses and quotes to avoid leading -'s as in:

```
calc '(-23 + 47)'
```

One may also use a double dash to denote that calc options have ended as in:

```
calc -- -23 + 47

calc -q -- -23 + 47
```

If '!' is to be used to indicate the factorial function, for shells like csh(1) for which '!' followed by a non-space character is used for history substitution, it may be necessary to include a space or use a backslash to escape the special meaning of '!'. For example, the command:

```
print 27!^2
```

may have to be replaced by:

```
print 27! ^2   or   print 27^2
```

CALC STARTUP FILES

Normally on startup, if the environment variable $CALCRC is undefined and calc is invoked without the -q flag, or if $CALCRC is defined and calc is invoked with -e, calc looks for a file "startup" in the calc resource directory .calcrc in the user's home directory, and .calcinit in the current directory. If one or more of these are found, they are read in succession as calc scripts and their commands executed. When defined, $CALCRC is to contain a ':' separated list of names of files, and if calc is then invoked without either the -q or -e flags, these files are read in succession and their commands executed. No error condition is produced if a listed file is not found.

If the mode specified by -m disables opening of files for reading, then the reading of startup files is also disabled as if -q was given.

CALC FILE SEARCH PATH

If the environment variable $CALCPATH is undefined, or if it is defined and calc is invoked with the -e flag, when a file name not beginning with /, ~ or ./, is specified as in:

```
calc read myfile
```

calc searches in succession:

```
/usr/lib/myfile
/usr/lib/myfile.cal
/usr/share/calc/custom/myfile
/usr/share/calc/custom/myfile.cal
```

If the file is found, the search stops and the commands in the file are executed. It is an error if no readable file with the specified name is found. An alternative search path can be specified by defining $CALCPATH in the same way as PATH is defined, as a ':' separated list of directories, and then invoking calc without the -e flag.

Calc treats all open files, other than stdin, stdout and stderr as files available for reading and writing. One may present calc with an already open file using sh(1), ksh(1), bash(1)-like shells is to:

```
calc 3<open_file 4<open_file2
```

For more information use the following calc commands:

```
help help
help overview
help usage
help environment
help config
```

SHELL SCRIPT MODE

If the first line of an executable file begins #! followed by the absolute pathname of the calc program and the flag -f as in:

```
   #!/usr/bin/calc [other_flags ...] -f
```

the rest of the file will be processed in shell script mode. Note that -f must at the end of the initial ''#!'' line. Any other optional other_flags must come before the -f.

In shell script mode the contents of the file are read and executed as if they were in a file being processed by a read command, except that a "command" beginning with '#' followed by whitespace and ending at the next newline is treated as a comment. Any optional other_flags will be parsed first followed by the later lines within the script itself.

In shell script mode, -s is always assumed. In addition, -d and -p are automatically set if -i is not given.

For example, if the file /tmp/mersenne:

```
   #!/usr/bin/calc -q -f
   #
   # mersenne - an example of a calc shell script file

   /* parse args */
   if (argv() != 1) {
     fprintf(files(2), "usage: %s exp\n", config("program"));
     abort "must give one exponent arg";
   }

   /* print the mersenne number */
   print "2^": argv(0) : "-1 =", 2^eval(argv(0))-1;
```

is made an executable file by:

```
   chmod +x /tmp/mersenne
```

then the command line:

```
   /tmp/mersenne 127
```

will print:

```
   2^127-1 = 170141183460469231731687303715884105727
```

Note that because -s is assumed in shell script mode and non-dashed args are made available as strings via the argv() builtin function. Therefore:

```
   2^eval(argv(0))-1
```

will print the decimal value of 2^n-1 but

```
   2^argv(0)-1
```

will not.

DATA TYPES

Fundamental builtin data types include integers, real numbers, rational numbers, complex numbers and strings.

By use of an object, one may define an arbitrarily complex data types. One may define how such objects behave a wide range of operations such as addition, subtraction, multiplication, division, negation, squaring, modulus, rounding, exponentiation, equality, comparison, printing and so on.

For more information use the following calc commands:

```
help types
help obj
show objfuncs
```

VARIABLES

Variables in calc are typeless. In other words, the fundamental type of a variable is determined by its content. Before a variable is assigned a value it has the value of zero.

The scope of a variable may be global, local to a file, or local to a procedure. Values may be grouped together in a matrix, or into a a list that permits stack and queue style operations.

For more information use the following calc commands:

```
help variable
help mat
help list
show globals
```

INPUT/OUTPUT

A leading ''0x'' implies a hexadecimal value, a leading ''0b'' implies a binary value, and a ''0'' followed by a digit implies an octal value. Complex numbers are indicated by a trailing ''i'' such as in ''3+4i''. Strings may be delimited by either a pair of single or double quotes. By default, calc prints values as if they were floating point numbers. One may change the default to print values in a number of modes including fractions, integers and exponentials.

A number of stdio-like file I/O operations are provided. One may open, read, write, seek and close files. Filenames are subject to '' '' expansion to home directories in a way similar to that of the Korn or C-Shell.

For example:

```
~/.calcrc
~chongo/lib/fft_multiply.cal
```

For more information use the following calc command:

```
help file
```

CALC LANGUAGE

The calc language is a C-like language. The language includes commands such as variable declarations, expressions, tests, labels, loops, file operations, function calls. These commands are very similar to their counterparts in C.

The language also include a number of commands particular to calc itself. These include commands such as function definition, help, reading in resource files, dump files to a file, error notification, configuration control and status.

For more information use the following calc command:

```
help command
help statement
help expression
help operator
help config
```

FILES

/usr/bin/calc
calc binary

/usr/bin/cscript/*
calc shell scripts

/usr/lib/*.cal
calc standard resource files

/usr/lib/help/*
help files

/usr/lib/bindings
non-GNU-readline command line editor bindings

/usr/include/calc/*.h
include files for C interface use

/usr/lib/libcalc.a
calc binary link library

/usr/lib/libcustcalc.a
custom binary link library

/usr/share/calc/custom/*.cal
custom resource files

/usr/share/calc/custhelp/*
custom help files

ENVIRONMENT

CALCPATH

A :-separated list of directories used to search for calc resource filenames that do not begin with /, ./ or ~.

Default value: .:./cal:~/.cal:/usr/share/calc:/usr/share/calc/custom

CALCRC

On startup (unless -h or -q was given on the command line), calc searches for files along this :-separated environment variable.

Default value: /usr/share/calc/startup:~/.calcrc:./.calcinit

CALCBINDINGS

On startup (unless -h or -q was given on the command line, or -m disallows opening files for reading), calc reads key bindings from the filename specified by this environment variable. The key binding file is searched for along the $CALCPATH list of directories.

Default value: binding

This variable is not used if calc was compiled with GNU-readline support. In that case, the standard readline mechanisms (see readline(3)) are used.

Credits

The main chunk of calc was written by David I. Bell.

The calc primary mirror, calc mailing list and calc bug report processing is performed by Landon Curt Noll.

Landon Curt Noll maintains the master reference source, performs release control functions as well as other calc maintenance functions.

Thanks for suggestions and encouragement from Peter Miller, Neil Justusson, and Landon Noll.

Thanks to Stephen Rothwell for writing the original version of hist.c which is used to do the command line editing.

Thanks to Ernest W. Bowen for supplying many improvements in accuracy and generality for some numeric functions. Much of this was in terms of actual code which I gratefully accepted. Ernest also supplied the original text for many of the help files.

Portions of this program are derived from an earlier set of public domain arbitrarily precision routines which was posted to the net around 1984. By now, there is almost no recognizable code left from that original source.

COPYING / CALC GNU LESSER GENERAL PUBLIC LICENSE

Calc is open software, and is covered under version 2.1 of the GNU Lesser General Public License. You are welcome to change it and/or distribute copies of it under certain conditions. The calc commands:

```
help copyright
help copying
help copying-lgpl
```

should display the contents of the COPYING and COPYING-LGPL files. Those files contain information about the calc's GNU Lesser General Public License, and in particular the conditions under which you are allowed to change it and/or distribute copies of it.

You should have received a copy of the version 2.1 of the GNU Lesser General Public License. If you do not have these files, write to:

Free Software Foundation, Inc.
51 Franklin Street
Fifth Floor
Boston, MA 02110-1301
USA

Calc is copyrighted in several different ways. These ways include:

Copyright (C) year David I. Bell
Copyright (C) year David I. Bell and Landon Curt Noll
Copyright (C) year David I. Bell and Ernest Bowen
Copyright (C) year David I. Bell, Landon Curt Noll and Ernest Bowen

CALC MAILING LIST / CALC UPDATES / ENHANCEMENTS

To contribute comments, suggestions, enhancements and interesting calc resource files, and shell scripts please join the low volume calc mailing list.

To join the low volume calc mailing list, send EMail to:

calc-tester-request at asthe dot com

Your subject must contain the words:

calc mailing list subscription

You may have additional words in your subject line.

Your message body must contain:

```
subscribe calc-tester address
end
name your_full_name
```

where address s your EMail address and your_full_name is your full name. Feel free to follow the name line with additional EMail text as desired.

BUG REPORTS / BUG FIXES

Send bug reports and bug fixes to:

calc-bugs at asthe dot com

[[NOTE: Replace 'at' with @, 'dot' is with . and remove the spaces]]
[[NOTE: The EMail address uses 'asthe' and the web site URL uses 'isthe']]

Your subject must contain the words:

calc bug report

You may have additional words in your subject line.

See the BUGS source file or use the calc command:

help bugs

for more information about bug reporting.

```
CALC WEB SITE

       Landon Noll maintains the the calc web site is located at:

          www.isthe.com/chongo/tech/comp/calc/

       Share and Enjoy! :-)
```

Calc to do items:

```
    The following items should be addressed sometime in the short to
    medium term future, if not before the next release.

    Code contributions are welcome.  Send patches to:

          calc-contrib at asthe dot com

    Calc bug reports, however, should send to:

          calc-bugs at asthe dot com

    [[ NOTE: Replace 'at' with @, 'dot' is with . and remove the spaces ]]
    [[ NOTE: The EMail address uses 'asthe' and the web site URL uses 'isthe' ]]

    See the BUGS file or try the calc command:

          help bugs

    See also the 'wishlist' help files for the calc enhancement wish list.

=-=

Very High priority items:

    * Improve the way that calc parses statements such as if, for, while
      and do so that when a C programmer does.  This works as expected:

          if (expr) {
            ...
          }

      However this WILL NOT WORK AS EXPECTED:

          if (expr)
          {
            ...
          }

      because calc will parse the if being terminated by
      an empty statement followed by a

          if (expr) ;
          {
            ...
          }

      See also "help statement", "help unexpected", "help todo", and
      "help bugs".

    * Consider using GNU autoconf / configure to build calc.

    * It is overkill to have nearly everything wind up in libcalc.
      Form a libcalcmath and a libcalclang so that an application
      that just wants to link with the calc math libs can use them
      without dragging in all of the other calc language, I/O,
```

```
        and builtin functions.

      * Fix any 'Known bugs' as noted in the BUGS file or as
        displayed by 'calc help bugs'.

=-=

High priority items:

      * Verify, complete or fix the 'SEE ALSO' help file sections.

      * Verify, complete or fix the 'LINK LIBRARY' help file sections.

      * Verify, complete or fix the 'LIMITS' help file sections.

      * Verify, complete or fix the 'SYNOPSIS' and 'TYPES' help file sections.

      * Perform a code coverage analysis of the 'make check' action
        and improve the coverage (within reason) of the regress.cal suite.

      * Address, if possible and reasonable, any Calc Mis-features
        as noted in the BUGS file or as displayed by 'calc help bugs'.

      * Internationalize calc by converting calc error messages and
        text strings (e.g., calc startup banner, show output, etc.)
        into calls to the GNU gettext internationalization facility.
        If somebody translated these strings into another language,
        setting $LANG would allow calc to produce error messages
        and text strings in that language.

=-=

Medium priority items:

      * Complete the use of CONST where appropriate:

        CONST is beginning to be used with read-only tables and some
        function arguments.  This allows certain compilers to better
        optimize the code as well as alerts one to when some value
        is being changed inappropriately.  Use of CONST as in:

            int foo(CONST int curds, char *CONST whey)

        while legal C is not as useful because the caller is protected
        by the fact that args are passed by value.  However, the
        in the following:

            int bar(CONST char *fizbin, CONST HALF *data)

        is useful because it calls the compiler that the string pointed
        at by 'fizbin' and the HALF array pointer at by 'data' should be
        treated as read-only.

        One should make available a the fundamental math operations
        on ZVALUE, NUMBER and perhaps COMPLEX (without all of the
        other stuff) in a separate library.

      * Clean the source code and document it better.
```

```
* Add a builtin function to access the 64 bit FNV hash which
  is currently being used internally in seed.c.
```

Calc Enhancement Wish List:

```
    Send calc comments, suggestions, bug fixes, enhancements and
    interesting calc scripts that you would like you see included in
    future distributions to:

          calc-contrib at asthe dot com

    The following items are in the calc wish list.  Programs like this
    can be extended and improved forever.

    Calc bug reports, however, should be sent to:

          calc-bugs at asthe dot com

    [[ NOTE: Replace 'at' with @, 'dot' is with . and remove the spaces ]]
    [[ NOTE: The EMail address uses 'asthe' and the web site URL uses 'isthe' ]]

    See the 'todo' help file for higher priority todo items.

=-=

    *  In general use faster algorithms for large numbers when they
       become known.  In particular, look at better algorithms for
       very large numbers -- multiply, square and mod in particular.

    *  Implement an autoload feature.  Associate a calc resource filename
       with a function or global variable.  On the first reference of
       such item, perform an automatic load of that file.

    *  Add error handling statements, so that QUITs, errors from the
       'eval' function, division by zeroes, and so on can be caught.
       This should be done using syntax similar to:

            ONERROR statement DO statement;

       Something like signal isn't versatile enough.

    *  Add a debugging capability so that functions can be single stepped,
       breakpoints inserted, variables displayed, and so on.

    *  Figure out how to write all variables out to a file, including
       deeply nested arrays, lists, and objects.

       Add the ability to read and write a value in some binary form.
       Clearly this is easy for non-neg integers.  The question of
       everything else is worth pondering.

    *  Eliminate the need for the define keyword by doing smarter parsing.

    *  Allow results of a command (or all commands) to be re-directed to a
       file or piped into a command.

    *  Add some kind of #include and #define facility.      Perhaps use
       the C pre-processor itself?

    *  Support a more general input and output base mode other than
```

```
   just dec, hex or octal.

*  Implement a form of symbolic algebra.  Work on this has already
   begun.  This will use backquotes to define expressions, and new
   functions will be able to act on expressions.  For example:

      x = `hello * strlen(mom)`;
      x = sub(x, `hello`, `hello + 1`);
      x = sub(x, `hello`, 10, `mom`, "curds");
      eval(x);

   prints 55.

*  Place the results of previous commands into a parallel history list.
   Add a binding that returns the saved result of the command so
   that one does not need to re-execute a previous command simply
   to obtain its value.

   If you have a command that takes a very long time to execute,
   it would be nice if you could get at its result without having
   to spend the time to reexecute it.

*  Add a binding to delete a value from the history list.

   One may need to remove a large value from the history list if
   it is very large.  Deleting the value would replace the history
   entry with a null value.

*  Add a binding to delete a command from the history list.

   Since you can delete values, you might as well be able to
   delete commands.

*  All one to alter the size of the history list thru config().

   In some cases, 256 values is too small, in others it is too large.

*  Add a builtin that returns a value from the history list.
   As an example:

      histval(-10)

   returns the 10th value on the history value list, if such
   a value is in the history list (null otherwise).      And:

      histval(23)

   return the value of the 23rd command given to calc, if
   such a value is in the history list (null otherwise).

   It would be very helpful to use the history values in
   subsequent equations.

*  Add a builtin that returns command as a string from the
   history list.  As an example:

      history(-10)

   returns a string containing the 10th command on the
```

history list, if a such a value is in the history list (empty string otherwise). And:

```
history(23)
```

return the string containing the 23rd command given to calc, if such a value is in the history list (empty string otherwise).

One could use the eval() function to re-evaluate the command.

* Allow one to optionally restore the command number to calc prompts. When going back in the history list, indicate the command number that is being examined.

  The command number was a useful item. When one is scanning the history list, knowing where you are is hard without it. It can get confusing when the history list wraps or when you use search bindings. Command numbers would be useful in conjunction with positive args for the history() and histval() functions as suggested above.

* Add a builtin that returns the current command number. For example:

  ```
  cmdnum()
  ```

  returns the current command number.

  This would allow one to tag a value in the history list. One could save the result of cmdnum() in a variable and later use it as an arg to the histval() or history() functions.

* Add a factoring builtin functions. Provide functions that perform multiple polynomial quadratic sieves, elliptic curve, difference of two squares, N-1 factoring as so on. Provide a easy general factoring builtin (say factor(foo)) that would attempt to apply whatever process was needed based on the value.

  Factoring builtins would return a matrix of factors.

  It would be handy to configure, via config(), the maximum time that one should try to factor a number. By default the time should be infinite. If one set the time limit to a finite value and the time limit was exceeded, the factoring builtin would return whatever if had found thus far, even if no new factors had been found.

  Another factoring configuration interface, via config(), that is needed would be to direct the factoring builtins to return as soon as a factor was found.

* Allow one to config calc break up long output lines.

  The command: calc '2^100000' will produce one very long line. Many times this is reasonable. Long output lines are a problem for some utilities. It would be nice if one could configure, via config(), calc to fold long lines.

  By default, calc should continue to produce long lines.

```
   One option to config should be to specify the length to
   fold output.  Another option should be to append a trailing
   \ on folded lines (as some symbolic packages use).

*  Allow one to use the READ and WRITE commands inside a function.

*  Remove or increase limits on factor(), lfactor(), isprime(),
   nextprime(), and prevprime().  Currently these functions cannot
   search for factors > 2^32.

*  Add read -once -try "filename" which would do nothing
   if "filename" was not a readable file.
```