Listack

v0.4.1.0

Charles Fout, July 2023

Listack is a symmetric, flat, concatenative, polymorphic, interpreted programming language. Like most concatenative languages, it is stack based. Listack was inspired by the stack-based languages Factor, False, XY, and Forth. The manual scope system was inspired by Fish. The object type system was inspired by Euphoria. This document refers to words (the traditional concatenative language term) interchangeably with functions and procedures. Listack is an impure functional language.

Listack is symmetric because it employs a universal function call syntax, where almost every word (command) can be in a *prefix* (+: 1 2), *infix* (1 + 2), or *postfix* (1 2 .+) form. Prefix (word:) and infix (word) forms are translated at run time into postfix (.word) form. The infix form is always used after the first argument. There is also a more "normal looking" *callable* form: +(1, 2). This is translated to postfix by the parser when a program is loaded and is thus more efficient than the prefix and infix forms. Immediate words are always executed, well, immediately, and never relocated by the parser or the runtime interpreter. It is important to understand that all words are implemented and executed as either postfix or immediate.

Listack is flat because every function is inlined. There is no call stack of functions to return to. Looping is conducted purely by recursive inlining of functions. The only data flow primitives in the language are variants of if and <=>. All the "higher level" words are written in Listack, and can be examined in the system.ls file.

Listack is concatenative because data flow is through the stack. Functions are composed through juxtaposition, simply by writing them one after another with no further syntax. They all take their data from and add it to the stack, much like the *nix pipe '|' command. ("TOS" hereafter refers to the Top Of Stack.) The order of operations is always "first come, first served", noting that all words are executed as either postfix or immediate.
Example: [1, 2, [3, 4], 5] reverse flatten reverse → [1, 2, 4, 3, 5]
Example: 3 times {"Hi." print sp} println: "Bye!" → *Hi. Hi. Hi. Bye!*

Listack is polymorphic through overloading and multiple dispatch. Every word (function) definition requires a list of argument types. Each word can be defined multiple times using different arguments. Listack does, however, restrict each word to using the same number of arguments for each variant (each word has a single arity). This can be avoided by using a collection type (List, Block, or Seq) as one or more arguments, or by defining words in different namespaces. Listack also allows a singular type "Otherwise" to handle error conditions where the expected data types are not found on the stack.

Listack makes use of name spaces. Program files may be imported, and each definition occupies space in an environment based on the name of the file. Name spaces can be created on the fly, as desired. The default name space is "default". Note that "core" words cannot be altered, although their use may be extended through non-Otherwise variants. "System" words can have user created "Otherwise" variants". All variables are local, and are not part of a name space.

Listack has two types of built-in variables: local variables a through z, and global stacks A through Z. Users may create (and destroy) new variables as desired. Local variable scope (indicated by "|> <|") is manually handled by the programmer, but must be properly nested. Every local scope redefines a..z and initializes each to *none*. Variable names are searched from newest to oldest scope. Local scope also creates a new data stack, optionally taking data from the stack beneath it upon creation. Closing scope moves all current stack items to the next lower data stack. The data stack can also be saved and restored independently of scope.

Another feature of Listack is that the functional forms of "good" and "bad" are built in to every piece of data. (These correspond to the functional concepts of 'some' and 'none'.) An invalid or corrupt piece of data is marked bad(type), and will contaminate everything it gets used with if not handled in a timely manner. Local variables a..z are initialized to the special *none* value, which is implemented as >bad(nil) .

Listack handles errors in a quiet way. Most words will log non-parsing errors and write the total number of errors to the screen without halting the program. Users can create their own error handling through the use of "Otherwise" variants.

Blocks are code blocks (lambda functions). (The traditional concatenative term is 'quote', short for "quoted program".) Lists are collections of data. Seqs (sequences) are code blocks that are immediately executed. Other than words, variable names, and sequences, each item in the command queue is, upon evaluation, simply moved to the top of the data stack. "\" defers evaluation of the following item, so \(1+2) will move (1+2) to the top of the data stack instead of being immediately evaluated to 3.

Line comments begin with '**#**' and go to the end of the line. Block comments begin with **#:** and continue until **.#** is found.

Commas (outside strings and characters) are considered to be white space by the parser. Underscores in numbers are ignored, so 100_000 is a valid integer.

Strings are surrounded by either 'single' or "double" quotes. Characters are preceded by `a back tick. The normal escape characters are used. Listack currently supports only ASCII characters (codes 0 to 127).

To compile Listack, place all the files together in one directory and type the following on the *nix command line. (This assumes you have already installed Nim.)

nim c listack.nim ***or*** nim c -d:release listack.nim *(faster but less safe)*

The name Listack is a portmanteau of list and stack, the basic building blocks. Listack 0.4.1 is implemented in Nim 1.6.14. Call Listack from the command prompt as follows:

To execute a program file: ./listack *filename.ls* [-debug] [-verbose]

To explore Listack with the interactive REPL: ./listack

Local Variables

Local variable names a through z are always available, and are recreated in each scope. The standard variables are initialized to *none*.

Create: **init** [Any, Wordy]	1.414 init “my_sqrt2”
Destroy: **free** [Wordy] (cannot destroy a..z)	“my_sqrt2” free
Access: *name or* **get** [Alpha] *or* **@<***name or* **@*** *name*	my_sqrt2
Alter: **set** [Alpha] *or* **@>***name*	0.707 @>my_sqrt2
Clear: **@/***name* (resets it to *none*)	@/my_sqrt2
Evaluate: **call** [Alpha] *or* **@!***name*	@!my_sqrt2
Check status: **@?***name* (returns 1, or 0 if *none*)	@?my_sqrt2

Scope

|> [Num] / [Bool] *immediate*

- Open scope
- Creates new local variables a through z.
- Assigns n items from the old data stack to a..n (maximum 26, where a is deepest).
- Creates a new data stack.
- If n is positive, it moves n items from the old stack to the new one.
- If n is negative, it removes n items from the old stack without copying them to the new.
- If n is true, copies every item on the old stack to the new one, copying the top 26 (at most) values to variables a..z (a is deepest).
- If n is false, moves every item from the old stack to the new one, copying the top 26 (at most) values to variables a..z (a is deepest).

<| [] *immediate*

- Close scope
- Deletes current local variables.
- Moves every item in the current stack to the end of the older stack.
- Deletes the current stack.

Examples:

- 1 2 3 2 |> dump clear 4 b <| dump
  - prints “2 3”, then prints “1 4 3”
- 1 2 3 -2 |> dump clear 4 a <| dump
  - prints “”, then prints “1 4 2”
- 1 2 3 true |> clear a b c <| dump
  - prints “1 2 3 1 2 3”
- 1 2 3 false |> clear c b a <| dump
  - prints “3 2 1”

Global Variables

Global variables are auxiliary stacks with names A through Z.
They are initialized at startup as empty stacks.
Users can create and destroy their own global variables.
System global variables begin with an underscore ('_') and cannot be destroyed.

Create: **create_global** [Wordy]
Destroy: **free** [Wordy] (cannot destroy A..Z or the system globals)
Access: *name* (returns copy of top item)
Pop from top: **get** [Alpha] *or* **@<***name*
Push to top: **set** [Alpha] *or* **@>***name*
Clear: **@/***name* (empties stack)
Evaluate: **call** [Alpha] *or* **@!***name* (evaluates copy of top item)
Check depth: **@?***name*
Copy entire stack: **@***** *name*

Examples:

- create_global: "_dipped"
- def: "dip" [Any, Any] {swap @>_dipped eval @<_dipped}

Types

Listack uses a few basic types: **Bool**, **Int**, **Float**, **Char**, **String**, **Word**, **List**, **Block**, **Seq**, **Object** and **Null**. Ints and Floats are both 64 bit. The only object of type Null is *nil*, which is used along with *none* to indicate invalid data. Types can be entered as either Words or Strings. There are several combined ("or"/"additive") data types:

- **Coll** = List, Block, Seq
- **Item** = Bool, Int, Float, Char, String, Word, Object
- **Num** = Int, Float
- **Alpha** = Char, String, Word
- **Alphanum** = Char, String, Word, Int, Float
- **Wordy** = String, Word
- **Blocky** = Block, Seq
- **Listy** = List, Block
- **Executable** = Word, Block, Seq
- **Otherwise** = A catch-all for when the stack doesn't match the expected types.

create_type [Wordy, Listy]

- create your own combined types. Must be a unique name.

A List is enclosed in brackets: [] and is used for data.
A Block is enclosed in braces: {} and is used for programs.
A Seq is enclosed in parenthesis: () and is used for immediately executed programs and "callable" function forms (which are immediately executed unless deferred).
An Object can be created as follows: ($ Type [persistent data] value $)
Strings are in 'single' or "double" quotes.
Chars are preceded by `a back tick.

Definitions

def [Wordy, Listy, Blocky] / [Wordy, Listy, Word]
- Creates a word and adds it to the current name space.
- If the word already exists, it adds a variant.
- Remember that all non-immediate words are actually executed as postfix.
  - def: "dip" [Any, Any] {swap @>_dipped eval @<_dipped}
  - "nth*" def [Coll, List] {dupd {.nth nip} .each}
  - "nth" [Coll, List] {{.nth nip} .each} .def
  - def("def", [Wordy, Listy, Word], {>block .def})

def_immediate [Wordy, Listy, Blocky] / [Wordy, Listy, Word]
- Creates an immediate word and adds it to the current name space.
- If the word already exists, it adds a variant.
- Immediate words can have arguments.
  - def_immediate: "sp" [] {32 .emit}
  - def_immediate: “dupd” [Any, Any] {over swap}

def_sugar [Wordy]
- Creates a syntactic sugar word. This word is then removed from the command queue wherever found and eliminated by the parser.
- Built-in sugar words: **then**, **else**, **do**, **of**
- Commas are also syntactic sugar. They are replaced with a space by the parser.

word_def [Wordy]
- Creates a list of all the definitions of a word. Core words are defined as themselves.

Namespaces

_set_namespace [String] Create or use a namespace.
_reset_namespace [] Resets namespace to default.

- Listack automatically adds _create_namespace: ‘*filename* ‘ (without the trailing “.ls”) to the beginning of an imported file and _reset_namespace _end_ at the end.
- The default namespace is *default*.
- Built-in words are either in *core* or *system*.
  - Core words are written in Nim.
  - System words are written in Listack.
    - Most of the control flow words other than ‘<=>’ and ‘if’ variants are written in Listack. You can read them in *system.ls* .

; Identify a word’s namespace.
- wordname**;***namespace*
- This is only needed if the same word has separate meanings in different namespaces.

Important: Definitions have namespaces, variables have scope. Variable and definition names begin with a letter or symbol (<u>not</u> a digit), and can contain digits after a letter or underscore has been included. They may <u>not</u> contain the following characters:

“ ‘ ` [ ] { } ( ) | \ # ; : . ,

The Data Stack

dump [] *immediate*
- Prints the contents of the current data stack.

clear [] *immediate*
- Empties the current data stack.

reverse_stack [] *immediate*
- Reverses the elements in the current stack.

rotate_r [Int] *immediate*
- Rotates the stack n items to the right, wrapping around at the ends.

rotate_l [Int] *immediate*
- Rotates the stack n items to the left, wrapping around at the ends.

_save_stack [] *immediate*
- Creates a new, empty stack and places it above the current stack.

_copy_stack [] *immediate*
- Copies the current stack and places the copy above the current stack.

_restore_stack [] *immediate*
- Deletes the current stack and its contents.
- If the current stack is the bottommost (there are no saved stacks), it clears it instead.

_merge_stack [] *immediate*
- Adds every element in the stack to the stack beneath it, then deletes the top stack
- If the current stack is the bottommost, does nothing.

depth [] *immediate*
- Adds the current stack depth to TOS as an Int.

Every word in a stack-based language takes its arguments from, and returns its results to, the data stack. A stack is a last in, first out sequence, where the "top" holds the most recent item. Items are pushed to the stack and popped from the stack. Think of the stack as the scratch pad that is always used to do temporary work. Variables are assigned from data on the stack, and references to variables place their values on the stack.

A stack item can be *anything* in the language except comments (which are deleted by the parser). Numbers, words (function names), variables, blocks, lists, strings, anything and everything is worked on through the stack. When evaluated, everything except undeferred words , variables, and sequences simply gets pushed to TOS. Items are evaluated in the order they appear in the command queue. There is no precedence other than 'first come, first served', except for the "fix" (prefix, infix, postfix) notation of (non-immediate) words.

- 1 + 2 → 3
- 1 2 .+ → 3
- +: 1 2 → 3
- +(1, 2) → 3 # Commas are optional and considered to be spaces.
- 1 + 2 * 3 → 9
- 1 2 .+ 3 .* → 9
- 1 2 .+ * 3 → 9
- 3 * (1 + 2) → 9
- *(3 +(1, 2)) → 9
- 3, 1, 2 .+ .* → 9

Stack Manipulation

dup [Any] *immediate*	a → a a	
dup2 [Any, Any] *immediate*	a b → a b a b	(over over)
dupd [Any, Any] *immediate*	a b → a a b	(over swap)
drop [Any] *immediate*	a b → a	
drop2 [Any, Any] *immediate*	a b c → a	(drop drop)
swap [Any, Any] *immediate*	a b → b a	
over [Any, Any] *immediate*	a b → a b a	
over2 [Any, Any, Any] *immediate*	a b c → a b c a b	(pick pick)
nip [Any, Any] *immediate*	a b → b	(swap drop)
nip2 [Any, Any, Any] *immediate*	a b c → c	(nip nip)
tuck [Any, Any] *immediate*	a b → b a b	(swap over)
pick [Any, Any, Any] *immediate*	a b c → a b c a	
roll [Any, Any, Any] *immediate*	a b c → b c a	
dupd [Any, Any] *immediate*	a b → a a b	({dup} dip) / (over swap)
swapd [Any, Any, Any] *immediate*	a b c → b a c	({swap} dip)

Stack manipulation with function evaluation:

dip [Any, Executable] *immediate*

- a b {commands} dip → a commands b
  - def: “dupd” {{dup} dip}

keep [Any, Executable] *immediate*

- a b {commands} keep → a b commands b
  - def: "in*" [Item, Coll] {{.in} keep swap}

For the sake of readability, try to keep stack manipulation to a minimum.
If more complicated stack manipulation is required, use scoped variables instead.

- 1 2 3 4 5 -5 |> a c e <| → 1 3 5

Basic Control Flow / Conditionals

eval [Any] Evaluates the item on TOS.

- If a Block or Seq, unwrap it and push the contents to the front of the command queue.
- If a word, execute it.
- If a variable, push value on TOS or as appropriate when modified by @.
- Anything else gets pushed to TOS.
- A prefix or infix word with insufficient arguments following is still evaluated.
  - 2 3 .+ → 5
  - 2 3 + → 5
  - 2 3 +: → 5
  - 2 3 .+ 1 → 5 1
  - 2 3 + 1 → 2 4
  - 2 3 +: 1 → 2 4
  - 2 3 +: 1 7 → 2 3 8

if [Bool, Any, Any] / value [Blocky, Any, Any]

- if: {condition} then {do if true} else {do if false}
  - 10 {dup > 5} if {print " is > 5" println} {print " is not > 5" println}

if* [Bool, Any, Any]

- Same as if

if* value [Blocky, Any, Any]

- Duplicates value so it can be used after the conditional is evaluated.
  - 10 {> 5} if* {print " is > 5" println} {print " is not > 5" println}

iff [Bool, Any] / value [Blocky, Any]

- If and only if, no else block
  - 10 iff: {dup > 5} {print " is > 5" println} # if false, value is left on the stack

iff* [Bool, Any]

- Same as iff

iff* value [Blocky, Any]

- Keeps the initial value for use in the true block
  - 10 {> 5} {print " is > 5" println} .iff* # if false, value is *not* left on the stack

<=> [Num, Any, Any, Any] / [Blocky, Any, Any, Any]

- Trinary"starship operator"
- Compares number to zero
  - 10 <=> {"negative" println} {"zero" println} {"positive" println} → prints "positive"

dip [Any, Executable] *immediate*

- a b {commands} dip → a commands b
  - def: "dupd" {{dup} dip}

keep [Any, Executable] *immediate*

- a b {commands} keep → a b commands b
  - def: "in*" [Item, Coll] {{.in} keep swap}

then, **else**, **do**, **of**

- Syntactic sugar.
- Deleted by the parser so they do not interfere with argument counts or positions.
  - 10 if*: {> 5} then {print " > 5" print} else {print " <= 5" print} cr → prints "10 > 5"

while [Bool, Executable] / [Blocky, Executable]

- {conditional} {do while true body} .while
- Executes body 0 or more times.
- Written in Listack, part of system.
- true while {println: “Infinite loop!”}
- 10 {dup > 0} while {dup print sp dec} print " Blastoff!" println

until [Blocky, Bool] / [Word, Bool] / [Blocky, Blocky] / [Word, Blocky]

- {do while false body} until {conditional}
- Executes body at least once
- Written in Listack, part of system.
- 10 {dup print sp dec} until {dup < 0} drop “ Blastoff!” println
- {println: “Infinite loop!”} until {false}
- {println: “Prints only once!”} until true

_begin_loop_

- Marker for the beginning of a loop (for *continue*). Internal use only.

_end_loop_

- Marker for the end of a loop (for *break*). Internal use only.

end

- Marker for the end of a function or block. Use with *exit*.
- Automatically applied to the end of an imported file.

break

- Breaks out of a loop by ignoring everything until _end_loop_, end, or _end_proc_.

continue

- Ignores further commands until it finds the next _begin_loop_, _end_loop_, end, or _end_proc_.

exit

- Ignores all commands until it finds the next end or _end_proc_.

halt

- Ends the current program.
- The current REPL session will continue.

fail

- Ends the current program and generates an error.
- The current REPL session will continue.

quit

- Aborts Listack completely.
- Will end the REPL.

_end_proc_

- Automatically added to the end of each definition.
- Internal use only.

Advanced Control Flow
"Higher Order Functions"

These words are written in Listack itself. Look at *system.ls* to see their actual definitions.

each [Any, Executable]
- [List of things to act upon] {action to take on each item] .each
  - [1 2 3 4] {dup .* print sp} .each cr

apply_each [Any, List] / [Any, Executable]
- [List of things to act upon] [{List of actions} {to be taken in order}] .apply_each
  - [1 2 3 4] [{print sp} {dup .+ print sp} {dup .* print sp}] .apply_each cr

apply_each_then [Any, List, Executable] / [Any, Executable, Executable]
- [List of things] [{List}{of}{actions}] {thing to do between actions} .apply_each_then
  - [1 2 3 4 5] [{print sp} {dup .+ print sp} {dup .* print sp}] {cr} .apply_each_then

bi_each [Any, Executable, Executable] *immediate*
- Applies two words or blocks, in order, to an item or collection.
- Defined as {1 2 \.apply_each a_b_apply_f}
  - 5 {dup .+} {dup .*} bi_each → 10 25
  - [5 10] {dup .+} {dup .*} bi_each → 10 20 25 100

map [Any, Executable]
- [List of things] {Action to take on each item} .map
- map works as each, but collects the results in a new Coll (defaults to List).
  - [1 2 3 4 5] {dup .*} .map → [1 4 9 16 25]
  - {1 2 3 4 5} {dup dup .+ swap dup .*} .map → {2 1 4 4 6 9 8 16 10 25}
  - 5 {1 .+} .map → [6]

apply_map [Any, List] / [Any, Executable]
- [List of things to act upon] [{List of actions} {to be taken in order}] .apply_map
  - [1 2 3 4] [{print sp} {dup .+ print sp} {dup .* print sp}] .apply_map cr

apply_map_then [Any, List, Executable] / [Any, Executable, Executable]
- [List of things] [{List}{of}{actions}] {thing to do between actions} .apply_map_then
  - [1 2 3 4 5] [{print sp} {dup .+ print sp} {dup .* print sp}] {cr} .apply_map_then

bi_map [Any, Executable, Executable] *immediate*
- Applies two words or blocks, in order, to an item or collection, and produces 2 Colls.
- Defined as {1 2 \.apply_map a_b_apply_f}
  - 5 {dup .+} {dup .*} bi_map → [10] [25]
  - [5 10] {dup .+} {dup .*} bi_apply → [10 20] [25 100]

a_b_apply_f [Any, Any, Int, Int, Executable] *immediate*
- gathers b items into a list of executables, gathers a items into a list of data, then evaluates function f on them.
  - 3 5 {dup .+} {dup .*} 2 2 \.apply_map a_b_apply_f
    → [3, 5] [{dup .+}, {dup .*}] .apply_map
    → [6 10] [9 25]

when [Any, Executable]
- Similar to "iff", but evaluates argument with ">bool".
  - "Hello" when {println: "Howdy!"} → prints "Howdy!"
  - "" when {println: "Hello!"} → does nothing

when* Preserves the argument.
  - 42 when {dup .+} → 84

times [Executable, Int] / [Int, Executable]

- Does something a certain number of times. If number is zero or less, does nothing.
  - {print: “Hello! “} 3 .times → prints “Hello! Hello! Hello! “
  - 0 times {print:”Hello! “} → does nothing

times* [Executable, Int] / [Int, Executable]

- Does something a certain number of times, leaving the counter on TOS for the action to use each time.
- Counts down.
  - 5 times* {print sp} cr → prints “5 4 3 2 1”

for [Coll, Executable]

- [{initial state}{continuing condition}{incremental change}] {body to execute}
- Must be careful to not alter or delete the discriminator accidentally
  - for: [{1} {dup < 5} {inc}] {dup print sp} → prints “1 2 3 4”
  - for: [{1} {< 5} {inc}] {print sp} → creates errors because the discriminator is consumed

reduce [Coll, Executable]

- [List to be reduced to a single item] {Action to take on each item in sequence}
  - [1 2 3 4 5] reduce {.+} → 15
  - [] reduce {.+} → []

filter [Coll, Executable]

- [List of items to filter] {Discriminating function}
  - [1 -2 -3 4 5] filter {0 .<} → [-2 -3]

case [Any, Coll]

- Item, [[{check 1} {action 1}], [{check 2} {action 2}]]
- Checks item using the first function of each pair. If true, executes the corresponding action (second item in pair) and then halts.
- Default action is to leave bad(Item) on TOS.
  - "Hello" [[{= "Hi" }, {print:"Hi!"}], [{= "Hello"}, {print:"Howdy!"}]] .case → prints “Howdy!”
  - 42 [[{= "Hi"}{print:"Hi!"}], [{= "Hello"} {print: "Howdy!"}]] .case → bad(Int)

case* [Any, Coll]

- As case, but leaves item on TOS for action to use.
  - "Hi" [[{= "Hi"}{print println:"!"}], [{= "Hello"} {print println: "!"}]] .case → prints “Hi!”

match [Any, Coll]

- Similar to case, but checks for equality instead of a user defined condition.
- Must be an exact match. Do not add extraneous braces.
  - "Hello" [["Hi" {println: "Hi!"}] ["Hello" {println: "Howdy!"}]] .match → prints “Howdy!”
  - 42 [["Hi" {println: "Hi!"}] ["Hello" {println: "Howdy!"}]] .match → bad[Int]

match* [Any, Coll]

- Similar to case*, but checks for equality instead of a user defined condition.
- Must be an exact match. Do not add extraneous braces.
  - "Hello" [[{"Hi"}{print println:"!"}], ["Hello" {print println: "!"}]] .match* → prints “Hello!”
  - "Hi" [[{"Hi"}{print println:"!"}], ["Hello" {print println: "!"}]] .match* → bad(String)
  - {"Hi"} [[{"Hi"}{print println:"!"}], ["Hello" {print println: "!"}]] .match* → prints “{Hi!}”

pairwise [Coll, Executable]

- Executes a binary operation on each pair of items in the collection.
  - [1 2 3 4 5 10] pairwise \.+ → [3, 5, 7, 9, 15]

map_reduce [Coll, Executable]

- Similar to pairwise, but accumulates the result as the first item in the next pair.
  - [1 2 3 4 5 10\] map_reduce {.+} → [3, 6, 10, 15, 25]

both [Any, Any, Executable] *immediate*

- Applies one function to two arguments.
  - 3 4 {dup .*} both → 9 16

Booleans / Comparisons

= [Any, Any]
- Equivalence
- 42 = 42.0 → true
- “A” = `A → true
- “Word” = \Word → true

== [Any, Any]
- Strict equality
- 42 == 42.0 → false

!= [Any, Any]
- Not equivalent

!== [Any, Any]
- Not strictly equal to

~= [Any, Any]
- Approximately equal to
- When comparing Float to Float, true if difference is less than $1x10^{-12}$
- When comparing Float to Int or Int to Float, true if difference is less than $1x10^{-6}$
- Otherwise, applies =

< [Num, Num] [String, String] [Char, Char]
<= [Num, Num] [String, String] [Char, Char]
> [Num, Num] [String, String] [Char, Char]
>= [Num, Num] [String, String] [Char, Char]
not [Bool]
- true not → false

and [Bool, Bool]
- true and false → false

or [Bool, Bool]
- true or false → true

xor [Bool, Bool]
- true xor true → false

nor [Bool, Bool]
- false nor false → true

nand [Bool, Bool]
- true nand true → false

>**bool** [Any]
- Numbers are true if they are not 0.
- Strings are true if they are not empty (“”).
- Chars are true if they are not \0.
- Collections are true if they are not empty.
- Null (nil/none) is always false.
- The truth of an object is determined by its value.
- *Bad* items are always false.

The following queries each examine the TOS and return a Bool, preserving the argument.

bad?	
good?	
none?	*none* is >bad(nil)
Null?	*nil* is the only member of type Null.
Bool?	*true* or *false*
Char?	
String?	
Word?	
Int?	
Float?	
List?	
Block?	
Seq?	
Object?	
Num?	Int, Float
Alpha?	String, Word, Char
Alphanum?	String, Word, Char, Int, Float
Item?	Bool, Int, Float, String, Word, Char, Object
Coll?	List, Block, Seq
Wordy?	String, Word
Blocky?	Block, Seq
Listy?	List, Block
Executable?	Word, Block, Seq
local?	
global?	
variable?	
deferred?	
empty?	[], {}, (), “” → true

Bitwise words

Remember that all integers are signed 64 bit.

bit_and [Int, Int]
- 3 bit_and 1 → 1

bit_or [Int, Int]
- 4 bit_or 2 → 6

bit_xor [Int, Int]
- 7 bit_xor 3 → 4

bit_nand [Int, Int]
- 8 bit_nand 4 → -1

bit_nor [Int, Int]
- 8 bit_nor 6 → -15

bit_not [Int]
- Reverses each bit, maintaining order.
- 8 bit_not → -9

bit_<< [Int, Int]
- Left shifts a by b bits.
- 4 bit_<< 2 → 16

bit_>> [Int, Int]
- Right shifts a by b bits.
- 8 bit_>> 1 → 4

bit_rev [Int]
- Reverses the order of the bits.
- 9 bit_rev → -8070450532247928832

bit_print [Int]
- Prints the binary of the number.

bit_println [Int]
- Prints the binary of the number followed by a newline.

Object related words

($** Type [arguments] value **$)

- Object constructor and pretty-print display.
- Using this as a constructor does not check the type.
- Objects have a user defined type, which is simply a word that consumes an object and marks the object as *bad*.
- Type words should be capitalized.
- An object can contain anything, but is checked by the type function.

make_obj [Coll]

- Creates an object and checks the type.
- Format is [Type [arguments] value]

obj_val [Object] Returns the object's value.
obj_val* [Object] Preserves argument.
check_obj [Object] Applies the Type function to the value → good/bad.

- *Default object type action.*

check_obj? [Object] Applies the Type function to the value → true/false.
set_obj [Object] Sets the value, then type checks it.
do_obj [Object] Evaluates the value.
do_obj* [Object] Preserves argument.
obj_args [Object] Returns the arguments.
obj_args* [Object] Preserves argument.
set_obj_args [Object] Sets the object arguments and then type checks it.
obj_type [Object] Returns the object's type.
obj_type* [Object] Preserves argument.
x_obj [Object, Word] Evaluate an object based on a type word → good/bad.
x_obj? [Object, Word] Evaluate an object based on a type word → true/false.

Null_obj / **Null_obj?** [Object]
Int_obj / **Int_obj?** [Object]
Float_obj / **Float_obj?** [Object]
Bool_obj / **Bool_obj?** [Object]
Char_obj / **Char_obj?** [Object]
String_obj / **String_obj?** [Object]
Word_obj / **Word_obj?** [Object]
List_obj / **List_obj?** [Object]
Block_obj / **Block_obj?** [Object]
Seq_obj / **Seq_obj?** [Object]
Object_obj / **Object_obj?** [Object]
Coll_obj / **Coll_obj?** [Object]
Item_obj / **Item_obj?** [Object]
Num_obj / **Num_obj?** [Object]
Alpha_obj / **Alpha_obj?** [Object]
Alphanum_obj / **Alphanum_obj?** [Object]
Wordy_obj / **Wordy_obj?** [Object]
Blocky_obj / **Blocky_obj?** [Object]
Listy_obj / **Listy_obj?** [Object]

List and String words

len [Any] Length of Collection or String; nil = 0; anything else = 1.
len* [Any] Preserves argument.
first[Coll] / [String] Returns the first item of a Coll or first Char of a String.
first* [Coll] / [String] Preserves argument.
last [Coll] / [String] Returns the last item of a Coll or first Char of a String.
last* [Coll] / [String] Preserves argument.
but_first [Coll] / [String] Returns everything except the first item or character.
but_first* [Coll] / [String] Preserves argument.
but_last [Coll] / [String] Returns everything except the last item/character.
but_last* [Coll] / [String] Preserves argument.
delist [Coll] Unpacks a collection, leaving its length as an Int on TOS.
- [1 2 3] delist → 1 2 3 3

enlist [Int] The opposite of delist. Converts n stack items into a list.
- 1 2 3 3 enlist → [1 2 3]

enlist_all [] Enlists the entire stack.
concat [Coll, Coll] / [Coll, Item] / [Item, Coll] / [Item, Item] Nulls allowed
- Takes two things and converts them into a single list.
- [1 2 3] concat [4 5 6] → [1 2 3 4 5 6]
- [1 2 3] 4 .concat → [1 2 3 4]
- concat(1 [2 3 4]) → [1 2 3 4]
- concat(1, 2) → [1 2]
- concat: nil nil → [nil nil]

append [Coll, Coll] / [Coll, Item] / [Coll, Null]
- appends the second item to the first collection
- [1 2 3] append {4 5 6} → [1 2 3 {4 5 6}]
- {1 2 3} append 4 → {1 2 3 4}

append [String, String] / [String, Char] / [Char, String] / [Char, Char]
- Creates a new string formed from the two items.
- `a append `b → “ab”
- “abc” append “def” → “abcdef”

prepend [Coll, Coll] / [Coll, Item] / [Coll, Null]
- As append, but places second item at the beginning of the collection.
- [1 2 3] prepend {4 5 6} → [{4 5 6} 1 2 3]

prepend [String, String] / [String, Char] / [Char, String] / [Char, Char]
- Creates a new string formed from the two items in reverse order.
- “abc” prepend “def” → “defabc”

insert [Coll, Any, Int] / [String, String, Int] / [String, Char, Int]
- Inserts the second item at position n (0 = beginning, negative = from end) in the first.
- [1 2 3] insert “zero” 0 → [“zero” 1 2 3]
- “abef” “cd” 2 .insert → “abcdef”

delete [Coll, Int] / [String, Int]
- Deletes the item or character at position n (0 = beginning, negative = from end).
- [1 2 3 4] delete 3 → [1 2 3]
- [1 2 3 4] delete -1 → [1 2 3]
- “abcde” delete 10 → bad(“abcd”)

extract [Coll, Int] / [String, Int]
- returns the nth Item in a Coll or Char in a String, preserving the altered Coll/String.
- Can use negative index (from end)
  - [1 2 3] extract 1 → [1 3] 2

extract* [Coll, Int] / [String, Int]
- Preserves the original Coll/String.
  - "abcd" extract* 2 → "abcd" "abd" `c

nth [Coll, Int] / [String, Int]
- Returns the nth item in a collection or character in a string.
  - [1 2 3] nth 1 → 2

nth [Coll, List]
- Returns a sub-item from a collection, selected by the list index.
  - [1 2 [3 4 [5 6] 7] 8] nth [2, 2, 1] --> 6

nth* [Coll, Int] / [String, Int] Preserves the Coll/String.
  - [1 2 3] nth* 1 → [1 2 3] 2

nth* [Coll, List] Preserves the original Coll.
  - [1 2 [3 4 [5 6] 7] 8] nth [2, 2, 1] --> [1 2 [3 4 [5 6] 7] 8] 6

<nth [Coll, Int] / [String, Int]
- alias for *extract*

>>nth [Coll, Int, Any] / [String, Int, Char] / [String, Int, String]
- alias for *insert*

>nth [Coll, Int, Any] / [String, Int, Char] / [String, Int, String]
- Replaces the item or character at the indicated position (delete then insert).
  - [1 2 3] >nth 2 "Hi" → [1 2 "Hi"]

slice [Coll, Int, Int] / [String, Int, Int]
- Produces a range of items (or substring).
- Reversing is allowed.
- Negative indexing (from end) allowed.
  - [0 1 2 3 4 5 6] 2 4 .slice → [2 3 4]
  - [0 1 2 3 4 5 6] slice 4 2 → [4 3 2]
  - "abcdefg" -2 3 .slice → "fed"

slice* Preserves the original Coll/String

>list [Any] Converts anything to a list.

>block [Any] Converts anything to a block.

>seq [Any] Converts anything to a sequence.

>string [Any] Converts anything to a string.

>word [String] Converts a string with a valid name to a word.

>char [String] / [Int]
- Converts a string of length 1 to a character, or empty strings to `\0.
- Converts an integer 0..127 to an ASCII character.

>num [String] / [Char]
- Converts a String to an Int or Float, as appropriate.
- Converts a Char `0..`9 to an Int.

ord [Char]
- Converts a character to its ASCII order number
  - `A ord → 65

num_string? [String]
- returns true if the string is a valid representation of a number.

digit? [Char]
- returns true if the character is a valid representation of a decimal digit.

reverse [Coll] / [String]
- Reverses the collection or string.

sort [Coll]
- Sorts a collection in ascending order.

zip [Coll, Coll]
- Creates a list of 2 item lists, composed of alternating items from the two original collections. The length is determined by the shorter collection. Extra items from the longer collection are discarded.
  - [1 2 3] zip {`a `b `c `d} → [[1 `a] [2 `b] [3 `c]]

zip [String, String]
- Creates a new string by alternating the characters of the original strings. The length is determined by the shorter string. Extra Chars from the longer string are discarded.
  - “abc” zip “1234” → “a1b2c3”

unzip [Coll] / {String]
- Undoes the zip process.
  - [[1 `a] [2 `b] [3 `c]] unzip → [1 2 3] [`a `b `c]
  - “a1b2c3Q” unzip → “abc” “123”

range [Int, Int] / [Char, Char]
- Creates a list beginning with the first Int/Char, ending with the last.
- Ascends/descends as appropriate.
  - 1 5 .range → [1 2 3 4 5]
  - `c range `a → [`c `b `a]
  - 1 range 1 → [1]

range< [Int, Int] / [Char, Char]
- as range, but omits the ending Int/Char
  - 1 range< 5 → [1 2 3 4]
  - `d range< `a → [`d `c `b]
  - 1 range< 1 → []
  - `a range< `b → [`a]

... [Int, Int] / [Char, Char] *immediate*
- special infix syntax for range
  - 1 … 5 → [1 2 3 4 5]

..< [Int, Int] / [Char, Char] *immediate*
- special infix syntax for range<
  - `a ..< `d → [`a `b `c]

in [Item, Coll] / [Coll, Coll] / [Char, String] / [String, String]
- returns true if the first thing is contained in the second.
  - 1 in [1 2 3] → true
  - “QWE” in “abcde” → false

in* Preserves original Coll/String

<-in Reverses the order of arguments

<-in* Reverses the order of arguments and preserves original Coll/String

where [Item, Coll] / [Coll, Coll] / [Char, String] / [String, String]
- returns the location if the first thing is contained in the second, -1 otherwise.
- 1 where [1 2 3] → 0

where* Preserves original Coll/String
<-where Reverses the order of arguments
<-where* Reverses the order of arguments and preserves original Coll/String
string>list_char [String]
- “Hello” str>list_char → [`H `e `l `l `o]

string>list_word [String]
- “hello there” str>list_word → [“Hello”, “there”]

string>list_word_sep [String, Char] / [String, String]
- “hello:there” `: str>list_word_sep → [“hello”, “there”]

list_char>string [Coll]
- [`h `e `l `l `o] list_char>string → “hello”

list_word>string [Coll]
- [“Hello” “there”] list_word>string → “Hellothere”

list_word>string_space [Coll]
- [“Hello” “there”] list_word>string_space → “Hello there”

list_word>string_sep [Coll, Char] / [Coll, String]
- [”Hello”, “there”] “ “ .list_word>string_sep → “Hello there”

+ [String, String] / [String, Char] / [Char, String] / [Char, Char]
- alias for *append*, creates a new string.

+ [Char, Int]
- Increments the character by the number.
- `A + 1 → `B

+ [Int, Char]
- Increments the number by the order of the character.
- 1 + `A → 66

inc [Char]
- Increment by one.
- `A inc → `B

- [Char, Int]
- Decrements the character by the number.
- `B - 1 → `A

- [Int, Char]
- Decrements the number by the order of the character.
- 66 - `A → 1

- [Char, Char]
- Subtracts the ASCII order of the second character from the first, returning an Int.

dec [Char]
- Decrement by one.
- `B dec → `A

***** [Char, Int] / [String, Int]
- Repeats the character or string n times, returning a new string.
- `X * 3 → “XXX”

empty [Coll] / [String]

- Empties a collection or string to [], {}, (), or “”

flatten [Coll]

- Eliminates one nesting layer of a collection.
- [1 2 [3 4 [5] 6 [7] 8] 9] flatten → [1 2 3 4 [5] 6 [7] 8 9]

stomp [Coll]

- Eliminates all nesting layers of a collection.
- [1 2 [3 4 [5] 6 [7] 8] 9] flatten → [1 2 3 4 5 6 7 8 9]

resolve [Coll]

- Evaluates each variable in a collection.
  - 42 @>a {1 2 a} resolve a → {1, 2, 42} 42
  - 42 @>a [1 2 @<a] .resolve a → [1, 2, 42] 42
  - 42 @>a resolve: [1 2 a .+] a → [1, 2, 42 , .+] 42
  - 42 @>a {1 2 3 @>a 4} resolve a → {1 2 4} 3
- To evaluate everything in a list like a program, use: map \eval
  - 42 @>a [1 2 3 a .+ .* swap] map \eval → [90, 1]
  - 42 @>a [1 2 3 a + * swap] map \eval → [90, 1]
    - Evaluation of prefix and infix words with insufficient arguments (on the data stack and in the command queue) converts them in an attempt to find a form that can be executed.
      - prefix → infix → postfix
  - 42 @>a {1 2 3 a + * swap} eval → 90 1
  - 42 @>a 1 2 3 a + * swap → 90 1
    - The result is similar because + and * swap with each other twice.
      - + * → * .+ → .+ .*

Math

+ [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Int + Int → Int, otherwise Float.

- [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Int – Int → Int, otherwise Float.

***** [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Int * Int → Int, otherwise Float.

/ [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Always returns a Float.

// [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Integer division, truncates Floats before dividing, always returns an Integer.
  - 5.0 // 2.5 → 2

% [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Modulus, truncates Floats, always returns an Int.
  - 5.0 % 2.5 → 1

/% [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- "divmod", combined integer division and modulus.
  - 5 /% 2 → 2 1

pow [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- a to the power of b, always returns a Float
  - 2 pow 3 → 8.0

^ [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- Integer power
  - 2 ^ 3 → 8

root [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- b^{th} root of a, always returns a Float
  - 4 root 2 → 2.0

log [Num, Num] / [Num, Blocky] / [Blocky, Num] / [Blocky, Blocky]
- log base b of a, always returns a Float
  - 8 log 2 → 3.0

abs [Num] [Blocky] absolute value

negate [Num] [Blocky] negation

round [Num] [Blocky] round normally
- Float → Float, Int → Int

ceiling [Num] [Blocky] round up
- Float → Float, Int → Int

floor [Num] [Blocky] round down
- Float → Float, Int → Int

trunc [Num] [Blocky] round towards zero
- Float → Float, Int → Int

>round [Num] [Blocky] round normally to Int

>ceiling [Num] [Blocky] round up to Int

>floor [Num] [Blocky] round down to Int

>trunc [Num] [Blocky] round towards zero to Int

>int [Num] [Blocky] truncates to Int

>float [Num] [Blocky]

sqrt [Num] [Blocky] square root

sqr [Num] [Blocky] square
cbrt [Num] [Blocky] cube root
ln [Num] [Blocky] natural log
exp [Num] [Blocky] e to the power of a (inverse of natural log)
>deg [Num] [Blocky] radians to degrees
>rad [Num] [Blocky] degrees to radians
sin [Num] [Blocky] trig functions
cos [Num] [Blocky]
tan [Num] [Blocky]
sec [Num] [Blocky]
csc [Num] [Blocky]
asin [Num] [Blocky]
acos [Num] [Blocky]
atan [Num] [Blocky]
asec [Num] [Blocky]
acsc [Num] [Blocky]
sinh [Num] [Blocky]
cosh [Num] [Blocky]
tanh [Num] [Blocky]
sech [Num] [Blocky]
csch [Num] [Blocky]
asinh [Num] [Blocky]
acosh [Num] [Blocky]
atanh [Num] [Blocky]
asech [Num] [Blocky]
acsch [Num] [Blocky]

euler / **Euler** []	constant	2.718281828459045	e, basis for ln and exp
pi []	constant	3.141592653589793	
half_pi []	constant	1.570796326794897	
tau []	constant	6.283185307179586	twice pi
phi []	constant	1.618033988749895	golden ratio
sqrt2 []	constant	1.414213562373095	square root of two
half_sqrt2 []	constant	0.7071067811865476	half the square root of two

max [Alphanum, Alphanum]
min [Alphanum, Alphanum]

Input, Output & Errors

clear_screen []
print [Any] Print the item to the screen.
println [Any] Print followed by return.
pprint [Any] Pretty print the item. Color coded print format.
pprintln [Any] Pretty print followed by return.
emit [Int] / [Char] Print a single character.
sp [] Prints a single space. (32 emit)
cr [] Prints a single carriage return / newline. (10 emit)

get_line [] Input a line until the enter key is pressed.
get_char [] Input a single key press.
get_line_silent [] Input a line without showing it on the screen. Good for passwords.
get_char_silent [] Input a single key without showing it on the screen.

set_err [String] Create an error message.
get_err [] Move the latest error message to TOS as a String.
copy_err [] Copy the latest error message to TOS as a String.
print_err [] Print the most recent error message.
print_errors [] Print all error messages.
drop_err [] Delete the most recent error message.
clear_err [] Alias for drop_err.
clear_errors [] Delete all error messages.
count_errors [] Push the current number of error messages on TOS as an Int.
debug_on [] Turns on debug mode.
debug_off [] Turns off debug mode.

import [String] Import, parse, and execute a file.

- Listack automatically adds "_set_namespace *filename*" (without the .ls) to the beginning and "_reset_namespace" to the end.

coll_safety_checks [Bool] Default = true

- Directly sets the underlying representation (flexdeque) environment variable.
- Setting to false will speed execution, at the cost of not checking bounds and empty status on collections, including the stack and command queue.
- Use with extreme caution.

coll_sanity_checks [Bool] Default = false

- Directly sets the underlying representation (flexdeque) environment variable.
- Setting to true will slow execution.
- Use when a collection's representation may be corrupted.

Miscellaneous Words

timer_start [] *immediate*	Begin a timer.
timer_check [] *immediate*	Reports the number of nanoseconds since timer_start.
type [Any]	Checks the type of the item on TOS. Returns a String.
type* [Any]	Preserves the argument.
nil []	The only object of type Null.
none []	>bad(nil).
>good [Any]	Forces TOS to be good (not bad).
>bad [Any]	Forces TOS to be bad.
nop []	No operation. Does nothing, but takes up space.
****	Defer the following item. Generally handled by the parser.
defer [Any] *immediate*	Defer TOS.
words [] immediate	Prints all words in all namespaces.

Program Examples

```
# Computing the summation of 0..number

def: "summ" [Int]
  {dup <=>     # would be a bit faster if this were postfix
    {dup {1 .+ dup 0 .<} {dup roll .+ swap} .while}   # negative
    {dup}                                              # zero
    {dup {1 .- dup 0 .>} {dup roll .+ swap} .while}    # positive
  drop  }
timer_start 100_000 summ timer_check          # timer works in nanoseconds
"summ(100,000) time: " print 1_000_000 .// print " ms" println
println cr

def: "fast_summ" [Int]
  { dup # <=>          This is a good way to indicate what's going on with postfix words.
    {dup 1 .- .* -2 .//}
    {nop}
    {dup 1 .+ .* 2 .//}
  .<=> }
timer_start 100_000 fast_summ timer_check
"fast_summ(100,000) time: " print 1_000 .// print " us" println
println


# Examples of different ways to do the same thing in Listack.

10 @>n         # set local variable n to 10
# The following all set the local variable a to (n+1)*2 → 22.
n 1 .+ 2 .* @>a
n + 1 * 2 @>a
(n + 1 * 2) @>a
(n + 1) * 2 @>a
n.inc * 2 @>a
set('a', (n + 1 * 2))
set: `a (n + 1 * 2)
"a" set (+: n 1 * 2)
\a (n 1 .+ 2 .*) .set
set(\a, *(+(n, 1), 2))
set(\a *(+(n 1) 2))
\a set (+: @<n 1 2 .*)
\n.get 1 .+ 2 .* @>a
{n + 1 * 2} eval @>a
+: @*n 1 * 2 \a swap .set
# And many more, with increasing levels of obfuscation.
```

The actual definitions of *while*, *until*, and *times*:

```
# while
create_global: "_while_cond"
create_global: "_while_body"
# internal use only
def: "_while_" []
  {@!_while_cond
  {@!_while_body _begin_loop_
   _while_}
  .iff}

def: "while" [Blocky, Blocky]
  {@>_while_body @>_while_cond
  _while_ _end_loop_
  @<_while_body drop @<_while_cond drop}

def: "while" [Bool, Blocky]
  {@>_while_body @>_while_cond
  _while_ _end_loop_
  @<_while_body drop @<_while_cond drop}

def: "while" [Bool, Word] {>block .while}
def: "while" [Blocky, Word] {>block .while}


# {body} until {condition}
def: "until" [Blocky, Blocky]
  { {not} .concat @>_while_cond @>_while_body
  @!_while_body _begin_loop_ _while_ _end_loop_}
      # always executes body at least once
      # {body to execute} until {condition to terminate}
      # {println:"infinite loop"} until {false}

def: "until" [Word, Blocky] {swap >block swap .until}
def: "until" [Blocky, Bool] {>block .until}
def: "until" [Word, Bool] {>block swap >block swap .until}

# {body} times count
create_global: "_times_counter"
def: "times" [Executable, Int]
  {dup 0 .>   # if
    {@>_times_counter {@<_times_counter dec dup @>_times_counter 0 .>=} swap .while
      @<_times_counter drop}
    {drop2} .if}

def: "times" [Int, Executable] {swap .times}
# 5 times \print(“Hi! ”) cr
```