Listack documentation

version 0.34

Listack is an experiment in making a symmetric, stackless, stack-based, concatenative language. Listack was inspired by both Factor and Falsish , which is itself 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. 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 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. *Concatenative* languages, which are normally stack-based, are similar to many functional languages in that function composition is accomplished by typing one command after another. The output from one word is the input to the next word via the data stack, much like the *nix pipe ("|") command.

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, 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.

- Prefix +: 1 2
- Infix 1 + 2
- Postfix 1 2 .+
- The data types are INT, FLOAT, STR, LIST, BLOCK, BOOL, and WORD
- Comma “,” is 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 immutable and pushed onto the stack.
- Strings are immutable and pushed onto the stack. Strings can be converted into lists and vice-versa.
- Sequences (lists and blocks) are mutable and pushed onto the stack. There is very 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 "get", while $name is shorthand for "set".

- Scope is administered manually using esolang fish inspired "(" and ")" commands.
- "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 choose and exec. ".if" is actually defined as {.choose .exec}.
- "#" begins a comment (except in meta programming). Comment extends to end of line.
- A rather long list of command words is available. 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 couple thousand lines of Python 3.10 code.

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.
- [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 used by cont and break, and evaluates to nop (no operation / pass).
- while: (prefix) is implemented as: \ *.while _ins_f2*
- while (infix) is implemented as: \ *.while _ins_f1*

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

Math functions

- **+:**, **+**, **.+** 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, b^{th} 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 .ln
- **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

Boolean logic

- **False**, **false**, **FALSE**, **F**, **f**
  Boolean False (also 0 and empty List/Block/String)
- **True**, **true**, **TRUE**, **T**, **t**
  Boolean True (also anything that isn't False)
- **<:**, **<**, **.<** 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
- **=:**, **=**, **.=** a b .=
  is a equal to b
- **!=:**, **!=**, **.!=** a b .!=
  is a not equal to b
- **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

Blocks, lists, variables, scope, side stacks

- **{}** {a b c}
  block of words, anonymous (lambda) function.
  by itself, pushed to the stack. Can be called or done. Use with global functions.
- **[]** [1 2 3]
  list of data. Pushed to the stack. Can be called or done.
- **`** `word
  backquote. Shorthand for {word}
- **** \ word
  Ignore the following word and push it to the stack for later use
- **init**, **.init** value “name” .init
  create local variable and store value. Value can be item or sequence
- **init:** “variable_name” [list of values]
  note swapped arguments
- **def**, **.def** {a b c} “name” .def
  create global function and store value
- **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
- **@** @name
  synonym for “name”.get
- **$** $name
  synonym for “name”.set
- **free:**, **.free** varname.free
  delete a variable
- variable_name push value to TOS.
- function_name execute value by pushing it to the front of the command queue.
- 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: , ' ` " [ ] { } ( ) \ - # %
- Local variables **A..Z** are always available, and are initialized to “”.
- 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. IT also deletes the current local variable scope, restoring the previous one.
- Side stacks **a..z** are globally available.
- Side stacks are used with **push_*n***, **pop_*n***, **copy_*n*** and **depth_*n***, where n is the side stack’s name (a..z).

- Several built in functions use some of the side stacks.
  e: each, map, filter, total
  f: for counter
  g: filter current list
  h: filter filtered list
  n: times, times*
  o: case, match
  p: case, match
  t: total
  u: total

Control flow and higher functions

- **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 sequence (“{0 .>}”)
- **if:**, **if**, **.if** {boolean} {do if true} {do if false} .if → {do if ...} .exec
  .if = .choose .exec
  {10 > 0} if then {println:”10 > 0”} else {”Incorrect universe error!”.println halt}
- **iff:**, **iff**, **.iff** {boolean} iff {do if true} → {do if true} *or* nop .exec
  if and only if, if without else block
- **if*:**, **if***, **.if*** a if*: {bool} {true} {false} → {a true} *or* {false} .exec
- **iff*:**, **iff***, **.iff*** a {bool} iff* {true} → {a true} *or* nop .exec
- **while:**, **while**, **.while** {condition} {body} .while
  do body while condition is true
- **then**, **else**, **do**, **of** syntactic sugar, does nothing, removed by parser
- **break** leave a while loop
- **cont** go back to the beginning of a while loop
- **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, automatically added to end of code
- **nop** no operation, do nothing
- **each:**, **each**, **.each** [list] {instructions} .each → results
  apply instructions to each element in list, in order, left to right
  uses side stack 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 top of stack 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)
- **total:**, **total**, **.total** [list] {action} .total → result ::: [1 2 3 4] {.+} .total → 10
  total of action applied to each element of list from left to right
  action must be postfix or immediate
  uses side stacks t, u, 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
- **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
    both item and body *must* be in a block or list
    [{dup}{nop}] is automatically added to ensure there is a default condition
    uses side stacks o, p