# Listack documentation

version 0.37

**Listack** is an experiment in making a symmetric, stackless, stack-based, concatenative language. Listack was inspired by Factor and Falsish , which is a variant of False inspired by fish ><>.

Listack is *symmetric* in that most command words are available in a prefix, infix, and postfix form. The user can choose which forms to use, and can thus mimic Lisp (prefix), Forth (postfix), or use a mix of all three forms in the style of most imperative languages. The prefix and infix forms are created from the base postfix form by meta-programming, which is fully available to the user.

Listack is *stackless* in that the implementation is very nearly a Turing machine, with a stack for past data, the current command, and then a queue for future commands. Commands are read from the front of the queue, and the data computed by these commands is pushed onto the stack, creating, in effect, an infinite tape. As such, the language is implemented as a simple loop with no recursion and no return stack. Calls to global functions ("words") merely place the function definition on the front of the command queue. Loops are implemented by repeatedly pushing the body of the loop back onto the front of the command queue.

Listack is *stack-based*. Most commands operate directly upon the data stack, taking their arguments directly from it and leaving rtheir esults on the top of the stack. Local and global variable references copy their values to the stack. There are auxiliary side stacks (a..z) and local variables (A..Z) to facilitate programming.

*Concatenative* languages, which are generally stack-based, are similar to many functional languages in that function composition is accomplished by typing one command after another. Functions and entire programs are created as a list of words, numbers, strings, and sequences. The output from one word is the input to the next word via the data stack, much like the unix pipe ("|") command. For example:

```
1 2 .+ * 3  4 ./% enlist: 2 → [2 1]
# 1 plus 2 = 3;  3 times 3 = 9;  9 divmod 4 = 2 remainder 1;
# top 2 items on the stack (2, 1) are converted to a list: [2 1]
```

Listack is a pun on List and Stack based programming. It was created by McChuck and implemented in Python 3.10 in January 2023.

## Features

Commands are called words in the concatenative tradition. Every instruction is a word, and every word is an instruction. Most Listack words are available in prefix, infix, and postfix variants. By convention, the prefix variant ends with ":" and the postfix begins with ".". Commands are implemented as postfix (or immediate), with the variants constructed using metaprogramming. After all, the only difference between these commands, other than their tokens, is the order in which the operands appear.

- Stack effects are shown as follows.
    - +: 1 2 → 3
    - 8 / 2 → 4
    - 3 2 .* → 6
- The data types are INT, FLOAT, STR, LIST, BLOCK, BOOL, and WORD
- Commas are treated as white space.
- Floating point numbers contain a period.
- Words beginning with a period must be separated from numbers by white space.
- Numbers are pushed onto the stack.
- Strings are pushed onto the stack. Strings can be converted into lists and vice-versa.
- Sequences (lists and blocks) are pushed onto the stack. There is little practical difference between a list and a block in the language implementation. As the saying goes, "It's all just data."
- Blocks are executed when called, while Lists are pushed to the stack.
- Lists "[ ]" contain data (which can include words).
- Blocks "{ }" contain words (which can include data), and serve as lambda function expressions. The traditional concatenative programming term "quotation" is eschewed in favor of "block", as in code block, thus the curly braces.
- Words execute their function definitions.
- Both local and global variables are available.
- Local variables A..Z are always available and initialized to "".
- Global "side" stacks a..z are always available using push_, pop_, copy_, and depth_.
- Local variables can be created using "init".
- Local variables place their contents onto the data stack.
- Global variables/functions can be created using "def".
- Global functions place their definition at the front of the command queue.
- Variables/functions can examined and altered with "get" and "set", and their contents can be executed with "call".
- Variables are referenced by name and act as words.
- Variable names cannot begin with a number or reserved character.
- "." can only appear at the beginning of a word. (Floats, of course, contain a period by definition.)
- “:" can only appear at the end of a word.
- @name is shorthand for “name”.get, $name is shorthand for “name”.set and !name is shorthand for “name”.call.

- Local variable scope is administered manually using "(" and ")". They must be balanced.
- "n (" removes n items from the top of the data stack, creates a local scope, initializes local variables A..Z, sets the values of A..N to the items taken from the stack, creates a new data stack (saving the old one for later use), and pushes the n items onto the new stack.
- ")" copies all remaining items on the current stack onto the old stack, then clears and deletes the current data stack and local variable set.
- Yes, this means there is a stack of stacks. Yes, you can manually save and restore the data stack.
- The data stack is implemented as a deque (double ended queue). It is reversible and rotatable.
- Control flow, other than the execution of words one by one, is accomplished with a series of more and more complex functions based on "if" and meta-programming. ".if" could be defined as {.choose .exec}.
- The trinary "starship" selector <=> is available for acting upon numbers based on their sign.
- *if:* and *if* may behave differently from *.if*, because they have not had their arguments evaluated before they are executed. Remember that they act upon the following items *as they are*. When in doubt, use code blocks instead of bare words.
- "#" begins a comment (except in meta programming). Comments extend to end of line.
- Due to the limitations of the parser, a backslash ("\") cannot be the final character in a quoted string. Add a space after it, before the closing quote mark.
- Not every possible function has been implemented, as the language is intended as an experiment and proof of concept.
- Listack is by no means efficient. It is parsed and interpreted using a few thousand lines of Python 3.10 code.
- Invoke a Listack program as: *python listack.py name.ls[p] [debug] [verbose]*
- When run, the *name.ls* program file will be parsed and a formatted *name.lsp* file created. This is a pickled version of the nested deque data structure.
- .lsp files can be run to avoid the parsing step, speeding up code execution.
- You can *load* auxiliary files in your program. The loaded file's code is immediately run.
- The debug option will run through the program code one word at a time.
- The verbose option will show extra information about the internal state of the program.
- When combined, these two options will slow program execution to a crawl, but make debugging much easier.
- Most errors will terminate execution with a readable error message. Unfortunately, Listack does not internally represent line numbers to help reference back to the original code.
- Errors detected during parsing will show a line number.
- You can manually use *err_msg* to create your own error messages at check points.

# List of words

Postfix words begin with a period. Prefix words end with a colon. Infix and immediate words don't have punctuation.

## Stack manipulation

- **drop** a b c d drop → a b c
  Deletes the item on the top of stack (TOS).
- **dup** a b c d dup → a b c d d
  Duplicate TOS.
- **swap** a b c d swap → a b d c
  Swap top two items.
- **roll** a b c d roll → a c d b
  Rotate top three items.
- **over** a b c d over → a b c d c
  Copy the second item from top.
- **reverse** a b c d e reverse → e d c b a
  Reverse entire list.
- **rot_r**:, **.rot_r** a b c d e 2 .rot_r → d e a b c
  Rotate entire stack right n places.
- **rot_l:**, **.rot_l** a b c d e rot_l: 2 → c d e a b
  Rotate entire stack left n places.
- **depth** a b c d depth → a b c d 4
  Puts the number of stack items on TOS.
- **save_stack**
  Save the stack to memory.
- **restore_stack**
  Restore the last saved stack. Overwrites current stack.
- **push_n** a b c d 21 push_a → a b c
  Pushes TOS onto side stack n, where n is a letter a..z.
- **copy_n** a b c d copy_a → a b c d 21
  Copies TOS of side stack n (a..z) to TOS.
- **pop_n** a b c d pop_a → a b c d 21
  Copies and deletes the top item of side stack a..z and puts it on TOS.
- **depth_n** a b c d depth_a → a b c d 0
  Puts the number of item sin side stack n (a..z) on TOS
- ` `word
  Backquote. Shorthand for {word}.
- \ \ word
  Ignore the following word and push it to the stack for later use.
  "\" is prepended to "word", so it works correctly with pre- and in-fix functions.
- **clear**
  Clears the data stack , removing everything.

## Math functions

- Floating point numbers are rounded off at the 15th decimal place. This generally improves performance, as it eliminates a lot of *9.0 / 3.0 = 2.9999999999999999* errors.
- **+:**, **+**, **.+** a b .+ ::: 1 2 .+ → 3
- **-:**, **-**, **.-** a b .- ::: 2 - 1 → 1
- ***:**, *****, **.*** a b .* ::: *: 3 2 → 6
- **/:**, **/**, **./** a b ./ ::: 5 / 2 → 2.5
- **//:**, **//**, **.//** a b .// ::: 5 2 .// → 2
  Integer division.
- **%:**, **%**, **.%**, **mod:**, **mod**, **.mod** a b .% ::: 5 % 2 → 1
  Modulus (remainder).
- **/%:**, **/%**, **./%**, **divmod:**, **divmod**, **.divmod** a b ./% ::: 5 /% 2 → 2 1
  Integer division with remainder.
- **pow:**, **pow**, **.pow** a b .pow ::: 2 pow 3 → 8
  Raise a to power b.
- **root:**, **root**, **.root** a b .root ::: root: 8 3 → 2
  Raise a to power 1/b, AKA bth root of a.
- **sqrt:**, **.sqrt** a .sqrt ::: 4 .sqrt → 2
  Square root.
- **sqr:**, **.sqr** a .sqr ::: sqr: 3 → 9
  Square.
- **log:**, **log**, **.log** a b .log ::: 100 10 .log → 2
  Logarithm of a with base b.
- **ln:**, **.ln** a .ln ::: 10.ln → 2.302
  Natural log of a.
- **exp:**, **.exp** a .exp ::: exp:2.302 → 10
  e to the power a, reverse of natural log.
- **sin:**, **.sin** sin: 0→ 0
  Sine from radians.
- **cos:**, **.cos** pi .cos → 1
  Cosine from radians.
- **tan:**, **.tan** 0.785 .tan → 1
  Tangent from radians.
- **deg>rad**:, **.deg>rad** a .deg>rad ::: 90 .deg>rad → 1.570
  Convert degrees to radians.
- **rad>deg:**, **.rad>deg** a .rad>deg ::: rad>deg: 1.570 → 90
  Convert radians to degrees.
- **pi** pi → 3.141592653589793
- **bit_not**, **.bit_not** int .bit_not # bitwise not
- **bit_and:**, **bit_and**, **.bit_and** int bit_and int # bitwise and
- **bit_or:**, **bit_or**, **.bit_or** bit_or: int int # bitwise or
- **bit_xor:**, **bit_xor**, **.bit_xor** int bit_xor int # bitwise xor
- **bit_r:**, **bit_r**, **.bit_r** int how_many .bit_r # bitwise right shift (negative = left)
- **bit_l:**, **bit_l**, **.bit_l** bit_l: int how_many # bitwise left shift (negative = right)

### Boolean logic

- **False**, **false**, **FALSE**  
  Boolean *False*
- **True**, **true**, **TRUE**  
  Boolean *True*
- **make_bool:**, **.make_bool**     “”.make_bool → False     make_bool: 0 → True  
  Returns *False* for False/false/FALSE or empty string/block/list, otherwise *True*
- **<:**, **<**, **.<**     a b .<     :::     1 < 2 → True  
  Is a less than b?
- **>:**, **>**, **.>**     a b .>     :::     >: 1 2 → False  
  Is a greater than b?
- **<=:**, **<=**, **.<=**, **~>:**, **~>**, **.~>**     a b .<=  
  Is a less than or equal to (not greater than) b?
- **>=:**, **>=**, **.>=**, **~<:**, **~<**, **.~<**     a ~< b  
  Is a greater than or equal to (not less than) b?
- **=:**, **=**, **.=**     0 True .= → True  
  Is a equal to b? Forgiving of mixed types (int/float, str/word, list/block)  
  If only one of the arguments is a Boolean, runs *make_bool* on the other.
- **~=:**, **~=**, **.~=**     [] ~= True → True  
  Is a not equal to b? Forgiving of mixed types.  
  If only one of the arguments is a Boolean, runs *make_bool* on the other.
- **==:**, **==**, **.==**     [] == {} → False  
  Is a equal to b when they are the same type? False if different types.
- **~==:**, **~==**, **.~==**     0 ~== False → True  
  Is a not equal to b when they are the same type? True if different types.
- **not:**, **.not**     a .not     :::     not: True → False  
  Logical not.
- **and:**, **and**, **.and**     a b .and     :::     True and False → False  
  Logical and.
- **or:**, **or**, **.or**     a b .or     :::     True False .or → True  
  Logical or.
- **xor:**, **xor**, **.xor**     a b .xor     :::     xor: True True → False  
  Logical xor.

## Variables and scope

- **init**, **.init**   value “name” .init
  Create local variable and store value.  Value can be item or sequence.
  Local variables push their value to the TOS.
  - **init:** “variable_name” [list of values]
    Note swapped arguments.
- **def**, **.def**   {a b c} “name” .def
  Create global function and store value.
  Global functions push their value to the front of the command queue.
  If a block, it is unpacked for execution.
  - **def:** “function_name” {body of words to execute}
    Note swapped arguments.
- **get:**, **.get**   “name” .get
  Get referenced value, push to TOS.
- **set**, **.set**   value set “name”
  Set a variable/function value.
  - **set:**   set: “name” value
    Note the swapped arguments.
- **call:**, **.call**   “name”.call
  Execute referenced value by putting it on front of command queue.
  If a block, it is unpacked for execution.
- @name
  Synonym for “name”.get
- $name
  Synonym for “name”.set
- !name
  Synonym for “name”.call
- **free:**, **.free**   free: “name”
  Delete a variable.
- local_variable_name
  Push value to stack.
  - Names may not start with a number.  They may begin with a period, and may end with a colon.  Names (other than built in functions) may not otherwise include the following characters:  , ' ` " [ ] { } ( ) \ - # %
- global_function_name
  Execute value by pushing it to the front of the command queue.
  Blocks are unpacked for execution.  ({1 + 2} → 1 + 2)
- Local variables **A..Z** are always available, and are initialized to “” (empty string).
- Local scope is invoked with **(** and **)**.
  - n **(**   Saves the stack, creates a new one, pops n items from the old stack onto the new one, initializes local variables A..Z, and assigns the popped values to A..N.
  - **)**   Pops the contents of the stack onto the old one, then deletes the current stack and restores the old one with the new values added.  Also deletes the current local variable scope, restoring the previous one.

## Side stacks

- Side stacks **a..z** are globally available.
- **push_***n*

  Move TOS to side stack n.
- **pop_***n*

  Move top of side stack n to TOS.
- **copy_***n*

  Copy top of side stack n to TOS.
- **depth_***n*

  Push depth of side stack n to TOS.
- Several built in functions use side stacks for temporary data storage.

| | |
|---|---|
| **d:** apply_each | used for: command list |
| **e**: each, map, filter, total | used for: target list |
| **f**: for, for* | used for: counter |
| **g**: filter | used for: current list |
| **h**: filter | used for: filtered list |
| **n**: times, times* | used for: counter |
| **o**: case, match | used for: object (item to be matched) |
| **p**: case, match | used for: list of condition/body pairs |
| **r**: reduce | used for: target list |
| **s**: reduce | used for: result list |

## Blocks, lists, and strings

- **{ }** {a b c}
  Block of words, anonymous (lambda) function. Pushed to the stack.
  Will be executed when called as a global variable (function).
- **[ ]** [1 2 3]
  List of data. Pushed to the stack.
  Will be pushed to the stack when called as either a global or local variable.
- **type:**, **.type** [ list of items ] .type → LIST
  Returns: LIST, BLOCK, INT, FLOAT, STR, BOOL, WORD
- **len:**, **.len** len: [a b c d e] → [a b c d e] 5
  Returns count of items in list, preserving list.
- **first:**, **.first** [a b c d] .first → a
- **first*:**, **.first*** first*: [a b c d] . → [b c d] a
- **but_first:**, **.but_first** [1 2 3 4] .but_first → [2 3 4]
- **last:**, **.last** [a b c d] .last → d
- **last*:**, **.last*** last*: [a b c d] . → [a b c] d
- **but_last:**, **.but_last** but_last: [1 2 3 4] → [1 2 3]
- **delist:**, **.delist** a b [c d e] .delist → a b c d e 3
- **enlist:**, **.enlist** a b c d e 3 .enlist → a b [c d e]
- **enlist_all** 1 2 3 a b c d enlist_all → [1 2 3 a b c d]
  Converts the entire data stack to a list.
- **nth:**, **nth**, **.nth** [a b c d e ] nth 2 → [a b c d e] c
  Copies item n (beginning from 0, ending from -1) from list.
- i**nsert:**, **insert**, **.insert** [a b c d] insert {1 2 3} 1 → [a {1 2 3} b c d]
- **delete:**, **delete**, **.delete** delete: [a b c d] 3 → [a b c]
- **concat:**, **concat**, **.concat** [a b c] concat {1 2 3} → [a b c 1 2 3]
  Joins two items.
  - If both are sequences, takes the type of the first.
  - If only one is a sequence, takes that type.
  - If both are items, makes a list.
- **append:**, **append**, **.append** [1 2 3] append [4 5] → [1 2 3 [4 5]]
- **join:**, **join**, **.join** “Ahoy, ” join “matey!” → “Ahoy, matey!”
- **str>list_char**, **.str>list_char** “Hi there” .str>list_char → [“H” “i” “ “ “t” “h” “e” “r” “e”]
- **str>list_word**, **.str>list_word** str>list_word: “Hi there” → [“Hi” “there”]
- **list>str:**, **.list>str** [“Hi” “there”] .list>str → “Hithere”
- **list>str:_sp**, **.list>str_sp** list>str_sp: [“Hi” “there”] → “Hi there”
- **rev:**, **.rev** [a b c] .rev → [c b a]
- **in:**, **in**, **.in** 1 in [1 2 3] → True
- **str>word:**, **.str>word** “name” .str>word → name
  Will also convert numbers and Booleans. “” --> False
- **word>str:**, **.word>str** name .word>str → “name”
- **str>num:**, **.str>num** “123.45” .str>num → 123.45
  123 .str>num → 123 forgiving of types
- **num>str:**, **.num>str** 123 .num>str → “123”
- **list>block:**, **.list>block** [a b c] .list>block → {a b c}
- **block>list:**, .**block>list** block>list: {1 2 3} → [1 2 3]

- **range:**, **range**, **..**, **.range** start stop .range → [start next next ... stop]
  “d” .. ‘a’ → [d c b a] “..” is semantic sugar for infix range only
  Works with integers and characters, high to low or low to high
- **sort:**, **.sort** [5 2 4 1 3] .sort → [1 2 3 4 5]
  [e b “a” c d] .sort → [“a” b c d e]
  sort: [“a” “b” 3 4] fails, cannot sort mixed lists
- **valid_num?: .valid_num?** “3.14159” valid_num? → “3.14159” True
  checks if a string is a valid number
- **zip:**, **.zip** [1 2 3] [a b c] .zip → [[1 a] [2 b] [3 c]]
  extra elements in the longer list will be ignored
- **unzip:**, **unzip**, **.unzip** [[1 a] [2 b] [3 c]] .unzip → [1 2 3] [a b c]

## Control flow and combinators

- **exec:**, **.exec** {body} .exec
  Execute body, which can be sequence or item.
  Will convert a string with no spaces to a word or number.
  An empty string or sequence is consumed.
- **choose:**, **choose**, **.choose** bool choose {true} {false} → {true} or {false}
  “bool” can be an item (“True”) or a block (“{> 0}”) that evaluates to True or False.
- **if:**, **if**, **.if** {boolean} {do if true} {do if false} .if → {do if ...}
  *if: {10 > 0} then {println:”10 > 0”} else {”Incorrect universe error!”.err_msg fail}*
- **iff:**, **iff**, **.iff** {boolean} iff {do if true} → {do if true} *or* nop
  If and only if; if without else block
- **if*:**, **if***, **.if*** a if*: {bool} {true} {false} → {a true} *or* {false}
- **iff*:**, **iff***, **.iff*** a {bool} iff* {true} → {a true} *or* nop
- **<=>:**, **<=>**, **.<=>** number <=> {do if positive} {do if zero} {do if negative}
  Starship operator, selects based on sign (and zero).
- **while:**, **while**, **.while** {condition} {body} .while
  Do body while condition is true. Executes body 0 or more times.
- **until:**, **until**, **.until** {body} until {condition}
  Do body until condition is true. Executes body at least once.
- **break** Leave a while loop
- **cont** Go back to the beginning of a while loop.
- **then**, **else**, **do**, **of** Syntactic sugar, does nothing, removed by parser.
- **begin_while** A marker for cont and break, automatically added to while loops.
- **exit** Go to the next “end”.
- **end** Marker for “exit”, automatically added to the end of code.
- **halt** Immediately halt execution.
- **fail** Immedaitely halt execution as if an error occurred.
- **nop** No operation, do nothing.
- #  Comment
  Everything to the end of line is ignored and removed by the parser.
- **each:**, **each**, **.each** [list] {instructions} .each → results
  Apply instructions to each element in list, in order, left to right. Uses side stack e.
- **apply_each:**, **apply_each**, **.apply_each** [1 2 3] [{.print}{1 .+ .print}] .apply_each
  Apply a list of instructions to a list of arguments. Uses side stacks d and e.
- **map:**, **map**, **.map** [list] {instructions} .map → [results]
  Like each, but collects results in a list.
  Uses side stack e (each).
- **times:**, **times**, **.times** n {body} .times → result
  Executes body n times, n counts down towards 0.
  Uses side stack n to store the current value of n.
- **times*:**, **times***, **.times*** n {body} .times → result
  As .times, but the counter is pushed to TOS for body to use.
- **for:**, **for**, **.for** [{initial state} {incremental change} {exit condition}] {body} .for
  Executed body until exit condition is true, making incremental change to initial condition at the end of each loop.
  Uses side stack f.
- **for*:**, **for***, **.for*** [{initial state} {incremental change} {exit condition}] {body} .for
  As .for, but counter is pushed to top of stack for body to use.

- **filter:**, **filter**, **.filter** [list] {condition} .filter → [filtered list]
  Uses conditon to select items from list.
  Uses side stacks g, h, e (each, map).
- **reduce:**, **reduce**, **.reduce** [list] {action} .reduce → result
  [1 2 3 4] {.+} .reduce → 10
  [1] {.+} .reduce → 1
  [ ] {.*} .reduce → [ ] action is ignored is list is empty
  [1 2 3] { } .reduce → [1 2 3]
  [1 2 3] {.print} .reduce .println → "231"
  Action is applied progressively to each element of list from left to right .
  Action must be postfix or immediate.
  Uses side stacks r, s, e (each).
- **case:**, **case**, **.case**
  object [[cond1}{body1}][{cond2}{body2}]..[{cond_n}{body_n}]] .case
  Applies cond to object until returns True, then executes body.
  Both cond and body *must* be in a block or list.
  [{True}{drop}] is automatically added to ensure there is a default condition.
  Uses side stacks o, p.
- **case*:, case*, .case***
  As case, but object is pushed to the front of the stack for body to use.
- **match:**, **match**, **.match**
  object [[{item1}{body1}][{item2}{body2}..[{item_n}{body_n}]] .match
  Checks to see if sample is equal to item, then executes body.
  [{dup}{nop}] is automatically added to ensure there is a default condition.
  Uses side stacks o, p.
  - Can be used as a dictionary by using items instead of blocks:
    name [[name1, value1][name2, value2]..[name_n value_n]] .match
- **match*:**, **match***, **.match***
  As match, but object is pushed to the stack for body to use.

## Input and output

- **print:**, **.print** item .print ::: [1 2 3] .print
  Prints item, does not add a new line after.
  Sequences omit outer brackets/braces.
- **println:**, **.println** println: item ::: println: “Hello!”
  Prints item with a following newline.
  Sequences omit outer bracketsbraces.
- **print_quote:**, **.print_quote** item .print_quote
  Prints item with quotes around it.
  Sequences include outer braces/brackets.
- **println:**, **.println** println: item ::: println_quote: “Hello!”
  Prints item with quotes around it and a following newline.
  Sequences include outer braces/brackets.
- **emit:**, **.emit** n .emit
  Prints n as its ascii character. (10 = new line, 32 = space)
- **dump**
  Pretty-prints the entire data stack without changing it.
- **get_line**
  Inputs a string from the keyboard, ending with return (return not copied).
- **get_char**
  Reads a single key from the keyboard, printing it on the screen.
- **get_char_silent**
  Reads a single key from the keyboard without printing it on the screen.
- **load:**, **.load** load:“filename.ls” ***meta command***
  Loads filename.ls and runs it immediately.
  The file must be in the same directory and the “.ls” extension is mandatory.
  “filename”.load will execute correctly; the “.ls” will be added if missing.
- **err_msg:**, **.err_msg** “message” .err_msg ***meta command***
  sets the user error message, to be printed in case of error causing program termination.

## Meta programming

- Meta programming is used to create the pre- and infix variants and also to copy and move words around in the control flow words. Meta programming is fully available to the user.
- **_meta_** [num_past, num_future, "pattern"] _meta_
  Pop a number of items from the stack and the queue, and apply them using "pattern" to the front of the queue.
  - **#a** .. **#n** applies items from the stack, in order a (lower in stack) to n (top of stack).
  - **#A** .. **#N** applies items from the queue in the same way.
  - **%a** expands a sequence.
  - **#B0** takes the first item from the sequence denoted by B. Only one digit can be used, so you can only go 10 items deep into a sequence.
    - .*while* (postfix) is implemented as:
      [2, 0, "%a {%b begin_while #a #b .while} {nop} .if"].
      Note that "begin_while" is a target for *cont* and *break*, and does nothing.
    - *while:* (prefix) is implemented as: \ .while _ins_f2
    - *while* (infix) is implemented as: \ .while _ins_f1
- **_swap_ff**
  Swap the top two items on the command queue (“future”).
- **_swap_fp**
  Swap the top items on the command queue (“future”) and the data stack (“past”).
- **_ins_f1/2/3/4** \ .while _ins_f2
  Insert the item on the top of the stack 1/2/3/4 deep in the command queue.
- **load:**, **.load** load:“filename.ls”
  Loads filename.ls and runs it immediately.
  The file must be in the same directory and the “.ls” extension is mandatory.
  “filename”.load will execute correctly; the “.ls” will be added if missing.
- **err_msg:**, **.err_msg** “message” .err_msg
  Sets the user error message to be printed in case of an error causing program termination.

## Example programs

```
# Fibonacci sequence solutions
# syntax:  n  .whatever_fib              n is a number (positive integer to start with)

# Programs this terse parse correctly, but are hard to read.
{dup.print" --> “.print 0swap{.+}.times*.println}".naive_fib".def
# This works with positive integers and zero, but fails otherwise.

# This is a bit easier to read and handles negative numbers
 Def: ".simple_fib" {
 dup .print " --> ".print dup
 > 0 if { dup
         while: {1 .- dup 0 .>}
         do {dup roll .+ swap}
    drop .println}
 else {drop 0 .print cr}
 }
# This creates the prefix version from the postfix base.
def: "simple_fib:" {\.simple_fib  _ins_f1}

# This is a general solution, allowing floating point numbers.
 def: "general_fib" {
 dup .print " --> ".print dup
  <=>
         {dup {1 .+ dup 0 .<} {dup roll .+ swap} .while}      # if negative
         {dup}                                                # if zero
         {dup {1 .- dup 0 .>} {dup roll .+ swap} .while}      # if positive
 drop .println }

# This is the fast solution, using math instead of looping.
# Sum(n) = (n*(n+1)) / 2 for positive integers, with a fractional part adding: fraction*(n+1)
 def: "fast_fib" {
 dup dup .print " --> ".print
  <=>
         {1 ./% swap dup roll swap 1 .-  .*             # fractional_part * (n-1)
                 swap dup 1 .-  .*  -2  .// swap .-}    # (n*(n-1)) / -2
         {nop}                                          # nop = no operation, do nothing
         {1 ./% swap dup roll swap 1 .+  .*             # fractional_part * (n+1)
                 swap dup 1  .+  .* 2  .//  .+}         # (n*(n+1)) / 2, no final swap because 1+2=2+1
 .println }
```

```
# Prototypes for filter
# syntax:  [list] {condition} filt → [filtered list]
# filters list by condition, producing a filtered list

# Uses local variables, fails when items deeper in the stack are needed
def: "buggy_filt" {over swap .map 2 (clear [] $C B {if {A .first* swap $A C swap .append $C}
{A .but-first $A}} .each clear C )}

# Uses side stacks, allows full access to the data stack.
{ over swap .map                # produces a list of True/False values
swap push_g                     # g is the list to be filtered
    [ ] push_h                  # h is the filtered list, empty to begin with
    {if                         # True/False from the mapped list
        {pop_g .first* swap push_g pop_h swap .append push_h}
        {pop_g .but_first push_g}
    } .each
    pop_g drop pop_h
} “filt” .def

# Variable and function names are checked after built in commands.
# You can create variables and functions with the same names,
# but they won’t work unless you use get (@name), set ($name), or call (!name).
```