100 doors

There are 100 doors in a row that are all initially closed.

You make 100 passes by the doors.

The first time through, visit every door and *toggle* the door (if the door is closed, open it; if it is open, close it).

The second time, only visit every 2nd door (door #2, #4, #6, ...), and toggle it.

The third time, visit every 3rd door (door #3, #6, #9, ...), etc, until you only visit the 100th door.

Task

Answer the question: what state are the doors in after the last pass? Which are open, which are closed?

Alternate: As noted in this page's discussion page, the only doors that remain open are those whose numbers are perfect squares.

Opening only those doors is an optimization that may also be expressed; however, as should be obvious, this defeats the intent of comparing implementations across programming languages.

Unoptimized

```
doors: array/initial 100 'closed
repeat i 100 [
    door: at doors i
    forskip door i [change door either 'open = first door ['closed] ['open]]
]
```

Optimized

```
doors: array/initial 100 'closed
repeat i 10 [doors/(i * i): 'open]
```

Red

Unoptimized

```
doors: make vector! [char! 8 100]
repeat i 100 [change at doors i #"."]

repeat i 100 [
    j: i
    while [j <= 100] [
      door: at doors j
      change door either #"O" = first door [#"."] [#"O"]
      j: j + i
    ]
]

repeat i 10 [
  print copy/part at doors (i - 1 * 10 + 1) 10
]
```

99 Bottles of Beer

Display the complete lyrics for the song: **99 Bottles of Beer on the Wall.**

The lyrics follow this form:

```
    99 bottles of beer on the wall
    99 bottles of beer
    Take one down, pass it around
    98 bottles of beer on the wall

    98 bottles of beer on the wall
    98 bottles of beer
    Take one down, pass it around
    97 bottles of beer on the wall
```

... and so on, until reaching 0.

Grammatical support for "1 bottle of beer" is optional.

As with any puzzle, try to do it in as creative/concise/comical a way as possible (simple, obvious solutions allowed, too).

```
; The 'bottles' function maintains correct grammar.
```

```
bottles: func [n /local b][
        b: either 1 = n ["bottle"]["bottles"]
        if 0 = n [n: "no"]
        reform [n b]
]

for n 99 1 -1 [print [
        bottles n  "of beer on the wall" crlf
        bottles n  "of beer" crlf
        "Take one down, pass it around" crlf
        bottles n - 1  "of beer on the wall" crlf
]]
```

Output *(selected highlights)*:

```
99 bottles of beer on the wall    2 bottles of beer on the wall
99 bottles of beer                2 bottles of beer
Take one down, pass it around     Take one down, pass it around
98 bottles of beer on the wall    1 bottle of beer on the wall

...Continues...                   1 bottle of beer on the wall
                                  1 bottle of beer
                                  Take one down, pass it around
                                  no bottles of beer on the wall
```

This one prints with proper grammar. "Bottles" changed to "bottle" at the end of the 2 line, and throughout the 1 line. 0 changed to "No" in the last line:

```
for i 99 1 -1 [
     x: rejoin [
         i b: " bottles of beer" o: " on the wall. " i b
         ". Take one down, pass it around. " (i - 1) b o "^/"
     ]
     r: :replace j: "bottles" k: "bottle"
     switch i [1 [r x j k r at x 10 j k r x "0" "No"] 2 [r at x 40 j k]]
     print x
] halt
```

Here's a simple 1 line console version:

```
for i 99 1 -1[print rejoin[i b:" bottles of beer"o:" on the wall. "i b". Take one down, pass it around. "(i - 1)b o"^/"]]
```

Red

```
; The 'bottles' function maintains correct grammar.
```

```
bottles: function [n] [
  b: either 1 = n ["bottle"]["bottles"]
  if 0 = n [n: "no"]
  form reduce [n b]
]

repeat x 99 [
  n: 100 - x
  print [
    bottles n "of beer on the wall"     crlf
    bottles n "of beer"                 crlf
    "Take one down, pass it around"     crlf
    bottles n - 1 "of beer on the wall" crlf
  ]]
```

A+B

A+B —— a classic problem in programming contests, it's given so contestants can gain familiarity with the online judging system being used.

Given two integers, **A** and **B**.

Their sum needs to be calculated.

Input data

Two integers are written in the input stream, separated by space(s):

$$(-1000 \le A , B \le +1000)$$

Output data

The required output is one integer: the sum of **A** and **B**.

```
forever [x: load input  print x/1 + x/2]
```

Output:

```
1 2
3
2 2
4
3 2
5
```

Red

```
x: load input  print x/1 + x/2
```

Output:

```
1 2
3
2 2
4
3 2
5
```

Alternative implementations:

```
print (first x: load input) + x/2
```

```
print head insert load input 'add
```

```
print load replace input " " " + "
```

Abstract type

Abstract type is a type without instances or without definition.

For example in object-oriented programming using some languages, abstract types can be partial implementations of other types, which are to be derived there-from. An abstract type may provide implementation of some operations and/or components. Abstract types without any implementation are called **interfaces**. In the languages that do not support multiple inheritance (Ada, Java), classes can, nonetheless, inherit from multiple interfaces. The languages with multiple inheritance (like C++) usually make no distinction between partially implementable abstract types and interfaces. Because the abstract type's implementation is incomplete, OO languages normally prevent instantiation from them (instantiation must derived from one of their descendant classes).

The term **abstract datatype** also may denote a type, with an implementation provided by the programmer rather than directly by the language (a **built-in** or an inferred type). Here the word

abstract means that the implementation is abstracted away, irrelevant for the user of the type. Such implementation can and should be hidden if the language supports separation of implementation and specification. This hides complexity while allowing the implementation to change without repercussions on the usage. The corresponding software design practice is said to follow the information hiding principle.

It is important not to confuse this *abstractness* (of implementation) with one of the **abstract type**. The latter is abstract in the sense that the set of its values is empty. In the sense of implementation abstracted away, all user-defined types are abstract.

In some languages, like for example in Objective Caml which is strongly statically typed, it is also possible to have **abstract types** that are not OO related and are not an abstractness too. These are *pure abstract types* without any definition even in the implementation and can be used for example for the type algebra, or for some consistence of the type inference. For example in this area, an abstract type can be used as a phantom type to augment another type as its parameter.

Task: show how an abstract type can be declared in the language. If the language makes a distinction between interfaces and partially implemented types illustrate both.

```
; The "shape" class is an abstract class -- it defines the "pen"
; property and "line" method, but "size" and "draw" are undefined and
; unimplemented.

shape: make object! [
        pen:  "X"
        size: none

        line: func [count][loop count [prin self/pen]  prin crlf]
        draw: does [none]
]

; The "box" class inherits from "shape" and provides the missing
; information for drawing boxes.
```

```
box: make shape [
        size: 10
        draw: does [loop self/size [line self/size]]
]

; "rectangle" also inherits from "shape", but handles the
; implementation very differently.

rectangle: make shape [
        size: 20x10
        draw: does [loop self/size/y [line self/size/x]]
]

; Unlike some languages discussed, REBOL has absolutely no qualms
; about instantiating an "abstract" class -- that's how I created the
; derived classes of "rectangle" and "box", after all.

s: make shape []  s/draw ; Nothing happens.

print "A box:"
b: make box [pen: "O" size: 5]  b/draw

print [crlf "A rectangle:"]
r: make rectangle [size: 32x5]  r/draw
```

Red

```
; The "shape" class is an abstract class -- it defines the "pen"
; property and "line" method, but "size" and "draw" are undefined and
; unimplemented.

shape: make object! [
  pen:  "X"
  size: none

  line: func [count][loop count [prin self/pen] prin newline]
  draw: does [none]
]

; The "box" class inherits from "shape" and provides the missing
; information for drawing boxes.

box: make shape [
  size: 10
  draw: does [loop self/size [line self/size]]
]

; "rectangle" also inherits from "shape", but handles the
; implementation very differently.
```

```
rectangle: make shape [
  size: 20x10
  draw: does [loop self/size/y [line self/size/x]]
]

; Unlike some languages discussed, REBOL has absolutely no qualms
; about instantiating an "abstract" class -- that's how I created the
; derived classes of "rectangle" and "box", after all.

print "An abstract shape (nothing):"
s: make shape []                s/draw ; Nothing happens.

print [newline "A box:"]
b: make box [pen: "O" size: 5]  b/draw

print [newline "A rectangle:"]
r: make rectangle [size: 32x5]  r/draw
```

Accumulator factory

A problem posed by Paul Graham is that of creating a function that takes a single (numeric) argument and which returns another function that is an accumulator. The returned accumulator function in turn also takes a single numeric argument, and returns the sum of all the numeric values passed in so far to that accumulator (including the initial value passed when the accumulator was created).

Rules

The detailed rules are at http://paulgraham.com/accgensub.html and are reproduced here for simplicity (with additions in *small italic text*).

> Before you submit an example, make sure the function
>
> 1. Takes a number n and returns a function (lets call it g), that takes a number i, and returns n incremented by the accumulation of i from every call of function g(i). Although these exact function and parameter names need not be used
> 2. Works for any numeric type-- i.e. can take both ints and floats and returns functions that can take both ints and floats. (It is not enough simply to convert all input to floats. An accumulator that has only seen

integers must return integers.) *(i.e., if the language doesn't allow for numeric polymorphism, you have to use overloading or something like that)*

3. Generates functions that return the sum of every number ever passed to them, not just the most recent. *(This requires a piece of state to hold the accumulated value, which in turn means that pure functional languages can't be used for this task.)*
4. Returns a real function, meaning something that you can use wherever you could use a function you had defined in the ordinary way in the text of your program. *(Follow your language's conventions here.)*
5. Doesn't store the accumulated value or the returned functions in a way that could cause them to be inadvertently modified by other code. *(No global variables or other such things.)*

E.g. if after the example, you added the following code (in a made-up language) *where the factory function is called foo*:

```
x = foo(1);
x(5);
foo(3);
print x(2.3);
```

It should print 8.3. *(There is no need to print the form of the accumulator function returned by foo(3); it's not part of the task at all.)*

Create a function that implements the described rules.

It need not handle any special error cases not described above. The simplest way to implement the task as described is typically to use a closure, providing the language supports them.

Where it is not possible to hold exactly to the constraints above, describe the deviations.

```
make-acc-gen: func [start-val] [
    use [state] [
        state: start-val
        func [value] [
            state: state + value
        ]
    ]
]
```

Output:

```
>> x: make-acc-gen 1
>> x 5
== 6
>> make-acc-gen 3
>> print x 2.3
8.3
```

Ackermann function

The **Ackermann function** is a classic example of a recursive function, notable especially because it is not a primitive recursive function. It grows very quickly in value, as does the size of its call tree.

The Ackermann function is usually defined as follows:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Its arguments are never negative and it always terminates. Write a function which returns the value of A (m , n). Arbitrary precision is preferred (since the function grows so quickly), but not required.

See also

- Conway chained arrow notation for the Ackermann function.

```
ackermann: func [m n] [
    case [
        m = 0 [n + 1]
        n = 0 [ackermann m - 1 1]
        true [ackermann m - 1 ackermann m n - 1]
    ]
]
```

Add a variable to a class instance at runtime

Demonstrate how to dynamically add variables to an object (a class instance) at runtime.

This is useful when the methods/variables of an instance are based on a data file that isn't available until runtime. Hal Fulton gives an example of creating an OO CSV parser at An Exercise in Metaprogramming with Ruby. This is referred to as "monkeypatching" by Pythonistas and some others.

```
; As I understand it, a REBOL object can only ever have whatever
; properties it was born with. However, this is somewhat offset by the
; fact that every instance can serve as a prototype for a new object
; that also has the new parameter you want to add.

; Here I create an empty instance of the base object (x), then add the
; new instance variable while creating a new object prototyped from
; x. I assign the new object to x, et voila', a dynamically added
; variable.

x: make object! [] ; Empty object.

x: make x [ newvar: "forty-two" ; New property. ]

print "Empty object modifed with 'newvar' property:"
probe x

; A slightly more interesting example:

starfighter: make object! [
        model: "unknown"
        pilot: none
]
x-wing: make starfighter [
        model: "Incom T-65 X-wing"
]

squadron: reduce [
        make x-wing [pilot: "Luke Skywalker"]
        make x-wing [pilot: "Wedge Antilles"]
        make starfighter [
                model: "Slayn & Korpil B-wing"
                pilot: "General Salm"
        ]
]
```

```
; Adding new property here.
squadron/1: make squadron/1 [deathstar-removal-expert: yes]

print [crlf "Fighter squadron:"]
foreach pilot squadron [probe pilot]
```

Red

```
person: make object! [
  name: none
  age:  none
]

people: reduce [make person [name: "fred" age: 20] make person [name: "paul"
age: 21]]
people/1: make people/1 [skill: "fishing"]

foreach person people [
  print reduce [person/age "year old" person/name "is good at" any [select
person 'skill "nothing"]]
]
```

Align columns

Given a text file of many lines, where fields within a line are delineated by a single 'dollar' character, write a program that aligns each column of fields by ensuring that words in each column are separated by at least one space. Further, allow for each word in a column to be either left justified, right justified, or center justified within its column.

Use the following text to test your programs:

```
Given$a$text$file$of$many$lines,$where$fields$within$a$line$
are$delineated$by$a$single$'dollar'$character,$write$a$program
that$aligns$each$column$of$fields$by$ensuring$that$words$in$each$
column$are$separated$by$at$least$one$space.
Further,$allow$for$each$word$in$a$column$to$be$either$left$
justified,$right$justified,$or$center$justified$within$its$column.
```

Note that:

1. The example input texts lines may, or may not, have trailing dollar characters.
2. All columns should share the same alignment.
3. Consecutive space characters produced adjacent to the end of lines are insignificant for the purposes of the task.
4. Output text will be viewed in a mono-spaced font on a plain text editor or basic terminal.
5. The minimum space between columns should be computed from the text and not hard-coded.
6. It is *not* a requirement to add separating characters between or around columns.

```
specimen: {Given$a$text$file$of$many$lines,$where$fields$within$a$line$
are$delineated$by$a$single$'dollar'$character,$write$a$program
that$aligns$each$column$of$fields$by$ensuring$that$words$in$each$
column$are$separated$by$at$least$one$space.
Further,$allow$for$each$word$in$a$column$to$be$either$left$
justified,$right$justified,$or$center$justified$within$its$column.}

; Parse specimen into data grid.

data: copy []
foreach line parse specimen to-string lf [ ; Break into lines.
        append/only data parse line "$"        ; Break into columns.
]

; Compute independent widths for each column.

widths: copy []  insert/dup widths 0 length? data/1
foreach line data [
        forall line [
                i: index? line
                widths/:i: max widths/:i length? Line/1 ]
]

pad: func [n /local x][x: copy ""  insert/dup x " " n  x]

; These formatting functions are passed as arguments to entable.

right: func [n s][rejoin [pad n - length? s  s]]

left: func [n s][rejoin [s  pad n - length? s]]

centre: func [n s /local h][
        h: round/down (n - length? s) / 2
        rejoin [pad h  s  pad n - h - length? s]
]
```

```
; Display data as table.

entable: func [data format] [
        foreach line data [
                forall line [
                        prin rejoin [format  pick widths index? line  line/1
" "]
                ]
                print ""
        ]
]

; Format data table.

foreach i [left centre right] [
        print ["^/Align" i "...^/"]  entable data get i]
```

Output:

```
Align left ...

Given      a          text       file   of     many      lines,     where
fields  within  a      line
are        delineated by         a      single 'dollar'  character, write    a
program
that       aligns     each       column of     fields    by         ensuring
that    words   in     each
column     are        separated  by     at     least     one        space.
Further,   allow      for        each   word   in        a          column
to      be      either left
justified, right      justified, or     center justified within     its
column.

Align centre ...

  Given        a         text     file    of     many      lines,    where
fields  within    a    line
   are     delineated     by       a    single 'dollar'  character,  write
a    program
   that      aligns      each    column   of    fields       by     ensuring
that    words    in   each
  column      are     separated    by     at     least      one      space.
 Further,    allow       for      each   word     in         a       column
to      be    either left
justified,   right    justified,   or   center justified   within     its
column.

Align right ...
```

```
     Given          a       text   file     of      many     lines,    where
fields  within      a line
       are delineated         by      a single  'dollar' character,    write
a program
      that     aligns       each column     of    fields         by ensuring
that   words     in each
    column        are  separated     by     at     least        one   space.
  Further,      allow        for   each   word        in          a   column
to      be either left
justified,      right justified,     or center justified     within      its
column.
```

Red

```
text: {Given$a$text$file$of$many$lines,$where$fields$within$a$line$
are$delineated$by$a$single$'dollar'$character,$write$a$program
that$aligns$each$column$of$fields$by$ensuring$that$words$in$each$
column$are$separated$by$at$least$one$space.
Further,$allow$for$each$word$in$a$column$to$be$either$left$
justified,$right$justified,$or$center$justified$within$its$column.}

; Parse specimen into data grid.

data: copy []
foreach line split text lf [ append/only data split line "$" ]

; Compute independent widths for each column.

widths: copy []
foreach line data [
  forall line [
    i: index? line
    if i > length? widths [append widths 0]
    widths/:i: max widths/:i length? line/1
  ]
]

pad: function [n] [x: copy "" insert/dup x " " n x]

; These formatting functions are passed as arguments to entable.

right: func [n s][rejoin [pad n - length? s s]]

left: func [n s][rejoin [s pad n - length? s]]

centre: function [n s] [
  d: n - length? s
  h: round/down d / 2
  rejoin [pad h s pad d - h]
]
```

```
; Display data as table.

entable: func [data format] [
  foreach line data [
    forall line [
      prin rejoin [format pick widths index? line line/1 " "]
    ]
    print ""
  ]
]

; Format data table.

foreach i [left centre right] [
  print [newline "Align" i "..." newline]  entable data get i]
```

Anonymous recursion

While implementing a recursive function, it often happens that we must resort to a separate *helper function* to handle the actual recursion.

This is usually the case when directly calling the current function would waste too many resources (stack space, execution time), causing unwanted side-effects, and/or the function doesn't have the right arguments and/or return values.

So we end up inventing some silly name like **foo2** or **foo_helper**. I have always found it painful to come up with a proper name, and see some disadvantages:

- You have to think up a name, which then pollutes the namespace
- Function is created which is called from nowhere else
- The program flow in the source code is interrupted

Some languages allow you to embed recursion directly in-place. This might work via a label, a local *gosub* instruction, or some special keyword.

Anonymous recursion can also be accomplished using the Y combinator.

If possible, demonstrate this by writing the recursive version of the fibonacci function (see Fibonacci sequence) which checks for a negative argument before doing the actual recursion.

```
fib: func [n /f][ do f: func [m] [ either m < 2 [m][(f m - 1) + f m - 2]] n]
```

Apply a callback to an array

Take a combined set of elements and apply a function to each element.

```
map: func [
        "Apply a function across an array."
        f [native! function!] "Function to apply to each element of array."
        a [block!] "Array to process."
        /local x
][x: copy []  forall a [append x do [f a/1]]  x]

square: func [x][x * x]

; Tests:

assert: func [code][print [either do code ["  ok"]["FAIL"]  mold code]]

print "Simple loop, modify in place:"
assert [[1 100 81] = (a: [1 10 9]  forall a [a/1: square a/1]  a)]

print [crlf "Functional style with 'map':"]
assert [[4 16 36] = map :square [2 4 6]]

print [crlf "Applying native function with 'map':"]
assert [[2 4 6] = map :square-root [4 16 36]]
```

Output:

```
Simple loop, modify in place:
  ok [[1 100 81] = (a: [1 100 81] forall a [a/1: square a/1] a)]

Functional style with 'map':
  ok [[4 16 36] = map :square [2 4 6]]

Applying native function with 'map':
  ok [[2 4 6] = map :square-root [4 16 36]]
```

Arithmetic/Integer

Get two integers from the user, and then (for those two integers), display their:

- sum
- difference
- product
- integer quotient
- remainder
- exponentiation (if the operator exists)

Don't include error handling.

For quotient, indicate how it rounds (e.g. towards zero, towards negative infinity, etc.).

For remainder, indicate whether its sign matches the sign of the first operand or of the second operand, if they are different.

```
x: to-integer ask "Please type in an integer, and press [enter]: "
y: to-integer ask "Please enter another integer: "
print ""

print ["Sum:"  x + y]
print ["Difference:"  x - y]
print ["Product:" x * y]

print ["Integer quotient (coercion)                          :"  to-integer
x / y]
print ["Integer quotient (away from zero)                    :"  round x / y]
print ["Integer quotient (halves round towards even digits)  :"  round/even
x / y]
print ["Integer quotient (halves round towards zero)         :"  round/half-
down x / y]
print ["Integer quotient (round in negative direction)       :"  round/floor x
/ y]
print ["Integer quotient (round in positive direction)       :"  round/ceiling
x / y]
print ["Integer quotient (halves round in positive direction):"  round/half-
ceiling x / y]

print ["Remainder:"  r: x // y]

; REBOL evaluates infix expressions from left to right. There are no
; precedence rules -- whatever is first gets evaluated. Therefore when
; performing this comparison, I put parens around the first term
```

```
; ("sign? a") of the expression so that the value of /a/ isn't
; compared to the sign of /b/. To make up for it, notice that I don't
; have to use a specific return keyword. The final value in the
; function is returned automatically.

match?: func [a b][(sign? a) = sign? b]

result: copy []
if match? r x [append result "first"]
if match? r y [append result "second"]

; You can evaluate arbitrary expressions in the middle of a print, so
; I use a "switch" to provide a more readable result based on the
; length of the /results/ list.

print [
        "Remainder sign matches:"
        switch length? result [
                0 ["neither"]
                1 [result/1]
                2 ["both"]
        ]
]

print ["Exponentiation:" x ** y]
```

Output:

```
Please type in an integer, and press [enter]: 17
Please enter another integer: -4

Sum: 13
Difference: 21
Product: -68
Integer quotient (coercion)                          : -4
Integer quotient (away from zero)                    : -4
Integer quotient (halves round towards even digits)  : -4
Integer quotient (halves round towards zero)         : -4
Integer quotient (round in negative direction)       : -5
Integer quotient (round in positive direction)       : -4
Integer quotient (halves round in positive direction): -4
Remainder: 1
Remainder sign matches: first
Exponentiation: 1.19730367213036E-5
```

Array concatenation

Show how to concatenate two arrays in your language.
If this is as simple as *array1* + *array2*, so be it.

```
a1: [1 2 3]
a2: [4 5 6]
a3: [7 8 9]

append a1 a2 ; -> [1 2 3 4 5 6]

append/only a1 a3 ; -> [1 2 3 4 5 6 [7 8 9]]
```

Red

```
>> arr1: ["a" "b" "c"]
>> arr2: ["d" "e" "f"]
>> append arr1 arr2
== ["a" "b" "c" "d" "e" "f"]
>> arr3: [1 2 3]
>> insert arr1 arr3
>> arr1
== [1 2 3 "a" "b" "c" "d" "e" "f"]
>> arr4: [22 33 44]
== [22 33 44]
>> append/only arr1 arr4
== [1 2 3 "a" "b" "c" "d" "e" "f" [22 33 44]]
```

Arrays

This task is about arrays.

For hashes or associative arrays, please see Creating an Associative Array.

For a definition and in-depth discussion of what an array is, see Array.

Show basic array syntax in your language.

Basically, create an array, assign a value to it, and retrieve an element (if available, show both fixed-length arrays and dynamic arrays, pushing a value into it).

```
a: []      ; Empty.
b: ["foo"] ; Pre-initialized.
```

Inserting and appending.

```
append a ["up" "down"] ; -> ["up" "down"]
insert a [left right]  ; -> [left right "up" "down"]
```

Getting specific values.

```
first a ; -> left
third a ; -> "up"
last a  ; -> "down"
a/2     ; -> right (Note: REBOL is 1-based.)
```

Getting subsequences. REBOL allows relative motion through a block (list). The list variable returns the current position to the end of the list, you can even assign to it without destroying the list.

```
a         ; -> [left right "up" "down"]
next a    ; -> [right "up" "down"]
skip a 2  ; -> ["up" "down"]

a: next a ; -> [right "up" "down"]
head a    ; -> [left right "up" "down"]

copy a                 ; -> [left right "up" "down"]
copy/part a 2          ; -> [left right]
copy/part  skip a 2  2 ; -> ["up" "down"]
```

Red

```
arr1: []      ;create empty array
arr2: ["apple" "orange" 1 2 3]    ;create an array with data
>> insert arr1 "blue"
>> arr1
== ["blue"]
append append arr1 "black" "green"
>> arr1
== ["blue" "black" "green"]
>> arr1/2
== "black"
>> second arr1
== "black"
>> pick arr1 2
== "black"
```

A vector! is a high-performance series! of items. The items in a vector! must all have the same type. The allowable item types are: integer! float! char! percent! Vectors of string! are not allowed.

```
>> vec1: make vector! [ 20 30 70]
== make vector! [20 30 70]
>> vec1/2
== 30
>> second vec1
== 30
>> append vec1 90
== make vector! [20 30 70 90]
>> append vec1 "string"
*** Script Error: invalid argument: "string"
*** Where: append
*** Stack:
>> append vec1 3.0
*** Script Error: invalid argument: 3.0
*** Where: append
*** Stack:
```

Averages/Arithmetic mean

Write a program to find the mean (arithmetic average) of a numeric vector.

In case of a zero-length input, since the mean of an empty set of numbers is ill-defined, the program may choose to behave in any way it deems appropriate, though if the programming language has an established convention for conveying math errors or undefined values, it's preferable to follow it.

```
average: func [v /local sum][
        if empty? v [return 0]
        sum: 0
        forall v [sum: sum + v/1]
        sum / length? v
]

; Note precision loss as spread increased.

print [mold x: [] "->" average x]
print [mold x: [3 1 4 1 5 9] "->" average x]
print [mold x: [1000 3 1 4 1 5 9 -1000] "->" average x]
print [mold x: [1e20 3 1 4 1 5 9 -1e20] "->" average x]
```

Output:

```
[] -> 0
[3 1 4 1 5 9] -> 3.83333333333333
[1000 3 1 4 1 5 9 -1000] -> 2.875
[1E+20 3 1 4 1 5 9 -1E+20] -> 0.0
```

Averages/Median

Write a program to find the median value of a vector of floating-point numbers.

The program need not handle the case where the vector is empty, but *must* handle the case where there are an even number of elements. In that case, return the average of the two middle values.

There are several approaches to this. One is to sort the elements, and then pick the element(s) in the middle.

Sorting would take at least $O(n \log n)$. Another approach would be to build a priority queue from the elements, and then extract half of the elements to get to the middle element(s). This would also take $O(n \log n)$. The best solution is to use the selection algorithm to find the median in $O(n)$ time.

```
median: func [
    "Returns the midpoint value in a series of numbers; half the values are
above, half are below."
    block [any-block!]
    /local len mid
][
    if empty? block [return none]
    block: sort copy block
    len: length? block
    mid: to integer! len / 2
    either odd? len [
        pick block add 1 mid
    ][
        (block/:mid) + (pick block add 1 mid) / 2
    ]
]
```

Classes

In object-oriented programming **class** is a set (a transitive closure) of types bound by the relation of inheritance. It is said that all types derived from some base type T and the type T itself form a class T.

The first type T from the class T sometimes is called the **root type** of the class.

A class of types itself, as a type, has the values and operations of its own. The operations of are usually called **methods** of the root type. Both operations and values are called polymorphic.

A polymorphic operation (method) selects an implementation depending on the actual specific type of the polymorphic argument.

The action of choice the type-specific implementation of a polymorphic operation is called **dispatch**. Correspondingly, polymorphic operations are often called **dispatching** or **virtual**. Operations with multiple arguments and/or the results of the class are called **multi-methods**. A further generalization of is the operation with arguments and/or results from different classes.

- single-dispatch languages are those that allow only one argument or result to control the dispatch. Usually it is the first parameter, often hidden, so that a prefix notation $x.f()$ is used instead of mathematical $f(x)$.
- multiple-dispatch languages allow many arguments and/or results to control the dispatch.

A polymorphic value has a type tag indicating its specific type from the class and the corresponding specific value of that type. This type is sometimes called **the most specific type** of a [polymorphic] value. The type tag of the value is used in order to resolve the dispatch. The set of polymorphic values of a class

is a transitive closure of the sets of values of all types from that class.

In many OO languages the type of the class of T and T itself are considered equivalent. In some languages they are distinct (like in Ada). When class T and T are equivalent, there is no way to distinguish polymorphic and specific values.

Create a basic class with a method, a constructor, an instance variable and how to instantiate it.

```
; Objects are derived from the base 'object!' type. REBOL uses a
; prototyping object system, so any object can be treated as a class,
; from which to derive others.

cowboy: make object! [
        name: "Tex"  ; Instance variable.
        hi: does [   ; Method.
                print [self/name ": Howdy!"]]
]

; I create two instances of the 'cowboy' class.

tex: make cowboy []
roy: make cowboy [
        name: "Roy"  ; Override 'name' property.
]

print "Say 'hello', boys:"  tex/hi  roy/hi
print ""

; Now I'll subclass 'cowboy'. Subclassing looks a lot like instantiation:

legend: make cowboy [
        deed: "..."
        boast: does [ print [self/name ": I once" self/deed "!"]] ]

; Instancing the legend:

pecos: make legend [name: "Pecos Bill"  deed: "lassoed a twister"]

print "Howdy, Pecos!"  pecos/hi
print "Tell us about yourself?"  pecos/boast
```

Output:

```
Say 'hello', boys:
Tex : Howdy!
Roy : Howdy!
```

```
Howdy, Pecos!
Pecos Bill : Howdy!
Tell us about yourself?
Pecos Bill : I once lassoed a twister !
```

Comma quibbling

Comma quibbling is a task originally set by Eric Lippert in his blog.

Write a function to generate a string output which is the concatenation of input words from a list/sequence where:

1. An input of no words produces the output string of just the two brace characters "{}".
2. An input of just one word, e.g. ["ABC"], produces the output string of the word inside the two braces, e.g. "{ABC}".
3. An input of two words, e.g. ["ABC", "DEF"], produces the output string of the two words inside the two braces with the words separated by the string " and ", e.g. "{ABC and DEF}".
4. An input of three or more words, e.g. ["ABC", "DEF", "G", "H"], produces the output string of all but the last word separated by ", " with the last word separated by " and " and all within braces; e.g. "{ABC, DEF, G and H}".

Test your function with the following series of inputs showing your output here on this page:

- [] # (No input words).
- ["ABC"]
- ["ABC", "DEF"]
- ["ABC", "DEF", "G", "H"]

Note: Assume words are non-empty strings of uppercase characters for this task.

Straightforward implementation

```
comma-quibbling: func [block] [
    rejoin [
        "^{"

        to-string use [s] [
            s: copy block
            s: next s
            forskip s 2 [insert s either tail? next s [" and "] [", "]]
            s: head s
        ]

        "^}"
    ]
]

foreach t [[] [ABC] [ABC DEF] [ABC DEF G H]] [print comma-quibbling t]
```

Output:

```
{}
{ABC}
{ABC and DEF}
{ABC, DEF, G and H}
```

Alternative (more efficient) version with oxford comma switch

```
; builds string instead of using an intermediate block

comma-quibbling: func [block /oxford /local s length] [
    length: length? block
    rejoin [
        "^{"

        either length < 2 [to-string block] [
            s: to-string block/1
            for n 2 (length - 1) 1 [repend s [", " pick block n]]
            if all [oxford (length > 2)] [append s ","]
            repend s [" and " last block]
        ]

        "^}"
    ]
]

test: [[] [ABC] [ABC DEF] [ABC DEF G H]]
```

```
foreach t test [print comma-quibbling t]
print "Now with Oxford comma"
foreach t test [print comma-quibbling/oxford t]
```

Output:

```
{}
{ABC}
{ABC and DEF}
{ABC, DEF, G and H}
Now with Oxford comma
{}
{ABC}
{ABC and DEF}
{ABC, DEF, G, and H}
```

Copy a string

This task is about copying a string.

Where it is relevant, distinguish between copying the contents of a string versus making an additional reference to an existing string.

```
x: y: "Testing."
y/2: #"X"
print ["Both variables reference same string:" mold x "," mold y]

x: "Slackeriffic!"
print ["Now reference different strings:" mold x "," mold y]

y: copy x        ; String copy here!
y/3: #"X"        ; Modify string.
print ["x copied to y, then modified:" mold x "," mold y]

y: copy/part x 7 ; Copy only the first part of y to x.
print ["Partial copy:" mold x "," mold y]

y: copy/part  skip x 2  3
print ["Partial copy from offset:" mold x "," mold y]
```

Output:

```
Script: "String Copy" (16-Dec-2009)
Both variables reference same string: "TXsting." , "TXsting."
Now reference different strings: "Slackeriffic!" , "TXsting."
x copied to y, then modified: "Slackeriffic!" , "SlXckeriffic!"
```

```
Partial copy: "Slackeriffic!" , "Slacker"
Partial copy from offset: "Slackeriffic!" , "ack"
```

Red

```
originalString: "hello wordl"
copiedString: originalString
; OR
copiedString2: copy originalString
```

Date format

Display the current date in the formats of:

- **2007-11-23** and
- **Sunday, November 23, 2007**

```
; REBOL has no built-in pictured output.

zeropad: func [pad n][
    n: to-string n
    insert/dup n "0" (pad - length? n)
    n
]
d02: func [n][zeropad 2 n]

print now ; Native formatting.

print rejoin [now/year  "-"  d02 now/month  "-"  d02 now/day]

print rejoin [
        pick system/locale/days now/weekday ", "
        pick system/locale/months now/month " "
        now/day ", " now/year
]
```

Output:

```
6-Dec-2009/10:02:10-5:00
2009-12-06
Sunday, December 6, 2009
```

Date manipulation

Given the date string "March 7 2009 7:30pm EST",
output the time 12 hours later in any human-readable format.

As extra credit, display the resulting time in a time zone different from your own.

```
; Only North American zones here -- feel free to extend for your area.

zones: [
        NST -3:30 NDT -2:30 AST -4:00 ADT -3:00 EST -5:00 EDT -4:00
        CST -6:00 CDT -5:00 MST -7:00 MDT -6:00 PST -8:00 PDT -7:00 AKST -9:00
        AKDT -8:00 HAST -10:00 HADT -9:00]

read-time: func [
        text
        /local m d y t z
][
        parse load text [
                set m word! (m: index? find system/locale/months to-string m)
                set d integer!  set y integer!
                set t time!  set tz word!]
        to-date reduce [y m d t  zones/:tz]
]

print 12:00 + read-time "March 7 2009 7:30pm EST"
```

Output:

```
8-Mar-2009/7:30-5:00
```

Red

```
d: 07-Mar-2009/19:30 + 12:00
print d
8-Mar-2009/7:30:00
d/timezone: 1
print d
8-Mar-2009/8:30:00+01:00
```

Day of the week

A company decides that whenever Xmas falls on a Sunday they will give their workers all extra paid holidays so that, together with any public holidays, workers will not have to work the following week (between the 25th of December and the first of January).

In what years between 2008 and 2121 will the 25th of December be a Sunday?

Using any standard date handling libraries of your programming language; compare the dates calculated with the output of other languages to discover any anomalies in the handling of dates which may be due to, for example, overflow in types used to represent dates/times similar to y2k type problems.

```
for y 2008 2121 1 [
        d: to-date reduce [y 12 25]
        if 7 = d/weekday [prin [y ""]]
]
```

Output:

```
2011 2016 2022 2033 2039 2044 2050 2061 2067 2072 2078 2089 2095 2101 2107
2112 2118
```

Red

```
Red []
repeat yy 114 [
  d: to-date reduce [25 12 (2007 + yy )]
  if 7 = d/weekday [ print d ] ;; 7 = sunday
]
;; or
print "version 2"

d: to-date [25 12 2008]
while [d <= 25/12/2121 ] [
  if 7 = d/weekday [
    print rejoin [d/day '. d/month '. d/year ]
  ]
  d/year: d/year + 1
]
```

Output:

```
25-Dec-2011  25-Dec-2016  25-Dec-2022
```

Detect division by zero

Write a function to detect a *divide by zero error* without checking if the denominator is zero.

```
; The 'try' word returns an error object if the operation fails for
; whatever reason. The 'error?' word detects an error object and
; 'disarm' keeps it from triggering so I can analyze it to print the
; appropriate message. Otherwise, any reference to the error object
; will stop the program.

div-check: func [
        "Attempt to divide two numbers, report result or errors as needed."
        x y
        /local result
] [
        either error? result: try [x / y][
                result: disarm result
                print ["Caught" result/type "error:" result/id]
        ] [
                print [x "/" y "=" result]
        ]
]

div-check 12 2       ; An ordinary calculation.
div-check 6 0        ; This will detect divide by zero.
div-check "7" 0.0001 ; Other errors can be caught as well.
```

Output:

```
12 / 2 = 6
Caught math error: zero-divide
Caught script error: cannot-use
```

Determine if a string is numeric

Create a boolean function which takes in a string and tells whether it is a numeric string (floating point and negative numbers included) in the syntax the language uses for numeric literals or numbers converted from strings.

```
; Built-in.

numeric?: func [x][not error? try [to-decimal x]]

; Parse dialect for numbers.

sign:   [0 1 "-"]
digit:  charset "0123456789"
int:    [some digit]
float:  [int "." int]
number: [
        sign float ["e" | "E"] sign int |
        sign int ["e" | "E"] sign int |
        sign float |
        sign int ]

pnumeric?: func [x][parse x number]

; Test cases.

cases: parse {
   10 -99
   10.43 -12.04
   1e99 1.0e10 -10e3 -9.12e7 2e-4 -3.4E-5
   3phase  Garkenhammer  e  n3v3r  phase3
} none
foreach x cases [print [x  numeric? x  pnumeric? x]]
```

Dot product

Create a function/use an in-built function, to compute the **dot product**, also known as the **scalar product** of two vectors.

If possible, make the vectors of arbitrary length.

As an example, compute the dot product of the vectors:

```
[1,  3, -5]     and
[4, -2, -1]
```

If implementing the dot product of two vectors directly:

- each vector must be the same length
- multiply corresponding terms from each vector
- sum the products (to produce the answer)

```
a: [1 3 -5]
b: [4 -2 -1]

dot-product: function [v1 v2] [sum] [
    if (length? v1) != (length? v2) [
        make error! "error: vector sizes must match"
    ]
    sum: 0
    repeat i length? v1 [
        sum: sum + ((pick v1 i) * (pick v2 i))
    ]
]

dot-product a b
```

Dynamic variable names

Create a variable with a user-defined name.

The variable name should *not* be written in the program text, but should be taken from the user dynamically.

```
; Here, I ask the user for a name, then convert it to a word and
; assign the value "Hello!" to it. To read this phrase, realize that
; REBOL collects terms from right to left, so "Hello!" is stored for
; future use, then the prompt string "Variable name? " is used as the
; argument to ask (prompts user for input). The result of ask is
; converted to a word so it can be an identifier, then the 'set' word
; accepts the new word and the string ("Hello!") to be assigned.

set  to-word  ask "Variable name? "  "Hello!"
```

Session output:

```
Variable name? glister
== "Hello!"
>> glister
== "Hello!"
```

Echo server

Create a network service that sits on TCP port 12321, which accepts connections on that port, and which echoes complete lines (using a carriage-return/line-feed sequence as line separator) back to clients. No error handling is required. For the purposes

of testing, it is only necessary to support connections from localhost (127.0.0.1 or perhaps ::1). Logging of connection information to standard output is recommended.

The implementation must be able to handle simultaneous connections from multiple clients. A multi-threaded or multi-process solution may be used. Each connection must be able to echo more than a single line.

The implementation must not stop responding to other clients if one client sends a partial line or stops reading responses.

```
server-port: open/lines tcp://:12321
forever [
    connection-port: first server-port
    until [
        wait connection-port
        error? try [insert connection-port first connection-port]
    ]
    close connection-port
]
close server-port
```

Environment variables

Show how to get one of your process's environment variables.

The available variables vary by system; some of the common ones available on Unix include:

- PATH
- HOME
- USER

```
print get-env "HOME"
```

Execute a system command

Run either the **ls** system command (**dir** on Windows), or the **pause** system command.

```
; Capture output to string variable:

x: ""  call/output "dir" x
```

```
print x

; The 'console' refinement displays the command output on the REBOL command
line.

call/console "dir *.r"
call/console "ls *.r"

call/console "pause"

; The 'shell' refinement may be necessary to launch some programs.

call/shell "notepad.exe"
```

Red

```
call/show %pause        ;The /show refinement forces the display of system's
shell window (Windows only).
call/show %dir
call/show %notepad.exe
```

Factorial

Definitions

- The factorial of **0** (zero) is defined as being 1 (unity).
- The **Factorial Function** of a positive integer, *n*, is defined as the product of the sequence:

$$n, \quad n-1, \quad n-2, \quad \ldots \quad 1$$

Write a function to return the factorial of a number.

Solutions can be iterative or recursive.

Support for trapping negative *n* errors is optional.

```
; Standard recursive implementation.

factorial: func [n][
        either n > 1 [n * factorial n - 1] [1]
]

; Iteration.
```

```
ifactorial: func [n][
	f: 1
	for i 2 n 1 [f: f * i]
	f
]

; Automatic memoization.
; I'm just going to say up front that this is a stunt. However, you've
; got to admit it's pretty nifty. Note that the 'memo' function
; works with an unlimited number of arguments (although the expected
; gains decrease as the argument count increases).

memo: func [
	"Defines memoizing function -- keeps arguments/results for later use."
	args [block!] "Function arguments. Just specify variable names."
	body [block!] "The body block of the function."
	/local m-args m-r
][
	do compose/deep [
		func [
			(args)
			/dump "Dump memory."
		][
			m-args: []
			if dump [return m-args]

			if m-r: select/only m-args reduce [(args)] [return m-
r]

			m-r: do [(body)]
			append m-args reduce [reduce [(args)] m-r]
			m-r
		]
	]
]

mfactorial: memo [n][
	either n > 1 [n * mfactorial n - 1] [1]
]

; Test them on numbers zero to ten.

for i 0 10 1 [print [i ":" factorial i  ifactorial i  mfactorial i]]
```

Output:

```
0 : 1 1 1
1 : 1 1 1
2 : 2 2 2
3 : 6 6 6
4 : 24 24 24
5 : 120 120 120
```

```
6 : 720 720 720
7 : 5040 5040 5040
8 : 40320 40320 40320
9 : 362880 362880 362880
10 : 3628800 3628800 3628800
```

Filter

Select certain elements from an Array into a new Array in a generic way.
To demonstrate, select all even numbers from an Array.

As an option, give a second solution which filters destructively, by modifying the original Array rather than creating a new Array.

```
a: []  repeat i 100 [append a i] ; Build and load array.

evens: []  repeat element a [if even? element [append evens element]]

print mold evens
```

Output:

```
[2 4 6 8 10 12 14 16 18 20 22 24
26 28 30 32 34 36 38 40 42 44 46 48 50
52 54 56 58 60 62 64 66 68 70 72 74 76
78 80 82 84 86 88 90 92 94 96 98 100]
```

Red

```
Red []
orig: [] repeat i 10 [append orig i]
?? orig
cpy: [] forall orig [if even? orig/1 [append cpy orig/1]]
;; or - because we know each second element is even :- )
;; cpy: extract next orig 2
?? cpy
remove-each ele orig [odd? ele]    ;; destructive
?? orig
```

Output:

```
orig: [1 2 3 4 5 6 7 8 9 10]
cpy: [2 4 6 8 10]
orig: [2 4 6 8 10]
```

Find the last Sunday of each month

Write a program or a script that returns the last Sundays of each month of a given year. The year may be given through any simple input method in your language (command line, std in, etc).

```
#!/usr/bin/env rebol

last-sundays-of-year: function [
    "Return series of last sunday (date!) for each month of the year"
    year [integer!] "which year?"
  ][
    d: to-date reduce [1 1 year]           ; start with first day of year
    collect [
        repeat month 12 [
            d/month: month + 1             ; move to start of next month
            keep d - d/weekday             ; calculate last sunday & keep
        ]
    ]
]

foreach sunday last-sundays-of-year to-integer system/script/args [print
sunday]
```

Output:

```
./last-sundays.reb 2013
27-Jan-2013
24-Feb-2013
31-Mar-2013
28-Apr-2013
26-May-2013
30-Jun-2013
28-Jul-2013
25-Aug-2013
29-Sep-2013
27-Oct-2013
24-Nov-2013
29-Dec-2013
```

First-class functions

A language has first-class functions if it can do each of the following without recursively invoking a compiler or interpreter or otherwise metaprogramming:

- Create new functions from preexisting functions at run-time
- Store functions in collections
- Use functions as arguments to other functions
- Use functions as return values of other functions

Write a program to create an ordered collection *A* of functions of a real number. At least one function should be built-in and at least one should be user-defined; try using the sine, cosine, and cubing functions. Fill another collection *B* with the inverse of each function in *A*. Implement function composition as in Functional Composition. Finally, demonstrate that the result of applying the composition of each function in *A* and its inverse in *B* to a value, is the original value. (Within the limits of computational accuracy).

(A solution need not actually call the collections "A" and "B". These names are only used in the preceding paragraph for clarity.)

This example is **incomplete**. Fails to demonstrate that the result of applying the composition of each function in A and its inverse in B to a value, is the original value Please ensure that it meets all task requirements and remove this message.

```
; Functions "foo" and "bar" are used to prove that composition
; actually took place by attaching their signatures to the result.

foo: func [x][reform ["foo:" x]]
bar: func [x][reform ["bar:" x]]

cube:  func [x][x * x * x]
croot: func [x][power  x  1 / 3]

; "compose" means something else in REBOL, so I "fashion" an alternative.

fashion: func [f1 f2][
        do compose/deep [func [x][(:f1) (:f2) x]]]
```

```
A: [foo sine    cosine    cube]
B: [bar arcsine arccosine croot]

while [not tail? A][
        fn: fashion get A/1 get B/1
        source fn ; Prove that functions actually got composed.
        print [fn 0.5  crlf]

        A: next A  B: next B  ; Advance to next pair.
]
```

FizzBuzz

Write a program that prints the integers from **1** to **100** (inclusive).
But:

- for multiples of three, print **Fizz** (instead of the number)
- for multiples of five, print **Buzz** (instead of the number)
- for multiples of both three and five, print **FizzBuzz** (instead of the number)

The *FizzBuzz* problem was presented as the lowest level of comprehension required to illustrate adequacy.

```
; Concatenative. Note use of 'case/all' construct to evaluate all
; conditions. I use 'copy' to allocate a new string each time through
; the loop -- otherwise 'x' would get very long...

repeat i 100 [
        x: copy ""
        case/all [
                0 = mod i 3 [append x "Fizz"]
                0 = mod i 5 [append x "Buzz"]
                "" = x      [x: mold i] ]
        print x ]
```

Here is an example by Nick Antonaccio.

```
repeat i 100 [
    print switch/default 0 compose [
        (mod i 15) ["fizzbuzz"]
        (mod i 3)  ["fizz"]
        (mod i 5)  ["buzz"]
    ][i]
]
```

And a minimized version:

```
repeat i 100[j:""if i // 3 = 0[j:"fizz"]if i // 5 = 0[j: join j"buzz"]if""=
j[j: i]print j]
```

The following is presented as a curiosity only, not as an example of good coding practice:

```
m: func [i d] [0 = mod i d]
spick: func [t x y][either any [not t  "" = t][y][x]]
zz: func [i] [rejoin [spick m i 3 "Fizz" ""  spick m i 5 "Buzz" ""]]
repeat i 100 [print spick z: zz i z i]
```

Flatten a list

Write a function to flatten the nesting in an arbitrary list of values.

Your program should work on the equivalent of this list:

```
  [[1], 2, [[3, 4], 5], [[[]]], [[[6]]], 7, 8, []]
```

Where the correct result would be the list:

```
   [1, 2, 3, 4, 5, 6, 7, 8]
```

```
flatten: func [
    "Flatten the block in place."
    block [any-block!]
][
    parse block [
        any [block: any-block! (change/part block first block 1) :block |
skip]
    ]
    head block
]
```

Sample:

```
>> flatten [[1] 2 [[3 4] 5] [[[]]] [[[6]]] 7 8 []]
== [1 2 3 4 5 6 7 8]
```

Red

```
flatten: function [
    "Flatten the block"
    block [any-block!]
][
    load form block
]

red>> flatten [[1] 2 [[3 4] 5] [[[]]] [[[6]]] 7 8 []]
== [1 2 3 4 5 6 7 8]

;flatten a list to a string
>> blk: [1 2 ["test"] "a" [["bb"]] 3 4 [[[99]]]]
>> form blk
== "1 2 test a bb 3 4 99"
```

Flow-control structures

Document common flow-control structures.

One common example of a flow-control structure is the goto construct.

Note that Conditional Structures and Loop Structures have their own articles/categories.

```
; return -- Return early from function (normally, functions return
; result of last evaluation).

hatefive: func [
        "Prints value unless it's the number 5."
        value "Value to print."
][
        if value = 5 [return "I hate five!"]
        print value
]

print "Function hatefive, with various values:"
hatefive 99
hatefive 13
hatefive 5
hatefive 3

; break -- Break out of current loop.

print [crlf "Loop to 10, but break out at five:"]
repeat i 10 [
        if i = 5 [break]
        print i
]
```

```
; catch/throw -- throw breaks out of a code block to enclosing catch.

print [crlf "Start to print two lines, but throw out after the first:"]
catch [
        print "First"
        throw "I'm done!"
        print "Second"
]

; Using named catch blocks, you can select which catcher you want when
throwing.

print [crlf "Throw from inner code block, caught by outer:"]
catch/name [
        print "Outer catch block."
        catch/name [
                print "Inner catch block."
                throw/name "I'm done!" 'Johnson
                print "We never get here."
        ] 'Clemens
        print "We never get here, either."
] 'Johnson

; try

div: func [
        "Divide first number by second."
        a b
        /local r "Result"
][
        if error? try [r: a / b] [r: "Error!"]
        r ; Functions return last value evaluated.
]

print [crlf "Report error on bad division:"]
print div 10 4
print div 10 2
print div 10 1
print div 10 0
```

Formatted numeric output

Express a number in decimal as a fixed-length string with leading zeros.

For example, the number **7.125** could be expressed as **00007.125**

```
; REBOL has no built-in facilities for printing pictured output.
; However, it's not too hard to cook something up using the
; string manipulation facilities.

zeropad: func [
        "Pad number with zeros or spaces. Works on entire number."
        pad "Number of characters to pad to."
        n "Number to pad."
        /space "Pad with spaces instead."
        /local nn c s
][
        n: to-string n  c: " "  s: ""
         if not space [
                c: "0"
                if #"-" = n/1 [pad: pad - 1  n: copy skip n 1  s: "-"]
        ]

        insert/dup n c (pad - length? n)
        insert n s
    n
]

; These tests replicate the C example output.

print [zeropad/space 9 negate 7.125]
print [zeropad/space 9 7.125]
print 7.125
print [zeropad 9 negate 7.125]
print [zeropad 9 7.125]
print 7.125
```

Output:

```
   -7.125
    7.125
7.125
-0007.125
00007.125
7.125
```

FTP

Connect to a server, change directory, list its contents and download a file as binary using the FTP protocol. Use passive mode if available.

```
system/schemes/ftp/passive: on
print read ftp://kernel.org/pub/linux/kernel/
write/binary %README read/binary ftp://kernel.org/pub/linux/kernel/README
```

Function composition

Create a function, compose, whose two arguments f and g, are both functions with one argument.

The result of compose is to be a function of one argument, (lets call the argument x), which works like applying function f to the result of applying function g to x.

Example

compose(f, g) (x) = $f(g(x))$

Reference: Function composition

Hint: In some languages, implementing compose correctly requires creating a closure.

```
; "compose" means something else in REBOL, therefore I use a 'compose-
functions name.

compose-functions: func [
    {compose the given functions F and G}
    f [any-function!]
    g [any-function!]
] [
    func [x] compose [(:f) (:g) x]
]
```

Functions "foo" and "bar" are used to prove that composition actually took place by attaching their signatures to the result.

```
foo: func [x] [reform ["foo:" x]]
bar: func [x] [reform ["bar:" x]]

foo-bar: compose-functions :foo :bar
print ["Composition of foo and bar:"  mold foo-bar "test"]

sin-asin: compose-functions :sine :arcsine
print [crlf "Composition of sine and arcsine:" sin-asin 0.5]
```

Output:

```
Composition of foo and bar: "foo: bar: test"

Composition of sine and arcsine: 0.5
```

Generic swap

Write a generic swap function or operator which exchanges the values of two variables (or, more generally, any two storage places that can be assigned), regardless of their types.

If your solution language is statically typed please describe the way your language provides genericity.

If variables are typed in the given language, it is permissible that the two variables be constrained to having a mutually compatible type, such that each is permitted to hold the value previously stored in the other without a type violation. That is to say, solutions do not have to be capable of exchanging, say, a string and integer value, if the underlying storage locations are not attributed with types that permit such an exchange.

Generic swap is a task which brings together a few separate issues in programming language semantics.

Dynamically typed languages deal with values in a generic way quite readily, but do not necessarily make it easy to write a function to destructively swap two variables, because this requires indirection upon storage places or upon the syntax designating storage places.

Functional languages, whether static or dynamic, do not necessarily allow a destructive operation such as swapping two variables regardless of their generic capabilities.

Some static languages have difficulties with generic programming due to a lack of support for (Parametric Polymorphism).

```
swap: func [
        "Swap contents of variables."
        a [word!] b [word!] /local x
][
        x: get a
        set a get b
        set b x ]

answer: 42  ship: "Heart of Gold"
swap 'answer 'ship ; Note quoted variables.
print rejoin ["The answer is " answer ", the ship is " ship "."]
```

Greatest common divisor

Find the greatest common divisor of two integers.

```
gcd: func [
    {Returns the greatest common divisor of m and n.}
    m [integer!]
    n [integer!]
    /local k
] [
    ; Euclid's algorithm
    while [n > 0] [
        k: m
        m: n
        n: k // m
    ]
    m
]
```

Greatest element of a list

Create a function that returns the maximum value in a provided set of values,
where the number of values may not be known until run-time.

```
max: func [
        "Find maximum value in a list."
        values [series!] "List of values."
] [
        first maximum-of values
]

print ["Max of"  mold d: [5 4 3 2 1]  "is"  max d]
print ["Max of"  mold d: [-5 -4 -3 -2 -1]  "is"  max d]
```

Output:

```
Max of [5 4 3 2 1] is 5
Max of [-5 -4 -3 -2 -1] is -1
```

Red

```
Red []
list: [1 2 3 5 4]
print  last sort list
```

Hailstone sequence

The Hailstone sequence of numbers can be generated from a starting positive integer, n by:

- If n is **1** then the sequence ends.
- If n is **even** then the next n of the sequence = n/2
- If n is **odd** then the next n of the sequence = (3 * n) + 1

The (unproven) Collatz conjecture is that the hailstone sequence for any starting number always terminates.

The hailstone sequence is also known as *hailstone numbers* (because the values are usually subject to multiple descents and ascents like hailstones in a cloud).

This sequence is also known as the *Collatz sequence*.

1. Create a routine to generate the hailstone sequence for a number.
2. Use the routine to show that the hailstone sequence for the number 27 has 112 elements starting with 27, 82, 41, 124 and ending with 8, 4, 2, 1
3. Show the number less than 100,000 which has the longest hailstone sequence together with that sequence's length. (But don't show the actual sequence!)

```
hail: func [
        "Returns the hailstone sequence for n"
        n [integer!]
        /local seq
] [
        seq: copy reduce [n]
        while [n <> 1] [
                append seq n: either n % 2 == 0 [n / 2] [3 * n + 1]
        ]
        seq
]
```

```
hs27: hail 27
print [
        "the hail sequence of 27 has length" length? hs27
        "and has the form " copy/part hs27 3 "..."
        back back back tail hs27
]

maxN: maxLen: 0
repeat n 99999 [
        if (len: length? hail n) > maxLen [
                maxN: n
                maxLen: len
        ]
]

print [
        "the number less than 100000 with the longest hail sequence is"
        maxN "with length" maxLen
]
```

Output:

```
the hail sequence of 27 has length 112 and has the form  27 82 41 ... 4 2 1
the number less than 100000 with the longest hail sequence is 77031 with
length 351
```

Higher-order functions

Pass a function *as an argument* to another function.

```
map: func [
        "Apply function to contents of list, return new list."
        f [function!] "Function to apply to list."
        data [block! list!] "List to transform."
        /local result i
][
        result: copy []  repeat i data [append result f i]  result]

square: func [
        "Calculate x^2."
        x [number!]
][x * x]

cube: func [
        "Calculate x^3."
        x [number!]
][x * x * x]

; Testing:
```

```
x: [1 2 3 4 5]
print ["Data:   "  mold x]
print ["Squared:"  mold map :square x]
print ["Cubed:  "  mold map :cube x]
print ["Unnamed:"  mold map func [i][i * 2 + 1] x]
```

Output:

```
Data:    [1 2 3 4 5]
Squared: [1 4 9 16 25]
Cubed:   [1 8 27 64 125]
Unnamed: [3 5 7 9 11]
```

Horner's rule for polynomial evaluation

A fast scheme for evaluating a polynomial such as:

$$-19 + 7x - 4x^2 + 6x^3$$

when

$$x = 3.$$

is to arrange the computation as follows:

$$((((0)x + 6)x + (-4))x + 7)x + (-19)$$

And compute the result from the innermost brackets outwards as in this pseudocode:

```
coefficients := [-19, 7, -4, 6] # list coefficients of all x^0..x^n in order
x := 3
accumulator := 0
for i in length(coefficients) downto 1 do
    # Assumes 1-based indexing for arrays
    accumulator := ( accumulator * x ) + coefficients[i]
done
# accumulator now has the answer
```

Task Description

> Create a routine that takes a list of coefficients of a polynomial in order of increasing powers of x; together with a value of x to compute its value at, and return the value of the polynomial at that value using Horner's rule.

```
horner: func [coeffs x] [
    result: 0
    foreach i reverse coeffs [ result: (result * x) + i ]
    return result
    ]

print horner [-19 7 -4 6] 3
```

Increment a numerical string

Increment a numerical string.

```
; Note the use of unusual characters in function name. Also note that
; because REBOL collects terms from right to left, I convert the
; string argument (s) to integer first, then add that result to one.

s++: func [s][to-string 1 + to-integer s]

; Examples. Because the 'print' word actually evaluates the block
; (it's effectively a 'reduce' that gets printed space separated),
; it's possible for me to assign the test string to 'x' and have it
; printed as a side effect. At the end, 'x' is available to submit to
; the 's++' function. I 'mold' the return value of s++ to make it
; obvious that it's still a string.

print [x: "-99" "plus one equals" mold s++ x]
print [x: "42" "plus one equals" mold s++ x]
print [x: "12345" "plus one equals" mold s++ x]
```

Output:

```
-99 plus one equals "-98"
42 plus one equals "43"
12345 plus one equals "12346"
```

Inheritance/Single

Inheritance is an operation of type algebra that creates a new type from one or several parent types. The obtained type is called **derived type**. It inherits some of the properties of its parent types. Usually inherited properties are:

- methods
- components
- parts of the representation

The class of the new type is a **subclass** of the classes rooted in the parent types. When all (in certain sense) properties of the parents are preserved by the derived type, it is said to be a Liskov subtype. When properties are preserved then the derived type is *substitutable* for its parents in all contexts. Usually full substitutability is achievable only in some contexts.

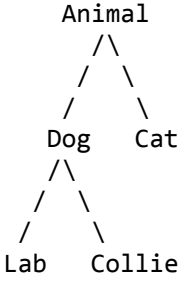
Inheritance is

- **single**, when only one parent is allowed
- **multiple**, otherwise

Some single inheritance languages usually allow multiple inheritance for certain abstract types, interfaces in particular.

Inheritance can be considered as a relation parent-child. Parent types are sometimes called **supertype**, the derived ones are **subtype**. This relation is transitive and reflexive. Types bound by the relation form a wp:Directed_acyclic_graph directed acyclic graph (ignoring reflexivity). With single inheritance it becomes a tree.

Task: Show a tree of types which inherit from each other. The top of the tree should be a class called Animal. The second level should have Dog and Cat. Under Dog should be Lab and Collie. None of the classes need to have any functions, the only thing they need to do is inherit from the specified superclasses (overriding functions should be shown in Polymorphism). The tree should look like this:



```
; REBOL provides subclassing through its prototype mechanism:

Animal: make object! [ legs: 4 ]

Dog: make Animal [ says: "Woof!" ]
Cat: make Animal [ says: "Meow..." ]

Lab: make Dog []
Collie: make Dog []

; Demonstrate inherited properties:

print ["Cat has" Cat/legs "legs."]

print ["Lab says:" Lab/says]
```

Output:

```
Cat has 4 legs.
Lab says: Woof!
```

Input loop

Read from a text stream either word-by-word or line-by-line until the stream runs out of data.

The stream will have an unknown amount of data on it.

```
; Slurp the whole file in:
x: read %file.txt

; Bring the file in by lines:
x: read/lines %file.txt

; Read in first 10 lines:
x: read/lines/part %file.txt 10

; Read data a line at a time:
f: open/lines %file.txt
while [not tail? f][
        print f/1
        f: next f ; Advance to next line.
]
close f
```

Interactive programming

Many language implementations come with an *interactive mode*.

This is a command-line interpreter that reads lines from the user and evaluates these lines as statements or expressions.

An interactive mode may also be known as a *command mode*, a *read-eval-print loop* (REPL), or a *shell*.

Show how to start this mode.

Then, as a small example of its use, interactively create a function of two strings and a separator that returns the strings separated by two concatenated instances of the separator (the 3rd argument).

Example

```
f('Rosetta', 'Code', ':')
```

should return

```
'Rosetta::Code'
```

Note

This task is *not* about creating your own interactive mode.

Start the REBOL/Core interpreter in quiet mode with -q. q to quit.

```
$ rebol -q
>> f: func [a b s] [print rejoin [a s s b]]
>> f "Rosetta" "Code" ":"
Rosetta::Code
>> q
```

Josephus problem

Josephus problem is a math puzzle with a grim description: n prisoners are standing on a circle, sequentially numbered from 0 to n − 1.

An executioner walks along the circle, starting from prisoner 0, removing every k-th prisoner and killing him.

As the process goes on, the circle becomes smaller and smaller, until only one prisoner remains, who is then freed. >

For example, if there are n = 5 prisoners and k = 2, the order the prisoners are killed in (let's call it the "killing sequence") will be 1, 3, 0, and 4, and the survivor will be #2.

Task

Given any n , $k > 0$, find out which prisoner will be the final survivor.

In one such incident, there were 41 prisoners and every 3^{rd} prisoner was being killed $(k = 3)$.

Among them was a clever chap name Josephus who worked out the problem, stood at the surviving position, and lived on to tell the tale.

Which number was he?

Extra

The captors may be especially kind and let m survivors free, and Josephus might just have $m - 1$ friends to save.

Provide a way to calculate which prisoner is at any given position on the killing sequence.

Notes

1. You can always play the executioner and follow the procedure exactly as described, walking around the circle, counting (and cutting off) heads along the way. This would yield the complete killing sequence and answer the above questions, with a complexity of probably O (k n). However, individually it takes no more than O (m) to find out which prisoner is the m-th to die.
2. If it's more convenient, you can number prisoners from 1 to n instead. If you choose to do so, please state it clearly.
3. An alternative description has the people committing assisted suicide instead of being executed, and the last person simply walks away. These details are not relevant, at least not mathematically.

```
execute: func [death-list [block!] kill [integer!]] [
    assert [not empty? death-list]
    until [
        loop kill - 1 [append death-list take death-list]
        (1 == length? remove death-list)
    ]
]

prisoner: [] for n 0 40 1 [append prisoner n]
execute prisoner 3
print ["Prisoner" prisoner "survived"]
```

Output:

```
Prisoner 30 survived
```

And any kind of list will do:

```
for-the-chop: [Joe Jack William Averell Rantanplan]
execute for-the-chop 2
print [for-the-chop "survived"]
```

Output:

```
William survived
```

JSON

Load a JSON string into a data structure. Also, create a new data structure and serialize it into JSON.

Use objects and arrays (as appropriate for your language) and make sure your JSON is valid (https://jsonformatter.org, https://codebeautify.org/jsonvalidator, https://jsonlint.com/ or https://extendsclass.com/json-validator.html).

Using json.org/json.r

```
json-str: {{"menu": {
    "id": "file",
    "string": "File:",
    "number": -3,
    "boolean": true,
    "boolean2": false,
    "null": null,
    "array": [1, 0.13, null, true, false, "\t\r\n"],
    "empty-string": ""
  }
}}

res: json-to-rebol json-str
js: rebol-to-json res
```

json-to-rebol Result:

```
make object! [
    menu: make object! [
        id: "file"
        string: "File:"
        number: -3
        boolean: true
        boolean2: false
        null: none
        array: [1 0.13 none true false "^-^M^/"]
        empty-string: ""
    ]
]
```

Keyboard macros

Show how to link user defined methods to user defined keys.

An example of this is the facility provided by emacs for key bindings.

These key bindings may be application-specific or system-wide; state which you have done.

```
; Application specific keyboard bindings using REBOL VID
; dialect. Implementation of the "Averageman" calculator --
; See http://www.atariarchives.org/bcc2/showpage.php?page=63 for details.

view layout [
        style btn button coal 46
        across

        display: h1 100 red maroon right ""  return

; Key shortcuts are specified as character arguments to widget
; descriptions in the layout.

        btn "1" #"1" [set-face display "1"]
        btn "+" #"+" [set-face display ""]
        return

        pad 54
        btn "=" #"=" [set-face display "3"]

        pad 1x100 return
        text "(c) 1977 G. Beker"
]
```

Knuth shuffle

The Knuth shuffle (a.k.a. the Fisher-Yates shuffle) is an algorithm for randomly shuffling the elements of an array.

Implement the Knuth shuffle for an integer array (or, if possible, an array of any type).

Specification

Given an array ***items*** with indices ranging from ***0*** to ***last***, the algorithm can be defined as follows (pseudo-code):

```
for i from last downto 1 do:
    let j = random integer in range 0 ≤ j ≤ i
    swap items[i] with items[j]
```

Notes:

- It modifies the input array in-place. If that is unreasonable in your programming language, you may amend the algorithm to return the shuffled items as a new array instead.
- The algorithm can also be amended to iterate from left to right, if that is more convenient.

Test cases

Input array	Possible output arrays
[]	[]
[10]	[10]
[10, 20]	[10, 20] [20, 10]
[10, 20, 30]	[10, 20, 30] [10, 30, 20] [20, 10, 30] [20, 30, 10] [30, 10, 20] [30, 20, 10]

```
fisher-yates: func [b [block!] /local n i j k] [
    n: length? b: copy b
    i: n
    while [i > 1] [
        if i <> j: random i [
            error? set/any 'k pick b j
            change/only at b j pick b i
            change/only at b i get/any 'k
        ]
        i: i - 1
    ]
    b
]
```

Last Friday of each month

Write a program or a script that returns the date of the last Fridays of each month of a given year.

The year may be given through any simple input method in your language (command line, std in, etc).

The longer version:

```
leap-year?:  function [year] [to-logic attempt [to-date reduce [29 2 year]]]

days-in-feb: function [year] [either leap-year? year [29] [28]]

days-in-month: function [month year] [
    do pick [31 (days-in-feb year) 31 30 31 30 31 31 30 31 30 31] month
]

last-day-of-month: function [month year] [
    to-date reduce [year month  days-in-month month year]
]

last-weekday-of-month: function [weekday month year] [
    d: last-day-of-month month year
    while [d/weekday != weekday] [d/day: d/day - 1]
    d
]

last-friday-of-month: function [month year] [last-weekday-of-month 5 month
year]

year: to-integer input
repeat month 12 [print last-friday-of-month month year]
```

Output:

```
rebol last-fridays.reb <<< 2012
27-Jan-2012
24-Feb-2012
30-Mar-2012
27-Apr-2012
25-May-2012
29-Jun-2012
27-Jul-2012
31-Aug-2012
28-Sep-2012
26-Oct-2012
30-Nov-2012
28-Dec-2012
```

A shorter version:

```
last-fridays-of-year: function [year] [
    collect [
        repeat month 12 [
            d: to-date reduce [1 month year]
            d/month: d/month + 1                        ; start of next month
            until [d/day: d/day - 1  d/weekday = 5]   ; go backwards until
find a Friday
            keep d
        ]
    ]
]

foreach friday last-fridays-of-year to-integer input [print friday]
```

NB. See "Find the last Sunday of each month" Rosetta for alternative (even more succinct) solution

Leap year

Determine whether a given year is a leap year in the Gregorian calendar.

```
leap-year?: func [
    {Returns true if the specified year is a leap year; false otherwise.}
    year [date! integer!]
    /local div?
][
    either date? year [year: year/year] [
        if negative? year [throw make error! join [script invalid-arg] year]
    ]
    ; The key numbers are 4, 100, and 400, combined as follows:
    ;   1) If the year is divisible by 4, it’s a leap year.
    ;   2) But, if the year is also divisible by 100, it’s not a leap year.
    ;   3) Double but, if the year is also divisible by 400, it is a leap
year.
    div?: func [n] [zero? year // n]
    to logic! any [all [div? 4  not div? 100] div? 400]
]
```

Mad Libs

Mad Libs is a phrasal template word game where one player prompts another for a list of words to substitute for blanks in a story, usually with funny results.

Write a program to create a Mad Libs like story.

The program should read an arbitrary multiline story from input.

The story will be terminated with a blank line.

Then, find each replacement to be made within the story, ask the user for a word to replace it with, and make all the replacements.

Stop when there are none left and print the final story.

The input should be an arbitrary story in the form:

```
<name> went for a walk in the park. <he or she>
found a <noun>. <name> decided to take it home.
```

Given this example, it should then ask for a name, a he or she and a noun (<name> gets replaced both times with the same value).

```
t: {<name> went for a walk in the park. <he or she> found a <noun>. <name>
decided to take it home.}
view layout [a: area wrap t  btn "Done" [x: a/text unview]]
parse x [any [to "<" copy b thru ">" (append w: [] b)] to end]
foreach i unique w [replace/all x i ask join i ": "]  alert x
```

MD5

Encode a string using an MD5 algorithm. The algorithm can be found on Wikipedia.

Optionally, validate your implementation by running all of the test values in IETF RFC (1321) for MD5.

Additionally, RFC 1321 provides more precise information on the algorithm than the Wikipedia article.

Warning: MD5 has known weaknesses, including **collisions** and forged signatures. Users may consider a stronger alternative when doing production-grade cryptography, such as SHA-256 (from the SHA-2 family), or the upcoming SHA-3.

If the solution on this page is a library solution, see MD5/Implementation for an implementation from scratch.

```
>> checksum/method "The quick brown fox jumped over the lazy dog" 'md5
== #{08A008A01D498C404B0C30852B39D3B8}
```

Menu

Given a prompt and a list containing a number of strings of which one is to be selected, create a function that:

- prints a textual menu formatted as an index value followed by its corresponding string for each item in the list;
- prompts the user to enter a number;
- returns the string corresponding to the selected index number.

The function should reject input that is not an integer or is out of range by redisplaying the whole menu before asking again for a number. The function should return an empty string if called with an empty list.

For test purposes use the following four phrases in a list:

```
fee fie
huff and puff
mirror mirror
tick tock
```

This example is **incorrect**.

Details: The function should return an empty string if called with an empty list. Please also check if this could really used as a function aka subroutine.

```
choices: ["fee fie" "huff and puff" "mirror mirror" "tick tock"]
choice: ""

valid?: func [
        choices [block! list! series!]
        choice
][
        if error? try [choice: to-integer choice] [return false]
        all [0 < choice  choice <= length? choices]
]

while [not valid? choices choice][
        repeat i length? choices [print ["  " i ":" choices/:i]]
        choice: ask "Which is from the three pigs? "
]
print ["You chose:" pick choices to-integer choice]
```

Output:

```
   1 : fee fie
   2 : huff and puff
   3 : mirror mirror
   4 : tick tock
Which is from the three pigs? klf
   1 : fee fie
   2 : huff and puff
   3 : mirror mirror
   4 : tick tock
Which is from the three pigs? 5
   1 : fee fie
   2 : huff and puff
   3 : mirror mirror
   4 : tick tock
Which is from the three pigs? 2
You chose: huff and puff
```

Multiplication tables

Produce a formatted 12×12 multiplication table of the kind memorized by rote when in primary (or elementary) school.

Only print the top half triangle of products.

```
size: 12

; Because of REBOL's GUI focus, it doesn't really do pictured output,
; so I roll my own. See Formatted_Numeric_Output for more
; comprehensive version:
```

```
pad: func [pad n][
    n: to-string n
    insert/dup n " " (pad - length? n)
    n
]
p3: func [v][pad 3 v]  ; A shortcut, I hate to type...

--: has [x][repeat x size + 1 [prin "+---"]  print "+"]  ; Special chars OK.

.row: func [label y /local row x][
        row: reduce ["|" label "|"]
        repeat x size [append row reduce [either x < y ["   "][p3 x * y] "|"]]
        print rejoin row
]

--  .row " x " 1  --  repeat y size [.row  p3 y  y]  --

print rejoin [ crlf  "What about "  size: 5  "?"  crlf ]
--  .row " x " 1  --  repeat y size [.row  p3 y  y]  --

print rejoin [ crlf  "How about "  size: 20  "?"  crlf ]
--  .row " x " 1  --  repeat y size [.row  p3 y  y]  --
```

Output:

(only 12x12 shown):

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+
| x |  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|
+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  1|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|
|  2|   |  4|  6|  8| 10| 12| 14| 16| 18| 20| 22| 24|
|  3|   |   |  9| 12| 15| 18| 21| 24| 27| 30| 33| 36|
|  4|   |   |   | 16| 20| 24| 28| 32| 36| 40| 44| 48|
|  5|   |   |   |   | 25| 30| 35| 40| 45| 50| 55| 60|
|  6|   |   |   |   |   | 36| 42| 48| 54| 60| 66| 72|
|  7|   |   |   |   |   |   | 49| 56| 63| 70| 77| 84|
|  8|   |   |   |   |   |   |   | 64| 72| 80| 88| 96|
|  9|   |   |   |   |   |   |   |   | 81| 90| 99|108|
| 10|   |   |   |   |   |   |   |   |   |100|110|120|
| 11|   |   |   |   |   |   |   |   |   |   |121|132|
| 12|   |   |   |   |   |   |   |   |   |   |   |144|
+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Mutual recursion

Two functions are said to be mutually recursive if the first calls the second, and in turn the second calls the first.

Write two mutually recursive functions that compute members of the Hofstadter Female and Male sequences defined as:

$$F(0) = 1 \quad ; \quad M(0) = 0$$

$$F(n) = n - M(F(n-1)), \; n > 0$$

$$M(n) = n - F(M(n-1)), \; n > 0.$$

(If a language does not allow for a solution using mutually recursive functions then state this rather than give a solution by other means).

```
f: func [
        "Female."
        n [integer!] "Value."
] [either 0 = n [1][n - m f n - 1]]

m: func [
        "Male."
        n [integer!] "Value."
] [either 0 = n [0][n - f m n - 1]]

fs: []  ms: []  for i 0 19 1 [append fs f i  append ms m i]
print ["F:" mold fs  crlf  "M:" mold ms]
```

Output:

```
F: [1 1 2 2 3 3 4 5 5 6 6 7 8 8 9 9 10 11 11 12]
M: [0 0 1 2 2 3 4 4 5 6 6 7 7 8 9 9 10 11 11 12]
```

Number reversal game

Given a jumbled list of the numbers **1** to **9** that are definitely *not* in ascending order.

Show the list, and then ask the player how many digits from the left to reverse.

Reverse those digits, then ask again, until all the digits end up in ascending order.

The score is the count of the reversals needed to attain the ascending order.

Note: Assume the player's input does not need extra validation.

```
print "NUMBER REVERSAL GAME"

tries: 0
goal: [1 2 3 4 5 6 7 8 9]
random/seed now

until [
    jumble: random goal
    jumble != goal ; repeat in the unlikely case that jumble isn't jumbled
]

while [jumble != goal] [
    prin jumble
    prin " How many to flip? "
    flip-index: to-integer input ; no validation!
    reverse/part jumble flip-index
    tries: tries + 1
]

print rejoin ["You took " tries " attempts."]
```

Perfect numbers

Write a function which says whether a number is perfect.

A perfect number is a positive integer that is the sum of its proper positive divisors excluding the number itself.

Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).

Note: The faster Lucas-Lehmer test is used to find primes of the form 2^n-1, all *known* perfect numbers can be derived from these primes using the formula $(2^n - 1) \times 2^{n - 1}$.

It is not known if there are any odd perfect numbers (any that exist are larger than 10^{2000}).

The number of *known* perfect numbers is **50** (as of September, 2018), and the largest known perfect number contains over **46** million decimal digits.

```
perfect?:  func [n [integer!] /local sum] [
    sum: 0
    repeat i (n - 1) [
        if zero? remainder n i [ sum: sum + i ]
    ]
    sum = n
]
```

Primality by trial division

Write a boolean function that tells whether a given integer is prime.

Remember that **1** and all non-positive numbers are not prime.

Use trial division.

Even numbers over two may be eliminated right away.

A loop from **3** to **√ n** will suffice, but other loops are allowed.

```
prime?: func [n] [
    case [
        n = 2 [ true  ]
        n <= 1 or (n // 2 = 0) [ false ]
        true [
          for i 3 round square-root n 2 [ if n // i = 0 [ return false ] ]
          true ]
    ]
]

repeat i 100 [ print [i prime? i]]
```

Queue/Definition

Implement a FIFO queue.

Elements are added at one side and popped from the other in the order of insertion.

Operations:

- push (aka *enqueue*) - add element
- pop (aka *dequeue*) - pop first element
- empty - return truth value when empty

Errors:

- handle the error of trying to pop from an empty queue (behavior depends on the language and platform)

```
; Define fifo class:

fifo: make object! [
    queue: copy []
    push:  func [x][append queue x]
    pop:   func [/local x][   ; Make 'x' local so it won't pollute global
namespace.
        if empty [return none]
        x: first queue  remove queue  x]
    empty: does [empty? queue]
]

; Create and populate a FIFO:

q: make fifo []
q/push 'a
q/push 2
q/push USD$12.34              ; Did I mention that REBOL has 'money!'
datatype?
q/push [Athos Porthos Aramis] ; List elements pushed on one by one.
q/push [[Huey Dewey Lewey]]   ; This list is preserved as a list.

; Dump it out, with narrative:

print rejoin ["Queue is "  either q/empty [""]["not "]  "empty."]
while [not q/empty][print ["  " q/pop]]
print rejoin ["Queue is "  either q/empty [""]["not "]  "empty."]
print ["Trying to pop an empty queue yields:" q/pop]
```

Output:

```
Queue is not empty.
   a
   2
   USD$12.34
   Athos
   Porthos
   Aramis
   Huey Dewey Lewey
Queue is empty.
Trying to pop an empty queue yields: none
```

Quine

A Quine is a self-referential program that can, without any external access, output its own source.

It is named after the philosopher and logician who studied self-reference and quoting in natural language, as for example in the paradox "'Yields falsehood when preceded by its quotation' yields falsehood when preceded by its quotation."

"Source" has one of two meanings. It can refer to the text-based program source. For languages in which program source is represented as a data structure, "source" may refer to the data structure: quines in these languages fall into two categories: programs which print a textual representation of themselves, or expressions which evaluate to a data structure which is equivalent to that expression.

The usual way to code a Quine works similarly to this paradox: The program consists of two identical parts, once as plain code and once *quoted* in some way (for example, as a character string, or a literal data structure). The plain code then accesses the quoted code and prints it out twice, once unquoted and once with the proper quotation marks added. Often, the plain code and the quoted code have to be nested.

Write a program that outputs its own source code in this way. If the language allows it, you may add a variant that accesses the code directly. You are not allowed to read any external files with the source code. The program should also contain some sort

of self-reference, so constant expressions which return their own value which some top-level interpreter will print out. Empty programs producing no output are not allowed.

There are several difficulties that one runs into when writing a quine, mostly dealing with quoting:

- Part of the code usually needs to be stored as a string or structural literal in the language, which needs to be quoted somehow. However, including quotation marks in the string literal itself would be troublesome because it requires them to be escaped, which then necessitates the escaping character (e.g. a backslash) in the string, which itself usually needs to be escaped, and so on.
    - Some languages have a function for getting the "source code representation" of a string (i.e. adds quotation marks, etc.); in these languages, this can be used to circumvent the quoting problem.
    - Another solution is to construct the quote character from its character code, without having to write the quote character itself. Then the character is inserted into the string at the appropriate places. The ASCII code for double-quote is 34, and for single-quote is 39.
- Newlines in the program may have to be reproduced as newlines in the string, which usually requires some kind of escape sequence (e.g. "\n"). This causes the same problem as above, where the escaping character needs to itself be escaped, etc.
    - If the language has a way of getting the "source code representation", it usually handles the escaping of characters, so this is not a problem.
    - Some languages allow you to have a string literal that spans multiple lines, which embeds the newlines into the string without escaping.
    - Write the entire program on one line, for free-form languages (as you can see for some of the solutions here, they run off the edge of the screen), thus

removing the need for newlines. However, this may be unacceptable as some languages require a newline at the end of the file; and otherwise it is still generally good style to have a newline at the end of a file. (The task is not clear on whether a newline is required at the end of the file.) Some languages have a print statement that appends a newline; which solves the newline-at-the-end issue; but others do not.

```
rebol [] q: [print ["rebol [] q:" mold q "do q"]] do q
```

Regular expressions

Task

- match a string against a regular expression
- substitute part of a string using a regular expression

```
string: "This is a string."

; REBOL doesn't use a conventional Perl-compatible regular expression
; syntax. Instead, it uses a variant Parsing Expression Grammar with
; the 'parse' function. It's also not limited to just strings. You can
; define complex grammars that actually parse and execute program
; files.

; Here, I provide a rule to 'parse' that specifies searching through
; the string until "string." is found, then the end of the string. If
; the subject string satisfies the rule, the expression will be true.

if parse string [thru "string." end] [
        print "Subject ends with 'string.'"]

; For replacement, I take advantage of the ability to call arbitrary
; code when a pattern is matched -- everything in the parens will be
; executed when 'to " a "' is satisfied. This marks the current string
; location, then removes the offending word and inserts the replacement.

parse string [
        to " a " ; Jump to target.
        mark: (
                remove/part mark 3 ; Remove target.
                mark: insert mark " another " ; Insert replacement.
        )
        :mark ; Pick up where I left off.
]
```

```
print [crlf "Parse replacement:" string]

; For what it's worth, the above operation is more conveniently done
; with the 'replace' function:

replace string " another " " a " ; Change string back.
print [crlf "Replacement:" string]
```

Output:

```
Subject ends with 'string.'

Parse replacement: This is another string.

Replacement: This is a string.
```

Remove duplicate elements

Given an Array, derive a sequence of elements in which all duplicates are removed.

There are basically three approaches seen here:

- Put the elements into a hash table which does not allow duplicates. The complexity is O(n) on average, and O(n^2) worst case. This approach requires a hash function for your type (which is compatible with equality), either built-in to your language, or provided by the user.
- Sort the elements and remove consecutive duplicate elements. The complexity of the best sorting algorithms is O(n log n). This approach requires that your type be "comparable", i.e., have an ordering. Putting the elements into a self-balancing binary search tree is a special case of sorting.
- Go through the list, and for each element, check the rest of the list to see if it appears again, and discard it if it does. The complexity is O(n^2). The up-shot is that this always works on any type (provided that you can test for equality).

```
print mold unique [1 $23.19 2 elbow 3 2 Bork 4 3 elbow 2 $23.19]
```

Output:

```
[1 $23.19 2 elbow 3 Bork 4]
```

Red

```
>> items: [1 "a" "c" 1 3 4 5 "c" 3 4 5]
>> unique items
== [1 "a" "c" 3 4 5]
```

Rot-13

Implement a **rot-13** function (or procedure, class, subroutine, or other "callable" object as appropriate to your programming environment).

Optionally wrap this function in a utility program (like tr, which acts like a common UNIX utility, performing a line-by-line rot-13 encoding of every line of input contained in each file listed on its command line, or (if no filenames are passed thereon) acting as a filter on its "standard input."

(A number of UNIX scripting languages and utilities, such as *awk* and *sed* either default to processing files in this way or have command line switches or modules to easily implement these wrapper semantics, e.g., Perl and Python).

The **rot-13** encoding is commonly known from the early days of Usenet "Netnews" as a way of obfuscating text to prevent casual reading of spoiler or potentially offensive material.

Many news reader and mail user agent programs have built-in **rot-13** encoder/decoders or have the ability to feed a message through any external utility script for performing this (or other) actions.

The definition of the rot-13 function is to simply replace every letter of the ASCII alphabet with the letter which is "rotated" 13 characters "around" the 26 letter alphabet from its normal cardinal position (wrapping around from **z** to **a** as necessary).

Thus the letters **abc** become **nop** and so on.

Technically **rot-13** is a "mono-alphabetic substitution cipher" with a trivial "key".

A proper implementation should work on upper and lower case letters, preserve case, and pass all non-alphabetic characters in the input stream through without alteration.

```
; Test data has upper and lower case characters as well as characters
; that should not be transformed, like numbers, spaces and symbols.

text: "This is a 28-character test!"

print "Using cipher table:"

; I build a set of correspondence lists here. 'x' is the letters from
; A-Z, in both upper and lowercase form. Note that REBOL can iterate
; directly over the alphabetic character sequence in the for loop. 'y'
; is the cipher form, 'x' rotated by 26 characters (remember, I have
; the lower and uppercase forms together). 'r' holds the final result,
; built as I iterate across the 'text' string. I search for the
; current character in the plaintext list ('x'), if I find it, I get
; the corresponding character from the ciphertext list
; ('y'). Otherwise, I pass the character through untransformed, then
; return the final string.

rot-13: func [
        "Encrypt or decrypt rot-13 with tables."
        text [string!] "Text to en/decrypt."
        /local x y r i c
] [
        x: copy ""  for i #"a" #"z" 1 [append x rejoin [i  uppercase i]]
        y: rejoin [copy skip x 26  copy/part x 26]
        r: copy ""

        repeat i text [append r  either c: find/case x i [y/(index? c)][i]]
        r
]

; Note that I am setting the 'text' variable to the result of rot-13
; so I can reuse it again on the next call. The rot-13 algorithm is
; reversible, so I can just run it again without modification to decrypt.

print ["    Encrypted:"  text: rot-13 text]
print ["    Decrypted:"  text: rot-13 text]


print "Using parse:"
```

```
clamp: func [
        "Contain a value within two enclosing values. Wraps if necessary."
        x v y
][
        x: to-integer x  v: to-integer v  y: to-integer y
        case [v < x [y - v] v > y [v - y + x - 1] true v]
]

; I'm using REBOL's 'parse' word here. I set up character sets for
; upper and lower-case letters, then let parse walk across the
; text. It looks for matches to upper-case letters, then lower-case,
; then skips to the next one if it can't find either. If a matching
; character is found, it's mathematically incremented by 13 and
; clamped to the appropriate character range. parse changes the
; character in place in the string, hence this is a destructive
; operation.

rot-13: func [
        "Encrypt or decrypt rot-13 with parse."
        text [string!] "Text to en/decrypt. Note: Destructive!"
] [
        u: charset [#"A" - #"Z"]
        l: charset [#"a" - #"z"]

        parse text [some [
              i:                                              ; Current position.
              u (i/1: to-char clamp #"A" i/1 + 13 #"Z") |     ; Upper case.
              l (i/1: to-char clamp #"a" i/1 + 13 #"z") |     ; Lower case.
              skip]]                                          ; Ignore others.
        text
]

; As you see, I don't need to re-assign 'text' anymore.

print ["    Encrypted:" rot-13 text]
print ["    Decrypted:" rot-13 text]
```

Output:

```
Using cipher table:
    Encrypted: Guvf vf n 28-punenpgre grfg!
    Decrypted: This is a 28-character test!
Using parse:
    Encrypted: Guvf vf n 28-punenpgre grfg!
    Decrypted: This is a 28-character test!
```

Runtime evaluation/In an environment

Given a program in the language (as a string or AST) with a free variable named *x* (or another name if that is not valid syntax), evaluate it with *x* bound to a provided value, then evaluate it again with *x* bound to another provided value, then subtract the result of the first from the second and return or print it.

Do so in a way which:

- does not involve string manipulation of the input source code
- is plausibly extensible to a runtime-chosen set of bindings rather than just *x*
- does not make *x* a *global* variable

or note that these are impossible.

```
prog: [x * 2]
fn: func [x] [do bind prog 'x]
a: fn 2
b: fn 4
subtract b a
```

Result:

```
4
```

Simple database

Write a simple tool to track a small set of data.

The tool should have a command-line interface to enter at least two different values.

The entered data should be stored in a structured format and saved to disk.

It does not matter what kind of data is being tracked. It could be a collection (CDs, coins, baseball cards, books), a diary, an

electronic organizer (birthdays/anniversaries/phone numbers/addresses), etc.

You should track the following details:

- A description of the item. (e.g., title, name)
- A category or tag (genre, topic, relationship such as “friend” or “family”)
- A date (either the date when the entry was made or some other date that is meaningful, like the birthday); the date may be generated or entered manually
- Other optional fields

The command should support the following Command-line arguments to run:

- Add a new entry
- Print the latest entry
- Print the latest entry for each category
- Print all entries sorted by a date

The category may be realized as a tag or as structure (by making all entries in that category subitems)

The file format on disk should be human readable, but it need not be standardized. A natively available format that doesn't need an external library is preferred. Avoid developing your own format if you can use an already existing one. If there is no existing format available, pick one of:

- JSON
- S-Expressions
- YAML
- others

```
write/append %rdb ""  db: load %rdb
switch system/options/args/1 [
    "new" [write/append %rdb rejoin [now " " mold/only next
system/options/args newline]]
```

```
    "latest" [print copy/part tail sort/skip db 4 -4]
    "latestcat" [
        foreach cat unique extract at db 3 4 [
            t: copy []
            foreach [a b c d] db [if c = cat [append t reduce [a b c d]]]
            print copy/part tail sort/skip t 4 -4
        ]
    ]
    "sort" [probe sort/skip db 4]
]
halt
```

Simple windowed application

Create a window that has:

1. a label that says "There have been no clicks yet"
2. a button that says "click me"

Upon clicking the button with the mouse, the label should change and show the number of times the button has been clicked.

```
clicks: 0

; Simple GUI's in REBOL can be defined with 'layout', a
; special-purpose language (dialect, in REBOL-speak) for specifying
; interfaces. In the example below, I describe a gradient background
; with a text label and a button. The block in the button section
; details what should happen when it's clicked on -- increment the
; number of clicks and update the label text.

; The 'view' function paints the layout on the screen and listens for
; events.

view layout [
        backdrop effect [gradient 0x1 black coal]

        label: vtext "There have been no clicks yet."

        button maroon "click me" [
                clicks: clicks + 1
                set-face label reform ["clicks:" clicks]
        ]
]
```

Sleep

Write a program that does the following in this order:

- Input an amount of time to sleep in whatever units are most natural for your language (milliseconds, seconds, ticks, etc.). This unit should be noted in comments or in a description.
- Print "Sleeping..."
- Sleep the main thread for the given amount of time.
- Print "Awake!"
- End.

```
naptime: to-integer ask "Please enter sleep time in seconds: "
print "Sleeping..."
wait naptime
print "Awake!"
```

Red

```
str-time: to integer! ask "Enter wait time "    ;get user input , convert to integer
print "waiting"
wait str-time       ;Seconds
print "awake"
```

Sort stability

When sorting records in a table by a particular column or field, a stable sort will always retain the relative order of records that have the same key.

For example, in this table of countries and cities, a stable sort on the **second** column, the cities, would keep the US Birmingham above the UK Birmingham. (Although an unstable sort *might*, in this case, place the US Birmingham above the UK Birmingham, a stable sort routine would *guarantee* it).

```
UK  London
US  New York
US  Birmingham
UK  Birmingham
```

Similarly, stable sorting on just the first column would generate “UK London” as the first item and “US Birmingham” as the last item (since the order of the elements having the same first word – “UK” or “US” – would be maintained).

1. Examine the documentation on any in-built sort routines supplied by a language.
2. Indicate if an in-built routine is supplied
3. If supplied, indicate whether or not the in-built routine is stable.

```
; REBOL's sort function is not stable by default. You need to use a custom comparator to make it so.

blk: [
    [UK London]
    [US New-York]
    [US Birmingham]
    [UK Birmingham]
]
sort/compare blk func [a b] [either a/2 < b/2 [-1] [either a/2 > b/2 [1] [0]]]

; Note that you can also do a stable sort without nested blocks.
blk: [
    UK London
    US New-York
    US Birmingham
    UK Birmingham
]
sort/skip/compare blk 2 func [a b] [either a < b [-1] [either a > b [1] [0]]]
```

Sorting algorithms/Insertion sort

An $O(n^2)$ sorting algorithm which moves elements one at a time into the correct position. The algorithm consists of inserting one element at a time into the previously sorted part of the array, moving higher ranked elements up as necessary. To start off, the first (or smallest, or any arbitrary) element of the unsorted array is considered to be the sorted part.

Although insertion sort is an $O(n^2)$ algorithm, its simplicity, low overhead, good locality of reference and efficiency make it a

good choice in two cases:
(i) small n,
(ii) as the final finishing-off algorithm for O(n log$n$) algorithms such as mergesort and quicksort.

The algorithm is as follows (from wikipedia):

```
function insertionSort(array A)
    for i from 1 to length[A]-1 do
        value := A[i]
        j := i-1
        while j >= 0 and A[j] > value do
            A[j+1] := A[j]
            j := j-1
        done
        A[j+1] = value
    done
```

Writing the algorithm for integers will suffice.

```
; This program works with REBOL version R2 and R3, to make it work with Red
; change the word func to function
insertion-sort: func [
        a [block!]
        /local i [integer!] j [integer!] n [integer!]
        value [integer! string! date!]
][
        i: 2
        n: length? a

        while [i <= n][
                value: a/:i
                j: i
                while [ all [   1 < j
                                value < a/(j - 1) ]][

                        a/:j: a/(j - 1)
                        j: j - 1
                ]
                a/:j: value
                i: i + 1
        ]
        a
]

probe insertion-sort [4 2 1 6 9 3 8 7]

probe insertion-sort [ "---Monday's Child Is Fair of Face (by Mother
Goose)---"
  "Monday's child is fair of face;"
  "Tuesday's child is full of grace;"
```

```
  "Wednesday's child is full of woe;"
  "Thursday's child has far to go;"
  "Friday's child is loving and giving;"
  "Saturday's child works hard for a living;"
  "But the child that is born on the Sabbath day"
  "Is blithe and bonny, good and gay."]

; just by adding the date! type to the local variable value the same function
can sort dates.
probe insertion-sort [12-Jan-2015 11-Jan-2015 11-Jan-2016 12-Jan-2014]
```

Output:

```
[1 2 3 4 6 7 8 9]
[{---Monday's Child Is Fair of Face (by Mother Goose)---}
    "But the child that is born on the Sabbath day"
    "Friday's child is loving and giving;"
    "Is blithe and bonny, good and gay."
    "Monday's child is fair of face;"
    "Saturday's child works hard for a living;"
    "Thursday's child has far to go;"
    "Tuesday's child is full of grace;"
    "Wednesday's child is full of woe;"
]
[12-Jan-2014 11-Jan-2015 12-Jan-2015 11-Jan-2016]
```

Sorting algorithms/Merge sort

The **merge sort** is a recursive sort of order n*log(n).

It is notable for having a worst case and average complexity of *O(n*log(n))*, and a best case complexity of *O(n)* (for pre-sorted input).

The basic idea is to split the collection into smaller groups by halving it until the groups only have one element or no elements (which are both entirely sorted groups).

Then merge the groups back together so that their elements are in order.

This is how the algorithm gets its *divide and conquer* description.

Write a function to sort a collection of integers using the merge sort.

The merge sort algorithm comes in two parts:

a sort function and
a merge function

The functions in pseudocode look like this:

```
function mergesort(m)
   var list left, right, result
   if length(m) ≤ 1
       return m
   else
       var middle = length(m) / 2
       for each x in m up to middle - 1
           add x to left
       for each x in m at and after middle
           add x to right
       left = mergesort(left)
       right = mergesort(right)
       if last(left) ≤ first(right)
          append right to left
          return left
       result = merge(left, right)
       return result

function merge(left,right)
   var list result
   while length(left) > 0 and length(right) > 0
       if first(left) ≤ first(right)
           append first(left) to result
           left = rest(left)
       else
           append first(right) to result
           right = rest(right)
   if length(left) > 0
       append rest(left) to result
   if length(right) > 0
       append rest(right) to result
   return result
```

Note: better performance can be expected if, rather than recursing until length(m) ≤ 1, an insertion sort is used for length(m) smaller than some threshold larger than **1**.

However, this complicates the example code, so it is not shown here.

```
msort: function [a compare] [msort-do merge] [
    if (length? a) < 2 [return a]
    ; define a recursive Msort-do function
    msort-do: function [a b l] [mid] [
        either l < 4 [
            if l = 3 [msort-do next b next a 2]
            merge a b 1 next b l - 1
        ] [
            mid: make integer! l / 2
            msort-do b a mid
            msort-do skip b mid skip a mid l - mid
            merge a b mid skip b mid l - mid
        ]
    ]
    ; function Merge is the key part of the algorithm
    merge: func [a b lb c lc] [
        until [
            either (compare first b first c) [
                change/only a first b
                b: next b
                a: next a
                zero? lb: lb - 1
            ] [
                change/only a first c
                c: next c
                a: next a
                zero? lc: lc - 1
            ]
        ]
        loop lb [
            change/only a first b
            b: next b
            a: next a
        ]
        loop lc [
            change/only a first c
            c: next c
            a: next a
        ]
    ]
    msort-do a copy a length? a
    a
]
```

Stack

A **stack** is a container of elements with last in, first out access policy. Sometimes it also called **LIFO**.

The stack is accessed through its **top**.

The basic stack operations are:

- *push* stores a new element onto the stack top;
- *pop* returns the last pushed stack element, while removing it from the stack;
- *empty* tests if the stack contains no elements.

Sometimes the last pushed stack element is made accessible for immutable access (for read) or mutable access (for write):

- *top* (sometimes called *peek* to keep with the *p* theme) returns the topmost element without modifying the stack.

Stacks allow a very simple hardware implementation.

They are common in almost all processors.

In programming, stacks are also very popular for their way (**LIFO**) of resource management, usually memory.

Nested scopes of language objects are naturally implemented by a stack (sometimes by multiple stacks).

This is a classical way to implement local variables of a re-entrant or recursive subprogram. Stacks are also used to describe a formal computational framework.

See stack machine.

Many algorithms in pattern matching, compiler construction (e.g. recursive descent parsers), and machine learning (e.g. based on tree traversal) have a natural representation in terms of stacks.

Create a stack supporting the basic operations: push, pop, empty.

```
stack: make object! [
       data: copy []
```

```
        push: func [x][append data x]
        pop: func [/local x][x: last data  remove back tail data  x]
        empty: does [empty? data]

        peek: does [last data]
]

; Teeny Tiny Test Suite

assert: func [code][print [either do code ["  ok"]["FAIL"]  mold code]]

print "Simple integers:"
s: make stack []  s/push 1  s/push 2 ; Initialize.

assert [2 = s/peek]
assert [2 = s/pop]
assert [1 = s/pop]
assert [s/empty]

print [lf "Symbolic data on stack:"]
v: make stack [data: [this is a test]] ; Initialize on instance.

assert ['test = v/peek]
assert ['test = v/pop]
assert ['a = v/pop]
assert [not v/empty]
```

Sample run:

```
Simple integers:
  ok [2 = s/peek]
  ok [2 = s/pop]
  ok [1 = s/pop]
  ok [s/empty]

Symbolic data on stack:
  ok ['test = v/peek]
  ok ['test = v/pop]
  ok ['a = v/pop]
  ok [not v/empty]
```

Stair-climbing puzzle

From Chung-Chieh Shan (LtU):

Your stair-climbing robot has a very simple low-level API: the "step" function takes no argument and attempts to climb one step as a side effect. Unfortunately, sometimes the attempt fails and

the robot clumsily falls one step instead. The "step" function detects what happens and returns a boolean flag: true on success, false on failure.

Write a function "step_up" that climbs one step up [from the initial position] (by repeating "step" attempts if necessary). Assume that the robot is not already at the top of the stairs, and neither does it ever reach the bottom of the stairs. How small can you make "step_up"? Can you avoid using variables (even immutable ones) and numbers?

Here's a pseudo-code of a simple recursive solution without using variables:

```
func step_up()
{
    if not step() {
        step_up();
        step_up();
    }
}
```

Inductive proof that step_up() steps up one step, if it terminates:

- Base case (if the step() call returns true): it stepped up one step. QED
- Inductive case (if the step() call returns false): Assume that recursive calls to step_up() step up one step. It stepped down one step (because step() returned false), but now we step up two steps using two step_up() calls. QED

The second (tail) recursion above can be turned into an iteration, as follows:

```
func step_up()
{
    while not step() {
        step_up();
    }
}
```

random/seed now

```
step: does [random/only reduce [yes no]]

; Iterative solution with symbol stack. No numbers, draws a nifty
; diagram of number of steps to go. This is intended more to
; demonstrate a correct solution:

step_up: func [/steps s] [
        either not steps [
                print "Starting up..."
                step_up/steps copy [|]
        ][
                while [not empty? s][
                        print ["    Steps left:" s]
                        either step [remove s][append s '|]
                ]
        ]
]

step_up  print ["Success!" crlf]

; Recursive solution. No numbers, no variables. "R" means a recover
; step, "+" means a step up.

step_upr: does [if not step [prin "R " step_upr  prin "+ " step_upr]]

step_upr  print ["Success!" crlf]

; Small recursive solution, no monitoring:

step_upt: does [if not step [step_upt step_upt]]

step_upt  print "Success!"
```

Output:

```
Starting up...
    Steps left: |
    Steps left: | |
    Steps left: |
Success!

R R + R R R R R R + + R + + + + + + R + R R + R + R + R + + + Success!

Success!
```

String interpolation (included)

Given a string and defined variables or values, string interpolation is the replacement of defined character sequences in the string by values or variable values.

> For example, given an original string of "Mary had a X lamb.", a value of "big", and if the language replaces X in its interpolation routine, then the result of its interpolation would be the string "Mary had a big lamb".
>
> (Languages usually include an infrequently used character or sequence of characters to indicate what is to be replaced such as "%", or "#" rather than "X").

The task is to

1. Use your languages inbuilt string interpolation abilities to interpolate a string missing the text "little" which is held in a variable, to produce the output string "Mary had a little lamb".
2. If possible, give links to further documentation on your languages string interpolation features.

Note: The task is not to create a string interpolation routine, but to show a language's built-in capability.

```
str: "Mary had a <%size%> lamb"
size: "little"
build-markup str

;REBOL3 also has the REWORD function
str: "Mary had a $size lamb"
reword str [size "little"]
```

Sum and product of an array

Compute the sum and product of an array of integers.

```
; Simple:

sum: func [a [block!] /local x] [x: 0  forall a [x: x + a/1]  x]

product: func [a [block!] /local x] [x: 1  forall a [x: x * a/1]  x]
```

```
; Way too fancy:

redux: func [
        "Applies an operation across an array to produce a reduced value."
        a [block!] "Array to operate on."
        op [word!] "Operation to perform."
        /init x    "Initial value (default 0)."
][if not init [x: 0]  forall a [x: do compose [x (op) (a/1)]]  x]

rsum: func [a [block!]][redux a '+]

rproduct: func [a [block!]][redux/init a '* 1]

; Tests:

assert: func [code][print [either do code ["  ok"]["FAIL"]  mold code]]

print "Simple dedicated functions:"
assert [55      = sum [1 2 3 4 5 6 7 8 9 10]]
assert [3628800 = product [1 2 3 4 5 6 7 8 9 10]]

print [crlf "Fancy reducing function:"]
assert [55      = rsum [1 2 3 4 5 6 7 8 9 10]]
assert [3628800 = rproduct [1 2 3 4 5 6 7 8 9 10]]
```

Tokenize a string

Separate the string "Hello,How,Are,You,Today" by commas into an array (or list) so that each element of it stores a different word. Display the words to the 'user', in the simplest manner possible, separated by a period. To simplify, you may display a trailing period.

```
print ["Original:"  original: "Hello,How,Are,You,Today"]
tokens: parse original ","
dotted: ""  repeat i tokens [append dotted rejoin [i "."]]
print ["Dotted:  "  dotted]
```

Output:

```
 Original: Hello,How,Are,You,Today
 Dotted:   Hello.How.Are.You.Today.
```

Red

```
str: "Hello,How,Are,You,Today"
```

```
>> tokens: split str ","
>> probe tokens
["Hello" "How" "Are" "You" "Today"]

>> periods: replace/all form tokens " " "."        ;The word FORM converts the
list series to a string removing quotes.
>> print periods                                                ;then REPLACE/ALL
spaces with period
Hello.How.Are.You.Today
```

Towers of Hanoi

Solve the [Towers of Hanoi] problem with recursion.

```
hanoi: func [
        {Begin moving the golden disks from one pole to the next.
         Note: when last disk moved, the world will end.}
        disks [integer!] "Number of discs on starting pole."
        /poles "Name poles."
        from to via
][
    if disks = 0 [return]
        if not poles [from: 'left  to: 'middle  via: 'right]

    hanoi/poles disks - 1 from via to
        print [from "->" to]
    hanoi/poles disks - 1 via to from
]

hanoi 4
```

Output:

```
left -> right
left -> middle
right -> middle
left -> right
middle -> left
middle -> right
left -> right
left -> middle
right -> middle
right -> left
middle -> left
right -> middle
left -> right
left -> middle
right -> middle
```

Web scraping

Create a program that downloads the time from this URL: http://tycho.usno.navy.mil/cgi-bin/timer.pl and then prints the current UTC time by extracting just the UTC time from the web page's HTML.

If possible, only use libraries that come at no *extra* monetary cost with the programming language and that are widely available and popular such as CPAN for Perl or Boost for C++

```
; Notice that REBOL understands unquoted URL's:

service: http://tycho.usno.navy.mil/cgi-bin/timer.pl

; The 'read' function can read from any data scheme that REBOL knows
; about, which includes web URLs. NOTE: Depending on your security
; settings, REBOL may ask you for permission to contact the service.

html: read service

; I parse the HTML to find the first <br> (note the unquoted HTML tag
; -- REBOL understands those too), then copy the current time from
; there to the "UTC" terminator.

; I have the "to end" in the parse rule so the parse will succeed.
; Not strictly necessary once I've got the time, but good practice.

parse html [thru <br> copy current thru "UTC" to end]

print ["Current UTC time:" current]
```

XML/Input

Given the following XML fragment, extract the list of *student names* using whatever means desired. If the only viable method is to use XPath, refer the reader to the task XML and XPath.

```
xml: {
<Students>
  <Student Name="April" Gender="F" DateOfBirth="1989-01-02" />
  <Student Name="Bob" Gender="M"  DateOfBirth="1990-03-04" />
  <Student Name="Chad" Gender="M"  DateOfBirth="1991-05-06" />
  <Student Name="Dave" Gender="M"  DateOfBirth="1992-07-08">
    <Pet Type="dog" Name="Rover" />
  </Student>
  <Student DateOfBirth="1993-09-10" Gender="F" Name="&#x00C9;mily" />
</Students>
}

; REBOL has a simple built-in XML parser. It's not terribly fancy, but
; it's easy to use. It converts the XML into a nested list of blocks
; which can be accessed using standard REBOL path operators. The only
; annoying part (in this case) is that it does try to preserve
; whitespace, so some of the parsed elements are just things like line
; endings and whatnot, which I need to ignore.

; Once I have drilled down to the individual student records, I can
; just use the standard REBOL 'select' to locate the requested
; property.

data: parse-xml xml
students: data/3/1/3 ; Drill down to student records.
foreach student students [
        if block! = type? student [ ; Ignore whitespace elements.
                print select student/2 "Name"
        ]
]
```

Output:

```
April
Bob
Chad
Dave
Émily
```

Y combinator

In strict functional programming and the lambda calculus, functions (lambda expressions) don't have state and are only allowed to refer to arguments of enclosing functions. This rules out the usual definition of a recursive function wherein a function is associated with the state of a variable and this variable's state is used in the body of the function.

The Y combinator is itself a stateless function that, when applied to another stateless function, returns a recursive version of the function. The Y combinator is the simplest of the class of such functions, called fixed-point combinators.

Define the stateless Y combinator and use it to compute factorials and Fibonacci numbers from other stateless functions or lambda expressions.

```
Y: closure [g] [do func [f] [f :f] closure [f] [g func [x] [do f :f :x]]]
```

usage example

```
fact*: closure [h] [func [n] [either n <= 1 [1] [n * h n - 1]]]
fact: Y :fact*
```