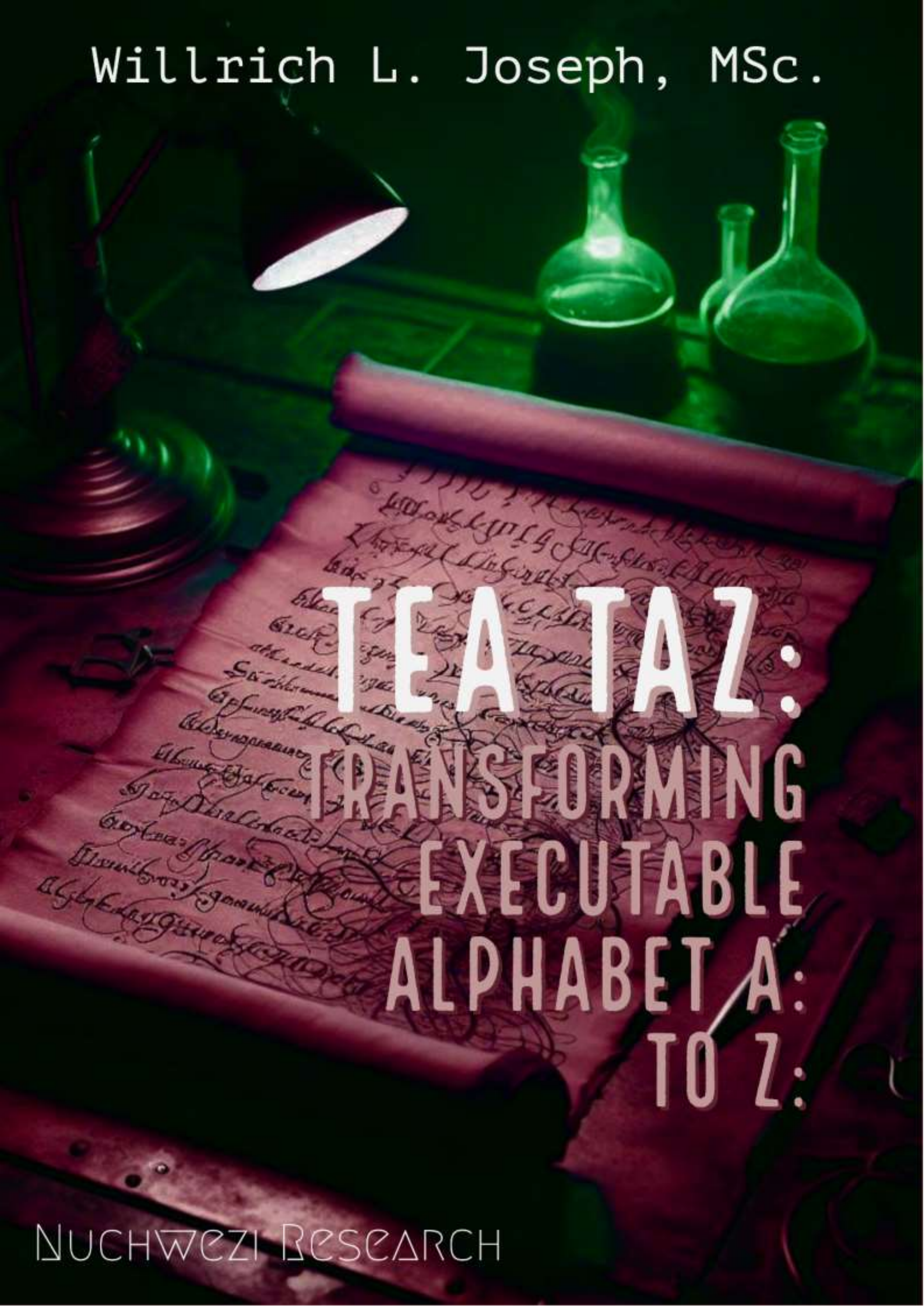
Willrich L. Joseph, MSc.

TEA TAZ: TRANSFORMING EXECUTABLE ALPHABET A: TO Z:



Nuchwezi Research

TEA TAZ – Transforming Executable Alphabet A: to Z: COMMAND SPACE SPECIFICATION

A formal introduction to the TEA Computer Programming Language

UPDATED ON: 20th SEPT, 2025

AUTHOR: Joseph L. Willrich (**jwl@nuchwezi.com**) as part of PhD work [2]

ABSTRACT: *This manuscript builds upon the earlier TEA language formalization & specification work in Joseph's PhD research diary [1]. For all practical purposes, this document is best treated as* ***a living document****; it is continually being enhanced as TEA evolves, but must also be considered the* ***official, authoritative formal reference*** *on matters concerning the definition, grammar, semantics, and processing of TEA programs.*

For the rest of this document, the following definitions & clarifications are important:

TEXT: *In the TEA computer programming language, Text is considered to be any form of data a TEA program can process and reason about. All TEA programs process only Text.*

Strings: *Let us assume a finite sequence of distinct characters from a finite set such as Unicode-8 or ASCII. This is the alphabet understood and processed by TEA programs. Let us call any such finite sequence, for example {a,b,c,d, ,f} the string "abcd f", and for TEA programs, we shall typically write an explicit string---such as "a b c", as an expression expressing the exact character and position---thus order, it occupies in the string, by either the common syntax "a b c" (such as some programming languages do… Java, Python and C), but for TEA, we shall also allow, in-fact, recommend that we express all strings in a program source-code using the earlier syntax; {abc f}. For TEA programs, all data is presented as Text, and at the source-code or even run-time level, data being processed by a TEA program is expected to be, and is treated as a string.*

Regular Expressions*: Because TEA is a Text Processing language at core, it means, advanced text processing power and capabilities need be built within the language by design. Among these is the ability to automatically discover patterns in strings and then do things based on or to them. A kind of intelligent or controllable and directable processing. For pattern matching and pattern-based conditional processing, TEA programs employ the concept and mechanics of Regular Expressions. For clarification purposes while reading this TEA specification as well as future literature and TEA source code built based on this standard, TEA regular expressions are to be written in a TEA program without any explicit delimiters except the standard TEA Parameter Expression Delimiter (refer to* Figure 1*) ":" ---to defer from strings. Thus, where it is expected to write an explicit regular expression (also typically referred to as a REGEX in this specification) such as ^$ to denote the REGEX used to match the empty string, typically expressed as "", shall likewise be written as ^$ when being expressed within a TEA program source code. However, the same, when being expressed or being passed around in a TEA program, shall instead be written as {^$} or perhaps "^$". Of course, it would have been possible to actually write both explicit strings and regular expressions using the same simple and bare syntax---for example, with the discarding of the string delimiting characters " and " for the typical, and { and } for TEA, but sometimes it is safer to sacrifice mathematical elegance while writing a program, and instead secure useful program source-code properties such as readability, program comphrensibility and lexical correctness – for both the human writing the code – such as we expect most TEA programs shall be, and the machines meant to read, parse and process things based on human-written TEA program source code.*

AI: Active Input --- This refers to the main input or data to be referred to or processed by the current TEA Program instruction at the time it is being evaluated.

IO: Instruction Output --- This refers to the main output or data to be returned by the current TEA Program instruction after it is fully executed.

Word: This refers to a sequence of non-whitespace characters

TEA Primitive: Also the same as "TEA Primitive Command" or "TEA Canonical Command", is any one of the letters in the Latin Alphabet, a to z or A to Z, followed by a single colon ":" character, and the letter used determines what purpose a TEA instruction has in a program. Also, each primitive has unique semantics as defined in the TEA Command Space specification. It is important to note that in TEA, much as the standard style is to use lowercase letters for TEA primitives such as "a:" or "m:", yet, case doesn't matter, and "a:" and "A:" are basically equivalent, but "a:" and "A!:" aren't, much as they reference the same basic TEA primitive "a". In a TEA primitive such as "a:", we may refer to the command letter "a" as the "Command Character".

TEA Inverse: When any of the TEA primitives such as "a:" has the command character followed by a TEA Command Qualifier such as the exclamation mark "!" or star-character "*", such as with "a!:" or "g*:", it is then considered to be the inverse or alternate form of the implied primitive. So, "a!:" is the alternate form of the command "a:", and unless where specified, typically the canonical form of a TEA primitive, such as "a:" has different effects and purpose from its inverse form "a!:"

INERT or UNDEFINED TEA Command: When a TEA Command is flagged as or defined as "INERT" or "UNDEFINED" or "RESERVED", it means that command or its implied form has no effect in a standard TEA program, and can be ignored by the TEA processor when the program is being executed. Typically, this occurs with the special treatment of Inverse forms of a TEA primitive, such as when "a:" is defined, but "a!:" is not or when "a:" is, but "a:WITH PARAMETERS" isn't. Typically, where for example a primitive such as "a:" is defined but its parameterized form isn't, it could be safe to assume that "a:WITH PARAMETERS" shall simply be ignored and not have effect in the program, much as "a:" is defined and would cause an effect in the program. In an advanced TEA program environment, using or writing an INERT form of a TEA command should either be flagged or reported as an error. Otherwise, typically, the safe judgment to make concerning INERT commands is that they shouldn't and won't modify or affect the AI, and thus, should transparently return the AI as IO, and thus can be considered to be non-existent in a TEA program.

1 THE TEA LANGUAGE DEFINITION

The Transforming Executable Alphabet (TEA) language, is a formal language specified by the following grammar, and which is then used to express automatons, or rather, computer programs, capable of running on any Turing Machine or Abstract Machine capable of interpreting or processing the TEA language.

TEA Grammar

Essentially, all TEA programs conform to the following simplified syntax template specified using the formal language of Regular Expressions:

```
([a-zA-Z]*?!?:.*(:.*)*|?)+(#.*)*
```

Figure 1 TEA Instruction Grammar

Essentially, we see here, the final, most generic specification of any legitimate TEA Instruction (TI), the implication is that a TEA program consists of one or more TI – with or without TEA comments (more about this later). We see that a TEA Instruction obeys the following syntax rules:

1. The instruction starts with a single letter from Latin alphabet, and that the case of the letter doesn't matter. This letter is what is called a ***TEA Primitive Command*** (TPC).
2. After the TPC, we might optionally have an exclamation mark (!) and or an asterisk *--- *TEA* ***Command Qualifiers*** (TCQ), and nothing else after the TPC but the full colon (:)---a *TEA* ***Command Delimiter***(TCD). When the TPC is followed by TCQ we then call that command the **Inverse** or **Alternate** form of the TPC.
3. After the TCD, everything that follows until the end of the line or until the vertical bar character (|)---the *TI Delimiter* (TID) (earlier ideas had included the possibility of delimiting multiple TI expressions, possibly on the same line, using either the (;) or (,) characters)---is a ***TI Parameter Expression*** (TIPE).
4. TIPE consists of one or more characters excluding the **TIPE Delimiter** (TIPED) symbol--- *also called "**TEA Parameter Expression Delimiter**"*, which is the full colon, ":", just like the TCD, followed by one or more TIPE.
5. After the TID, and on the same line, everything that follows is either another TI or is something essentially treated as either whitespace or a comment---thus a ***TEA Opaque Expression*** (TOE).
6. Taken together, the TCP•TCQ•TCD specify a ***TEA Command*** (TC).
7. When a line in a TEA program doesn't start with a valid TC with or without leading white space, such a line is treated as or interpreted as TOE.
8. All TOE in a TEA program are essentially ***TEA Comments*** (TCOM), and aren't processed by the TEA interpreter.

In summary, a TI can be produced thus:

```
TIL := WS*•TI•TI*•TOE•EOL

WS := White Space

TI := WS*•TC•TIPE•TID
```

```
TC := TCP•TCQ•TCD
TCP := [a-zA-Z]
TCQ := ! | * | *!
TCD := :
TIPE := NTIPED•(TIPED•NTIPED*)*
NTIPED := [^:]*
TID := |
TOE := NEOL* | TCOM
NEOL := [^\n]*
TCOM := #NEOL
EOL := NLC | NLC•CR
NLC := New Line Character
CR := Carriage Return
```

Where TIL is a ***TEA Instruction Line***, and thus a ***TEA Program*** (TP) can be fully produced thus

```
TP := TIL • (TOL* • TIL*)*
TOL := TOE • EOL | TCOML
TCOML := WS* • TCOM • EOL
```

Where TOL is a ***TEA Opaque Line***---essentially a line in a TEA program that can be entirely ignored by the processor or interpreter. Thus do we now have a full, and perhaps complete specification of the TEA programming language syntax. This should help with lexing TEA program source code, and thus parsing TEA Programs. However, as for the semantics of any given TEA program, it is important to combine knowledge of valid TEA program syntax, as well as the valid syntax and semantics of each individual TEA primitive command space. This is specified in the TEA A: to Z: Command Space Specification section of the TAZ.

1.1 Parsing and Processing Multi-line TI and Multiple TI on a single line

The minimal TEA language grammar defined in the previous section might not readily capture or express one small quirk about how TEA programs might be written in practice – such as when TI spans more than one line – an example being when a string parameter being passed to an instruction in the source code has to span multiple lines, or when a TEA comment needs do the same. The other quirky case is when multiple TI need be expressed on a single line – something which might not be immediately obvious by merely looking at the TEA language grammar.

An example TEA program that highlights these syntactic quirks follows…

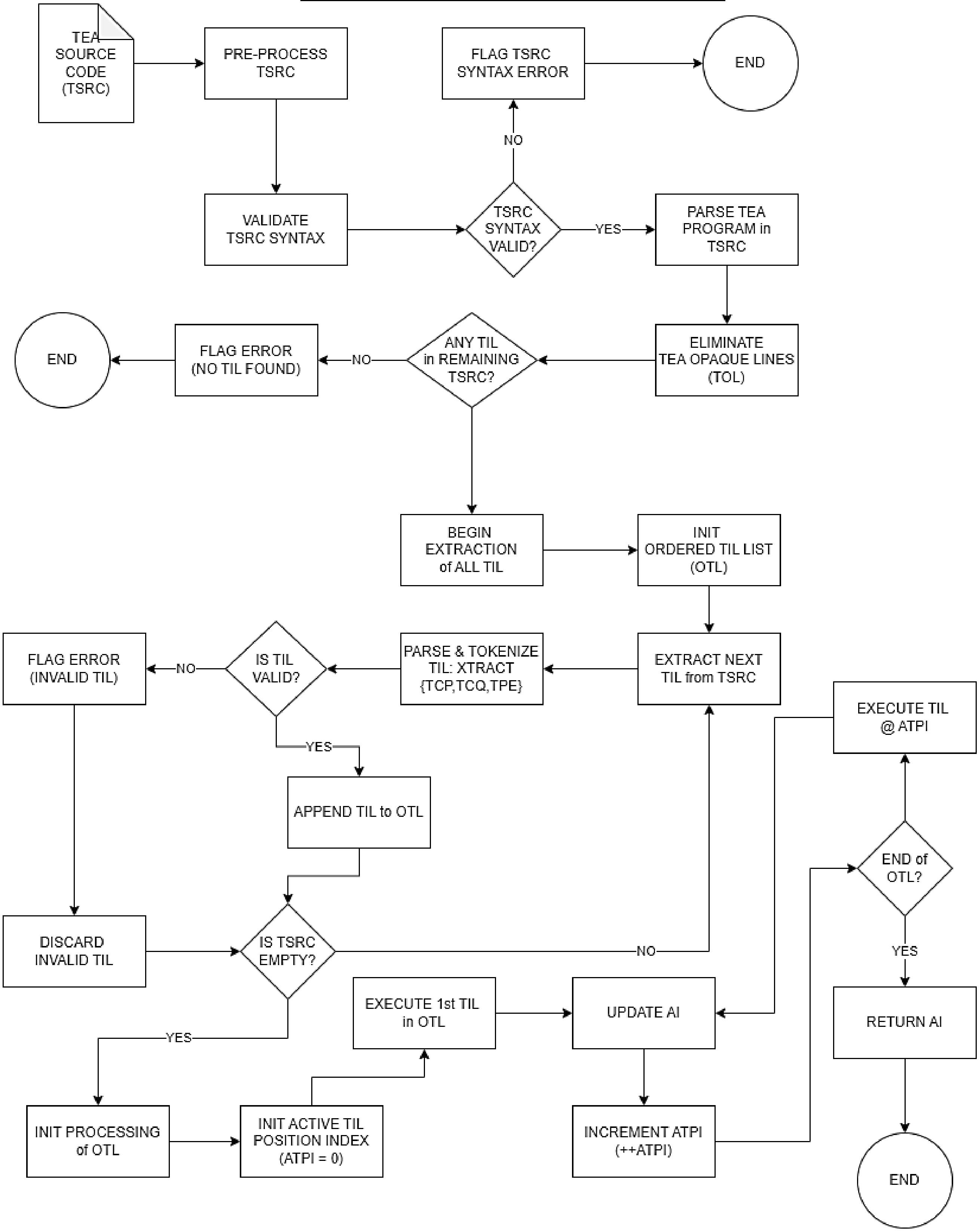
```
i: {This is a multi-line
string} | # followed by comment
u!: | g:
l:E | x:"1-"
f:^1-i:A:B | l:A | x!:-1 | j:C | l:B | i!:{T} | j:E
l:C | q!:
#(=1-istlnre-mgauTh-1)
```

In parsing such a program relative to the given TEA grammar, one might approach the task thus:

1. Have a TEA Instruction Line list to hold each complete TEA Instruction per entry, in the natural order they appear in the code.
    a. To help with dealing with Multi-line strings anywhere within the program, and which are the only reason parsing might become tricky, start by turning all such Multi-line instructions into single line forms. For example, given any such instructions shall be the kind involving Multi-line strings which are mandatorily delimited by either " & " or { & }, then find a means to momentarily substitute newline characters within such explicit strings in the code with some special marker such as a rare character RC. Thus, any such previously Multi-line strings shall take the modified form {…RC..RCRC…} After this transformation, we can comfortably proceed to process the source code as though all instructions either span a whole single line or multiple instructions sit on the same line (delimited of course).
2. Start by splitting the modified code by newlines
3. For each line, check for whether the line is an opaque line or a TEA Instruction Line
4. If opaque ignore and move to the next, otherwise extract the TEA instructions on that line as follows
    a. If the line is an instruction line with only a single TEA Instruction on it, reverse the RC-NL transform above if necessary, then add the instruction to the instruction list. Otherwise if multiple instructions exist on the line (delimited by |), then extract each instruction statement and add it to the instruction list in the exact order in which it appears on the line.
    b. Finally, once all the instructions have been extracted and stored in an ordered list, perhaps with clear annotation for what kind of instruction it is (for example noting label statements since they shall merely serve for control flow), then proceed to execute the TEA program by operating on the list of instructions.

Generally, one might appreciate the simplicity of parsing, validating and then executing TEA program source code by studying the TEA execution process as depicted in the following flowchart:

TEA PROGRAM EXECUTION PROCESS



2 THE TRANSFORMING EXECUTABLE ALPHABET COMMAND PRIMITIVE NAMES

First, let us look at the current reference list of the 26 TEA primitives and their formal names

In the rest of this manuscript, we then fully define each of these primitives, with focus on what purpose each primitive serves in a TEA program, what syntax it expects, as well as the function and semantics associated with it. For best clarity, each primitive shall be treated on its own page, but the approach and structure of the specification remains the same across all the 26 TEA primitives.

3 Concerning the TEA Command Naming Standard

All TEA commands are essentially verbs - they tell the TEA processor to do something, but also, based on the name and expression of the verb, they also specify or hint at how to do that thing or what exactly to do or not do. Thus, in choosing names for the TEA Instruction Set primitive commands – which, though they might already be easy to call by the names of their constituent TEA primitive letters “a” to “z”, would better be named suitably to distinguish them from ordinary Latin alphabet letters, and also to help reflect or clarify on their function in TEA. The following 4 guiding principles serve that purpose:

1. The Command Name must start with the same letter as the TEA command for which it is a name.
2. The Command Name must reflect or hint to the verb or action the command is designed to do, and not be ambiguous.
3. The Command Name must be a single word, preferably in English.
4. The Command Name must be unique across the instruction set, but this already follows from condition #1 in this list.

Thus, some tricky TEA primitives such as **W:** might perhaps better be named "Webify" instead of original proposal to use "Web", and for **U:** to be called "Uniqueify" instead of "Unique". But, condition #2 somewhat helps relax or dismiss the need for these renamings if they seem too extreme. However, with the Instruction Set as expressed above, we can boldly say each of the 26 primitive instructions in TEA clearly and non-ambiguously define not only what each instruction does, but also how. Just by looking at the Command name, or even just the command's first letter... Which surely is not just a clean design choice, but also shall serve to make learning, reading and applying TEA programs very easy. Of course, unlike many of not most existing programming languages, the TEA language can boast of having the simplest instruction set, and also one that's very precise in semantics and purpose, and which, with the use of mnemonics such as reading and memorizing the TEA command name map above, becomes readily palatable and learnable even for little kids just learning the alphabet.

4 TEA A: TO Z: COMMAND SPACE SPECIFICATION

In the rest of this section, we shall look at the full formal specification as well as some explanatory and or illustrative notes concerning each one of the 26 A: to Z: TEA primitives, one at a time. For historical purposes, the only other authoritative existing reference material in relation to the TAZ is in a couple of exploratory lectures [4] that the language's inventor gave mostly before this manuscript was prepared.

TEA PRIMITIVE	SEMANTICS	
A:	**NAME**	**Anagrammatize**
	PURPOSE	Compute anagrams
	SYNTAX & SEMANTICS	**a:** Set IO as the AI anagrammatized by words **a:STR** Same as a:, but operating on the string STR **a!:** Set IO as the AI anagrammatized by characters **a!:STR** Same as a!:, but operating on string STR **A*:vNAME** **A*!:vNAME** The first as a:, the second as a!:, but operating on string in vault vNAME
	NOTES	By the definitions above, the program: i!: "BC CB BA AB" | a: could return "CB BA AB BC" – note that **a:** basically shuffles the words in the AI, while the program i!:{BC CB BA AB} | a!: could return " ABBCB ACB" because **a!:** shuffles the contents of the AI

TEA PRIMITIVE	SEMANTICS	
B:	NAME	**Basify**
	PURPOSE	Compute the Lexical Base
	SYNTAX & SEMANTICS	**b:** Set IO as AI reduced to only its unique characters in their order of occurrence within AI **b:STR** Same as b:, but operating on string STR **b!:** Same as b: but with the results sorted in alphabetical order **b!:STR** Same as b!: but operating on STR **b*:vNAME** **b*!:vNAME** The first as b:, the second as b!:, but operating on string in vault vNAME
	NOTES	The **Lexical Base** of the AI is to be understood as the reduction of the AI to a string made of only the **unique** characters in AI, and these, then sorted in ascending **Lexical Order** for B-Inverse otherwise occurring in their order of occurrence. By the definitions above, the program: i!:{BC CB BA AB} | b: Should return "BC A". While i!:{bC CB BA aB} | b!: Should return "ABCab". But i!:{bC CB BA aB} | b: Shall return "bC BAa".

TEA	SEMANTICS
TEA PRIMITIVE	**SEMANTICS**
D:	(see below)

NAME	**Delete**
PURPOSE	Delete something from the AI
SYNTAX & SEMANTICS	**d:** INERT **d:REGEX** **d:RX1:RX2:…:RXN** Delete from AI all sections matching the single regular expression REGEX or any of the given regular expressions RX1, RX2,…, RXN **d!:** Delete all white-space from the AI (same as g:) **d!:REGEX** **d!:RX1:RX2:…:RXN** The Inverse of d:, for which only **sections not matching** the given patterns are deleted from AI. **d*:vREGEX** **d*:vRX1:vRX2:…:vRXN** Like the d:REGEX and d:RX1:…:RXN except, referencing the patterns stored in the named vaults. **d*!:vREGEX** **d*!:vRX1:vRX2:…:vRXN** Like the d!:REGEX and d!:RX1:…:RXN except, referencing the patterns stored in the named vaults.
NOTES	One can appreciate **D:** by the following illustrative examples: i!: "bC CB BA aB" | #(="bC CB BA aB") d:[aA] | #(="bC CB B B") But i!: {bC CB BA aB} | #(="bC CB BA aB") d:aA | #(="bC CB BA aB") because no pattern "aA" While i!: {bC CB BA aB} | #(="bC CB BA aB") d!: | #(="bCCBBAaB") And i!: {bC CB BA aB} | #(="bC CB BA aB") d:[aA] | #(="bC CB B B") d:.B | #(="bC") Or rather i!: {bC CB BA aB} | d:[aA]:.B | #(="bC") Note that parameterized inverse of the Delete command makes implementing powerful complex filters easier. For example, we eliminate anything in AI that isn't a punctuation mark using the following terse program: d!:[.,?;:]

	Otherwise, without c*!:vA, such as in the following version:

i!:{BC} | v: | c: | y: | a!: | v:vA | n:100 | v:vB | v:vGLUE:** | g*!:vGLUE:vA:vB

should have return something like CB**N or BC**N

TEA PRIMITIVE	SEMANTICS	
E:	NAME	Evaluate
	PURPOSE	Process memory as though it were a TEA program

TEA PRIMITIVE	SEMANTICS	
F:	NAME	Fork

TEA PRIMITIVE	SEMANTICS	
G:	NAME	Glue
	PURPOSE	Bind elements in AI using something
	SYNTAX & SEMANTICS	**g:**
		Reduce AI to a string with all whitespace characters removed. Essentially binds any word in AI to all the rest.
		g:GLUE **g:GLUE:REGEX**
		Set the IO to the result of reducing AI to a string where all instances of REGEX have already been replaced by GLUE. Without the second parameter – REGEX, g: merely replaces all whitespace in AI with GLUE.
		g!:
		INERT
		g!:GLUE
		Reduce AI to a string where all standard sentence punctuation marks, in addition to all whitespace, have been replaced with GLUE. GLUE is expected to be a string. The result of this transform then becomes IO.
		g*:GLUE:v1:v2 **g*:GLUE:v1:v2:v3:...:vN**
		Set IO to the result of joining the string stored in the vaults with the given names, using the specified EXPLICIT GLUE
		g*!:vGLUE:v1:v2 **g*!:vGLUE:v1:v2:v3:...:vN**
		Set IO to the result of joining the string stored in the vaults with the given names, using the glue stored in vault vGLUE
	NOTES	G: is the most magical primitive in TEA. g: for example automagically sucks all space out of things and sets them aside for later use! Let's look at an example: i!:{BC CB BA AB}| g: #(=BCCBBAAB) i!:{BC CB BA AB}| g:{_*_} #(=BC_*_CB_*_BA_*_AB) That's not without power. On the other hand, g!: can best be appreciated with examples processing regular human-readable text---which, is for example expected to have natural use of both whitespace and punctuation marks in it. Thus, the program i!:{Which of this, that or both do you want? None} | g!:{*} Should return "Which*of*this**that*or*both*do*you*want**None"
		h:

Other examples with their expected outputs shown in the comments:

TEA PRIMITIVE	SEMANTICS	
H:	**NAME**	**Hew**
	PURPOSE	Explode or split up AI using something
	SYNTAX & SEMANTICS	**h:** Split up AI such that all its elements are separated by a single space. In effect, padding the inner contents of AI with space. That becomes IO. **h:REGEX** Same as h:, but the splitting happens only at the beginning of where the contents of AI match REGEX. The original contents of AI remain preserved by this operation. **h!:** Same as h: but instead using the New Line character as separator. **h!:REGEX** Same as h!:, but the splitting happens only at the beginning of where the contents of AI match REGEX. **h*:** INERT **h*:vNAME:vREGEX** **h*!:vNAME:vREGEX** The vault operating versions of h: and h!: perform corresponding transformations, but instead operate on the data stored in the vault with the specified vault name, vNAME.
	NOTES	Unlike g:, h: is meant to help with fragmenting things. H: splits the AI by the Empty String, essentially seeming as though it has exploded, then rejoins each character to the next using a single space or new line (for the inverse). Some examples follow: i!:{ABC}| h: #(="A B C") Note that for a string of length N, h: will return a string of length 2N-1. Some examples follow… i!:{123}| h!: #(= "1 2 3") i!:{http://127.0.0.1/path}| h:[\/\.] #(="http: / /127 .0 .0 .1 /path") i!:{1234567890}|v:vIN|v:vHEW:[02468]|h*:vIN:vHEW #=1 23 45 67 89 0

TEA PRIMITIVE	SEMANTICS	
I:	**NAME**	**Interact**
	PURPOSE	Explicitly set the AI
	SYNTAX & SEMANTICS	**i:** Using the current AI as the prompt, prompt for and set whatever is the user-provided input---at runtime, as AI **i:VALUE** Only if AI is currently empty or unset, then set it to VALUE **i!:** Unconditionally set the AI to the EMPTY STRING **i!:VALUE** Unconditionally set the AI to the provided VALUE **i*:** Using the current AI as the visual prompt (to be rendered in a separate PLAIN TEXT/HTML rendering window with the input prompt displayed at the bottom), prompt for input from the user at runtime. Input returned set as the AI **i*:vPROMPT** With vPROMPT, same as i*:, but using text sourced from the vault vPROMPT or just the literal text vPROMPT if the vault doesn't exist. **i*!:** **i*!:vPROMPT** Without parameter, as i*:, but with no prompt message except just the visual prompt window. User input returned as AI. With vPROMPT, same as i*:vPROMPT
	NOTES	It is important to note that under standard TEA environments, it is possible for a TEA program to be invoked with a user or externally provided Active Input. In those cases, if the canonical form of **i:** instruction is used, with or without value, it doesn't make any changes to AI unless the command is used at a moment in the program where AI is essentially either empty or unset. For example, using the standard TTTT TEA operating environment on the command line [3], the following program if invoked thus tttt -i "ABC" -c "i:{XYZ} | q:XYZ | x!":-OK Shall return "ABC-OK" instead of "XYZ". Otherwise tttt -i "ABC" -c "i!"":{XYZ} | q:XYZ | x!"":-OK" Which, despite the queer command line invocation syntax---this example was adapted from a test via the Linux/GNU Bash Shell [6]. The essential TEA program itself is actually: i!:{XYZ} | q:XYZ | x!:-OK So, that, or even

bash -H -c 'set +H; tttt -i TEST -c "i!:{XYZ} | q:XYZ | x!:-OK"'

TEA PRIMITIVE	SEMANTICS

J:

NAME	**Jump**
PURPOSE	Jump across the TEA program
SYNTAX & SEMANTICS	**j:** INERT **j:LABEL** Unconditionally jump to the location in the program under the label LABEL. **j!:** Return to the Start of the TEA Program **j!: PARAM** INERT (or rather, *don't jump!*)

This command is one of the few flow-control instructions in a TEA program (the others are f: and q:)

NOTES

j!: (like q!:) is one of few branching commands in TEA that does useful work without any label or block references. Some illustrative examples follow…

```
i!:TEST | r:T:P | z:
f:PEST:A
j:B
l:A | x!:-KILL | q!:
l:B
x!:-OK
```

Should return “PEST-KILL” if instruction#2 replaces only the

```
EXPERIMENTS|< 13:20:05 $>* tttt -c "i:{What is your name please? }|i:|x:{Hello }"
What is your name please? Joseph
Hello Joseph
EXPERIMENTS|< 13:20:29 $>*
```

i:{What is your name please? }|i:

Such as we see in the screenshot below:

```
EXPERIMENTS|< 13:27:52 $>* tttt -c "i:{What is your name please? }|i:"
What is your name please? John Doe
John Doe
EXPERIMENTS|< 13:28:05 $>*
```

Finally, note that I: is not only the ONLY way to block a program and prompt for input from the user, but is also the

TEA PRIMITIVE	SEMANTICS	
K:	**NAME**	**Keep**
	PURPOSE	Conditionally filter out the contents of AI
	SYNTAX & SEMANTICS	**k:** INERT **k:REGEX** Only keep lines in AI that match REGEX **k!:** INERT **k!: REGEX** Only keep lines in AI that DON'T match REGEX **k*:** INERT **k*:vNAME:REGEX** **k*!:vNAME:REGEX** The vault operating versions of k: and k!: perform corresponding transformations, but instead operate on the data stored in the vault with the specified vault name, vNAME.
	NOTES	K: is for determining what to keep in the Active Input. It is only active when used together with a useful filter---such as in the following example, where we filter and keep only the lines in a multi-line poem that contain the words "I" or "O"; I!:{Myself should tell You O my Lord. I trust You Know Me.} | k:.*\w?[IO]\w?.* Shall return "You O my Lord. I trust You Know Me." Note that while k: processes AI by splitting it up into lines first, yet, it preserves the original lines by gluing the results back using a single New Line Character (sometimes with a Carriage Return on some systems such as Windows). In terms of TEA utilities for throwing things out of strings, the Keep command mostly helps with filter processing at the line level. For filtering at character or word level, perhaps d: and its inverse d!: are better.

TEA PRIMITIVE	SEMANTICS
L:	

Label

PURPOSE	Explicitly mark sections in a TEA Program accessible by unique names
SYNTAX & SEMANTICS	**L:** INERT **L:LABEL** Declare a jump position in a TEA program accessible by the name LABEL **L!:** INERT **L!:LABEL** **L!:LABEL1:LABEL2:LABEL3:..:LABELN** Similar to L:, but allows for the same position in the program to be accessible using any of the specified label names or tags, which are expected to be unique not only across the list, but also across the entire TEA Program.
NOTES	Note that for the overloadable TEA Label construct **L!:,** any branching command that references any of the label values in the label set expression for a particular **l!:** in the TEA program, makes the program jump there irrespective of which of the names was used. This allows TEA programs to implement useful ideas such as functions, polymorphism, more humane-APIs, etc. Also, note that, to emphasize this important TEA feature, only the Inverse Label construct allows overloading, much as both the **l:LABEL** and **l!:LABEL** constructs work the same for singular labels. The following is a non-trivial example of how labelling solves problems: in TEA programming:

```
# TEA data processing prog with some kind of error handling
v:vLOG:{--No Processing Yet--}
l:FETCH
w!: https://pastebin.org/KYC.csv # rqrd 2b sme non-empty result
f!:^$:PROCESS:ERROR | # process iff data is not empty
l:PROCESS
v:vDATA
z!: date -Ins | v:vDATE
# now combine current date & logs with the data
g*:{---}:vDATE:vLOG | q!: | # end by returning data report
l:ERROR
| v:vERROR:{--Data Access Error--}
g*:{***}:vLOG:vERROR | v:vLOG | # update the log
j:FETCH | # then re-try the data processing
```

TEA PRIMITIVE	SEMANTICS

TEA PRIMITIVE	SEMANTICS	
N:	**NAME**	**Number**
	PURPOSE	Generate a Random Number using some criteria
	SYNTAX & SEMANTICS	**n:** Return a random whole number n in the range (0, 9) inclusive **n:N1:N2:N3:GLUE** **n:LIMIT:LLIMIT:SIZE:GLUE** Same as n:, but using range (0,LIMIT) or rather n: invoked with arguments behaves as such: The first argument, N1 sets the upper limit, so that the number generated is from the range 0 <= n < N1 With the second argument too, we also control the lower limit so that the number generated lies in the range N2 <= n < N1 The third argument tells the n: command to return utmost N3 numbers in the specified range. Finally, the last argument, GLUE, specifies how to glue the generated numbers. By default, the glue used is the SINGLE SPACE CHARACTER. **n!:** Same as n: **n!:LIMIT** Same as n:LIMIT **n*:vNAME** **n*:vLIMIT:vLLIMIT:vSIZE:vGLUE** Same as n!:LIMIT or the longer form, n:LIMIT:LLIMIT:SIZE:GLUE but referencing values stored in vaults
	NOTES	Much as TEA is generally considered to be a Text Processing language by definition, and yet, with the N: TEA command space, we find primitive utilities in TEA, that make numerical transforms and computations somewhat possible---given TEA is essentially a string, and not number processing language. Merely by having an inbuilt mechanism to generate random numbers within TEA programs, many interesting and useful mathematical, or rather, numerical processing problems become readily solvable. The most basic Random Number Generator (RNG) possible, is simply implemented using the following minimalist TEA program: N: # (=6) even just n!: would similarly work The parameterized version of the N-command space instructions can be illustrated with the following basic, but very potent example – perhaps an example for how to generate random but correct IP addresses

TEA PRIMITIVE	SEMANTICS	
O:	**NAME**	**Order**
	PURPOSE	Order, or rather, Sort things
	SYNTAX & SEMANTICS	**o:** Return the AI sorted alphabetically by words it contains. **o:VALUE** Same as o:, but using VALUE instead of AI **o!:** Return the AI sorted alphabetically by characters it contains. **o!:VALUE** Same as o!:, but using VALUE instead of AI **o*:vNAME** **o*!:vNAME** Same as o:VALUE and o!:VALUE, but using the string stored in the vault with the name vNAME as the VALUE
	NOTES	Being able to order things at will is such a formidable power, it can't be underestimated that it is built into the TEA language as a primitive. Ordering makes design possible, prevents or limits chaos and randomness, and allows structure to be created or imposed on things. For TEA, the O: command space offers several utilities for performing ordering operations on strings directly in the program, user-provided or those stored in memory. A basic illustration of this power can be demonstrated by the following example TEA program that returns the alphabetically sorted initials of a person's name: # following URL is hypothetical, but expected to # return the full, undecorated name of someone W!: https://mit.edu/vc/name.txt # (="Terrance L. Epstein Von Zalta") D!:^.: [a-zA-Z] # (="T L E V Z") G: # (="TLEVZ") O!: # (="ELTVZ") Another example: i!:{mice ice best acts zap}|o: #should sort input by lexical order of words and then return… #=acts best ice mice zap i!:{mice ice best acts zap}|o!: #should sort input by lexical order of letters, and thus return: #= aabccceeeiimpsstt z

TEA PRIMITIVE	SEMANTICS	
P:	**NAME**	**Permutate**
	PURPOSE	Generate unique permutations of things
	SYNTAX & SEMANTICS	**p:** Return the AI expanded to utmost 100 unique permutations of its elements (which essentially means returning anagrams of the AI), glued together **p:VALUE** **p:VALUE:GLUE** **p:VALUE:GLUE:LIMIT** **p:VALUE:GLUE:LIMIT:LLIMIT** Same as p: but using VALUE instead of AI. Also, if the second argument is also provided, it becomes the GLUE string used for joining the permutations generated when constructing the IO. The default or if GLUE isn't specified, is the SINGLE SPACE CHARACTER. The last argument is expected to be a number LIMIT, to limit how many permutations to return at most. If LLIMIT is provided, then exactly restrict the output to between LLIMIT as lower limit and LIMIT as upper. If LLIMIT = LIMIT, return exactly LIMIT permutations if they can be generated. **p!:** Return a random string of utmost 100 characters. **p!:SIZE** **p!:SIZE:ALPHABET** **p!:SIZE:ALPHABET:GLUE** Return a random string of exactly SIZE characters. If the second parameter, the string ALPHABET, is provided, then use only the characters in the provided string for generating the random words, otherwise, will use the full Latin alphabet extended with the SINGLE SPACE CHAR. GLUE serves to join the generated strings if provided, otherwise is the default. **p*: vVALUE** **p*: vVALUE:vGLUE** **p*: vVALUE:vGLUE:vLIMIT:vLLIMIT** Permutates provided string. Same as p:VALUE:GLUE:LIMIT , but using the string parameters stored in the vaults with the specified names, vVALUE as the VALUE, vGLUE as glue, vLIMIT as limit. Only the first, parameter is mandatory. **p*!: vSIZE** **p*!: vSIZE:vALPHABET** **p*!: vSIZE:vALPHABET:vGLUE** Same as p!:SIZE:ALPHABET:GLUE but operating on values stored in vaults.

	NOTES	A basic illustration of this P: power can be demonstrated by the following example TEA program that returns all possible unique combinations of the first 3 English alphabet letters:
		
		`I!:{abc} | v:vA | v:vGLUE:- | p*:vA:vGLUE`
		`# (="abc-acb-bac-bca-cab-cba")`
		
		Perhaps, it is important to stress that since mathematically it is known that a string of N characters has at most N! (N-factorial) possible permutations or rather anagrams, then, without enforcing a hard limit such as the default 100, it can become very expensive to run the p: command especially on large

values. Thus, where necessary, ensure to specify the LIMIT parameter when invoking p: commands.

P: maps a string to a set of its anagrams. This also means, one could do what P: commands do, with clever, or rather creative use of the TEA fundamental anagram command “**a!:**”. This interesting fact is illustrated by the following application of TEA:

```
EXPERIMENTS|< 13:35:35 $>* awk -e "BEGIN{do{n++; system(\"tttt -i abc -c a!:\");}while(n<=100)}" | sort | uniq
```

Returns:

abc
acb
bac
bca
cab
cba

It produces similar output an invocation of p: would have given for the same input “abc”, however, it merely leverages the minimalist TEA program “a!:” and some clever use of the Awk programming language.

NOTE: The **P!:** command space is the only utility in TEA, with which one might elegantly generate random text. This is the equivalent of generating random numbers, for words. A utility many, if not most programming languages never provide a primitive solution for.

The most basic Random String Generator (RSG) possible, is simply implemented using the following minimalist TEA program:

P!:
could return “fnudgzwhh ztnttwhehb iptkerl chwuljtiw”

While, the following basic TEA program is guaranteed to generate useful random strings of exactly 10 characters:

P!:10
could return “ssrmykqzyz”, “dfooctwrid”, “gdo yoqqlt”, etc

Finally, an involved vault accessing example of p: is:

v:vSIZE:5|v:vALPH:"aeiou"|v:vGLUE:-
|p*!:vSIZE:vALPH:vGLUE

Which should return random string of exactly 5 characters, based on the provided alphabet "aeiou" glued with "-", e.g.
#aueoi-oueia-eoaiu-eoaui-eaiou-ueaoi-aouei-uoeai-iouea-ieauo

TEA PRIMITIVE	SEMANTICS	
Q:	**NAME**	**Quit**
	PURPOSE	Quit the TEA program sometimes conditionally
	SYNTAX & SEMANTICS	**q:** Quit the program if AI is currently empty. Returns Empty String **q:REGEX** Return the current AI and Quit the program iff AI matches REGEX **q!:** **Unconditionally quit** the TEA program returning the current AI **q!:REGEX** Return the current AI and Quit the program iff AI DOES NOT match REGEX **q*:** Quit the program if THERE ARE NO VAULTs set (doesn't quit if the DEFAULT VAULT was at least set) **q*:vREGEX** Like q:REGEX, but using pattern stored in vault vREGEX **q*!:** Quit the program if the DEFAULT VAULT is unset or is set to an empty value. **q*!:vREGEX** Like q!:REGEX, but using pattern stored in vREGEX
	NOTES	Like f:, q: helps to introduce conditional processing into TEA programs, and with the **q:** and **q!:** command, it becomes easy to ensure our TEA Programs shall always halt and that it's possible to control when they should halt. The following example program will avoid trying to run the expensive **p:** command on gibberish or unwanted externally obtained text: w!: https://pastebin.org/NIN.txt # expects alpha-numeric string with no spaces in it Q:^.*\s+.*| v:vA | p*:vA:- The following example demonstrates how Q: can be useful. For example, this program will return a random number if the input contains a hyphen "-" in it, otherwise will return the input with its content shuffled: #i:"hi there" i:"hi-there" a!:|Q!:-|n: IMPORTANT! Note that any TEA program whose first instruction is just "**q!:**" shall never do anything useful other than immediately quit/return. For the cases where we use the other variants of q:, especially those that can cause the program to quit

TEA PRIMITIVE	SEMANTICS	
R:	**NAME**	**Replace**
	PURPOSE	Replace things with other things
	SYNTAX & SEMANTICS	**r:** Replace all visible characters in AI with the EMPTY STRING and any whitespace except the NEW LINE character with the FULL-STOP character. **r:REGEX:SUBSTR** Return AI with the first section matching REGEX replaced by SUBSTR **r!:** The Inverse of r:; replaces all visible characters in AI with the SINGLE WHITESPACE, and all whitespace other than the NEW LINE character in AI with the FULL-STOP character. **r!:REGEX:SUBSTR** Return AI with ALL sections matching REGEX replaced by SUBSTR **r*:** INERT **r*: vNAME:REGEX:SUBSTR** Same as r:REGEX:SUBSTR,but operating on the string stored in the vault with the name vNAME instead of AI. With only vNAME, like r:, but operating on strings in the named vaults instead of AI **r*!: vNAME:REGEX:SUBSTR** Same as r!:REGEX:SUBSTR,but operating on the string stored in the vault with the name vNAME instead of AI.
	NOTES	String substitution as a core operation in most text processing, finds its main mechanics implemented using the R-command space in TEA. However, it should be noted that r: primitives aren't the only kind that can perform string substitution in TEA. Some kinds of text replacement operations are possible using other TEA primitives such as g: that replaces whitespace with empty space, but also does some automatic replacements on punctuation with the g! variant. Much can be accomplished with mere text substitution operations. The example below is one solution to compressing messages meant for SMS so as to keep the messages short, still meaningful, and thus save on SMS/data charges. r:[Ww]h:w | r:[iI][nN][gG]:in | r:and:n r:to:2 | r:for:4 | r:how:hw | r:ed:d r:[ ]*are[ ]: r | r:why:y | r:ou:u | r:[ ]be[ ]: b Concerning controlling what to replace in a string, note that r:REGEX:STR replaces only the FIRST occurrence of REGEX in what is being processed with STR, but r!: replaces ALL occurrences of the pattern in what is being processed. We see this by the following two examples: i!:I like this|r:[aeiou]:_: returns "I l_:ke this", while

i!:I like this|r!:[aeiou]:_:

returns "I l_:k_: th_:s". Basically, the R!: form performs multiple substitutions, while R: only replaces the first occurrence of the target pattern.

Note that the **r:** and **r!:** primitives offer distinct, but related cryptographic utilities for using whitespace as information in machine-readable cryptograms. Essentially, they offer a strange but useful means to read "words between lines" so to say; systematically converting usual writing using visible text to a kind of invisible but readable writing system – writing using whitespace, an important primitive capability for some families of cryptography.

Also, note that r: and r!: work very differently, though somewhat similarly. The first eliminates visible characters and recognizes spaces and new lines, while r!: recognizes visible characters and new lines, but simplifies reading white-space. Assuming we have the following simple text as the input:

Myself should tell
You O my Lord.
I trust You Know Me.

We can then appreciate the "Braillish" projections of it with r: and r!: as seen below:

i!:{Myself should tell
You O my Lord.
I trust You Know Me.} | r:

Should return

```
..

...

....
```

While

i!:{Myself should tell
You O my Lord.
I trust You Know Me.} | r!:

Returns what we see in the test screenshot below

```
CLITTTT|< 13:42:43 $>* cat tests/test_r2.tea | tttt
      .      .


   . .  .


 .    .   .    .
```

Clearly, we see that these two primitives serve a role very hard to replace with anything else!

TEA PRIMITIVE	SEMANTICS
S:	

NAME	**Salt**
PURPOSE	Randomly Salting or Un-salting strings
SYNTAX & SEMANTICS	**s:** Inject a SINGLE SPACE CHARACTER into AI at a random position. Salting the AI. **s:STR:N:N2** **s:STR:LIMIT:LLIMIT** Return AI with the string STR injected into it at some random Position. With N specified, injects at an index position (from 0) not greater than N within STR, injecting anywhere if N > |AI|. To precisely control where to operate, use N2, as a number to control the lower limit for the index search, so that, when N = N2, it basically means to precisely operate at index N, otherwise one can control the operating space more liberally. **s!:** Delete a SINGLE CHARACTER from AI at a random position. Unsalting AI. **s!:REGEX:N:N2** Randomly select one of the sections in AI matching REGEX, and delete it. With N specified, only deletes an occurrence not later than the Nth-occurrence of REGEX within AI, with N2, operates on the range between N2 and N, operating on exactly the Nth occurrence if N=N2 **s*: vNAME:STR:N:N2** Same as s:STR:N:N2, but operating on the string stored in the vault with the name vNAME instead of AI. **s*!: vNAME:REGEX:N:N2** Same as s!:REGEX:N:N2, but operating on the string stored in the vault with the name vNAME instead of AI.
NOTES	First, it should be noted that salting strings is not the same as substitution. In salting, the original string contents remain, but new content is injected at random (default) or defined positions in the string. Unsalting on the other hand, leaves the original string with some sections of it randomly deleted. Some examples follow: i!:TEST|s:__ # shall sometimes return "TE__ST", "__TEST", "T__EST", etc. i!:TEST|s!: # might sometimes return "TET", "EST", "TES", etc. Basically, unsalting The following example, called the "SIR Game"---perhaps, because of the TEA primitives it employs, uses salting to create a simple child's game where they are tasked with filling in the gap for a letter missing in a given 5-letter word, which is then marked by a "?". # randomly pick a word from the given list I!:{HONEY TRICK WINDS GAMES} | a: | d:[ ].*$ # (="TRICK") for example

S:-:4 | # randomly inject "-" b4 4th pstn
#(="TR-ICK") for example

TEA PRIMITIVE	SEMANTICS	
T:	**NAME**	**Transform**
	PURPOSE	Transform strings using certain methods
	SYNTAX & SEMANTICS	**t:** Apply the rightmost triangulation transform to AI **t:STR** Same as t:, but operating on the given string STR **t!:** Apply the leftmost triangulation transform to AI **t!:STR** Same as t!:, but operating on the given string STR **t*: vNAME** Same as t:, but operating on the string stored in the vault with the name vNAME **t*!: vNAME** Same as t!:, but operating on the string stored in the vault with the name vNAME
	NOTES	Some string transformations are so important in TEA, it was decided to make them primitives in the language so as to utilize them as first-level citizens of the language. Currently, the T-command space mostly implements the so-called "Cetamol Triangulation", "Cetamol Vertical Inversion Transformer" [1] transform on AI or strings held in vaults. Perhaps other transforms might also get this privilege in future TEA, or not. Time will tell, however, the inventor of the language chose to implement T: this way, and he has respectable reasons why it must be that way. A basic demonstration of the T-transform is demonstrated below using a command line invocation of TTTT: # implements the PARACETAMOL T-transform tttt -i PARACETAMOL -c t: PARACETAMOL ARACETAMOL RACETAMOL ACETAMOL CETAMOL ETAMOL TAMOL AMOL MOL OL L Talking of different ways to transform inputs using TEA APIs, note that currently, TEA is on the WEB! Yes, so, the above example can also be run on any device, any operating system, without installing anything, simply by visiting the following URL in a modern web browser: **https://tea.nuchwezi.com**?i=PARACETAMOL&c=t:&run

TEA PRIMITIVE	SEMANTICS
U:	

NAME — **Uniqueify**

PURPOSE — Compute the unique projection of things (effectively, their **modal sequence** statistic)

SYNTAX & SEMANTICS

u:
Return AI reduced to only its UNIQUE WORDS, their order representing their relative frequency in AI, most frequent first, otherwise preserving order of first occurrence.

u:STR
Same as u:, but operating on the given string STR

u!:
Return AI reduced to only its UNIQUE CHARACTERS, their order representing their relative frequency in AI, most frequent first, otherwise preserving order of first occurrence.

u!:STR
Same as u!:, but operating on the given string STR

u*: vNAME
Same as u:, but operating on the string stored in the vault with the name vNAME

u*!: vNAME
Same as u!:, but operating on the string stored in the vault with the name vNAME

NOTES

Uniquefying things is critical and primitive in TEA. Outputs of the U-command space can help in making critical decisions about things just by looking at their ordered base elements. For example, it is a common fact that in the English language, the letter “e” occurs the most in most across words. Now, such interesting analysis can be done primitively in TEA using the U-commands---especially because they not only tell us what the unique elements in a string are, but also which ones are most common, which ones rarest. Very useful foundational statistical analysis for free, so many classes of string processing problems become simpler to reason about in TEA.

A basic example of how to use U: is the problem of determining the winner at an election given the votes data. In this example TEA program, we assume a vote for candidate represented by letter A, appears as just the letter “A” in the list of votes, for candidate B, by “B”, and so on. Thus, a list of votes from a polling station, for 3 candidates W, C & A can be represented by the string “AWCCAWAWAAAAACCWACCCCWCACCA”. The following TEA program then, would help automatically rank the candidates by their votes, with the winner appearing first in the result.

```
I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:
```

```
CLITTTT|< 13:48:30 $>* tttt -c "i!:PARACETAMOL| t: | h: | a!: | r!:"
.
.....  ... ...  ..   .      . . ..   .. .. . . . .
....
.         . .  .. .  ... .  . . . .... .. .  . .  ... ..
```

```
EXPERIMENTS|< 13:56:15 $>* tttt -c "I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:"
ACW
EXPERIMENTS|< 13:56:21 $>* tttt -c "I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:" -d
No explicit INPUT found, using STDIN!
INPUT:
 None
```

TEA PRIMITIVE	SEMANTICS	
V:	**NAME**	**Vault**
	PURPOSE	Store and enable operating on stored things
	SYNTAX & SEMANTICS	**v:** Store AI into the default vault (unnamed vault) – stores the EMPTY STRING (TEA Null/None value) if AI is not yet set. Returns AI **v:vNAME** **v:vNAME:vVALUE** With only vNAME specified, store AI in a vault with that name. With vVALUE specified, store that instead of AI. Returns AI **v!:** Return the length of what is stored in the default vault. **v!:STR** Return the length of string STR **v*:vNAME** **v*:vNAME:vVALUE** Store value vVALUE in vault vNAME overriding where necessary. Returns AI **v*!: vNAME** Return the length of what is stored in the vault vNAME. Without vNAME, is like v!:

	NOTES	Every useful programming language needs some simple way to not only store or perhaps temporarily hold many bits of data, but also, be able to access or reference them by name, as well as be able to operate on data at rest. In TEA, the V-command space makes this possible, and it is the closest facility in TEA, to what other languages offer as variables. Because most TEA programs will be operating on AI, the v: and v!: primitives offer a means to easily store or tell the length of AI without explicitly referencing it by name.
		
		The v!: and v*!: primitives offer the only straightforward way to determine the size of things in TEA – as all things being processed in TEA are strings, it comes in very handy for controlling certain operations based on size or counts.
		
		The most basic example for how to return the length of any input is the following simple TEA program:
		
		`i!:ABC|v:|v!:`
		
		It should return just "3" for this example string "ABC".
		
		For more involved applications, see the following program that will help generate an 8-character password from any given non-empty initial string:
		
		`I!:{SOMEVALUE}`
		`G!:|A!: # glue and anagrammatize seed`
		`V:vSRC|V*!:vSRC # store it in vault and return length`
		`Q:0 # quit if seed was empty`
		`V:vPWD:{} #init password with empty string`

L:1TEST
V*!:vPWD # get current length of password

TEA PRIMITIVE	SEMANTICS	
W:	**NAME**	**Webify**
	PURPOSE	Read or Write to the Network
	SYNTAX & SEMANTICS	**w:** Take current AI as a URL and read whatever is at specified network resource address with an HTTP GET request, then set the result as IO iff the command successfully executed, otherwise return EMPTY STRING **w:URL** Read whatever is at specified network resource address URL and set the result as IO iff the command successfully executed, otherwise return EMPTY STRING **w!:** Take current AI as a URL and read whatever is at specified network resource address with an HTTP POST request, then set the result as IO iff the command successfully executed, otherwise return EMPTY STRING **w!:URL** Write or rather, POST whatever is in AI, to the specified network resource address URL as unnamed data and set the result as IO iff the command successfully executed, otherwise return EMPTY STRING **w*:** **w*:vURL** **w*:vURL:v1:v2:v:3..:vN** Same as w!:URL but will execute an HTTP GET request on the URL held in vault vURL with each specified vault (as vNAME=vVALUE pairs) in the query string. With only vURL, posts everything currently in the vaults, using their name and values. With no parameters at all, treats AI as URL, then sends everything in the vaults in the query string. **w*!:** **w*!:vURL** **w*!:vURL:v1:v2:v:3..:vN** Same as W*:, but performs an HTTP POST request and posts the data as form encoded data
	NOTES	TEA might not offer any usual means to read or write files such as many programming languages provide, however, it does offer a clean solution to reading and writing data to a network-accessible resource---which, given it can be purely offline or on-device, allows for a sort of File I/O in TEA via the W: command space. Perhaps to clarify things – when a TEA program reads using W:URL, it is expected that the body of the returned response – such as an HTML document for web requests, or perhaps a data dump in JSON, CSV or such, for an API call, is what is returned by W:. In cases where an error occurs or the resource doesn't exist, W: just returns an empty string. For writing/posting data, W! and W*!: come in handy. The first version posts AI just as a mere untagged value – just like posting a string value to some API end-point for example. The vault-accessing method though, because it has access to both the names and values in the vaults, will compose a sane multi-part request, where each vault and its value (or the empty string if no value) gets posted to the specified URL. HTTP POST requests are the default for W!* because where HTTP GET is desired, the necessary URL with the necessary name-value pairs in a query-string can be composed using TEA, but is not encouraged because of its lack of security, and potential to construct illegal URLs given arbitrary data.

The following example will fetch a person's current account balance from a bank's API given the account number and some special secret.

TEA PRIMITIVE **SEMANTICS**

X:

NAME	**Xenograft**
PURPOSE	Affix things to things
SYNTAX & SEMANTICS	**x:**
	Return AI with AI affixed to itself. Essentially, multiplying AI
	x:PREFIX
	Prefix **PREFIX** to AI. That becomes AI
	x!:
	Return Half of AI. Essentially, reducing AI
	x!:SUFFIX
	Affix SUFFIX to the end of AI
	x*:vPREFIX:vSTR
	Prefix the string in vault vPREFIX to the string in vault vSTR . That becomes AI. Without vSTR , operates on AI
	x*!:vSUFFIX:vSTR
	Suffix the string in vault vSUFFIX to the string in vault vSTR. That becomes AI. Without vSTR, operates on AI

NOTES

Xenografting is the only correct way to affix strings to other strings in TEA. Of course, the Glue command already allows some kind of xenografting especially with its g*: command space. For example, one could both prefix and suffix the string "---" to any other string, to easily turn it into a title in some text environments. The following example TEA program does this using pure X-primitives

```
V:vHEADLINE:{Interoperability Is Possible} |
v:vAFFIX:{---} | x*:vAFFIX:vHEADLINE | v:vHEADLINE
| x*!:vAFFIX:vHEADLINE
```

While the following does the same, using pure G-primitives:

```
V:vHEADLINE:{Interoperability Is Possible} |
v:vAFFIX:{---} | g*:{}:vAFFIX:vHEADLINE:vAFFIX
```

Both programs should return the result

"---Interoperability is Possible---"

Shall produce the following output:


```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "vAC": "123456XXX",
    "vKEY": "001001",
    "vTEST": "Hello World",
    "vURL": "http://httpbin.org/post"
  },
  "headers": {
```

TEA PRIMITIVE **Y:**

SEMANTICS

NAME	Yank
PURPOSE	Return things from memory
SYNTAX & SEMANTICS	**y:** Return whatever is currently stored in the default vault. (the unnamed vault) **y:vNAME** Return whatever is stored in vault with name vNAME **y!:** Return the length of whatever is currently stored in the default vault. (the unnamed vault) **y!:vNAME** Return whatever is currently stored in vault vNAME **y*:** Return whatever was the external, user provided initial input to the TEA program – the initial AI **y*:vNAME** Return whatever is currently stored in vault vNAME **y*!:vNAME** Return the length of whatever is currently stored in vault vNAME. **y*!:** Return the length of whatever was the external, user provided initial input to the TEA program – the initial AI
NOTES	Yanking is essentially the opposite operation of vaulting in TEA. The Y-command also allows TEA programs to correctly reference original user-provided input in a TEA program – to help avoid the idiomatic inclusion of default inputs as part of common practice in TEA programming---such as with the use of i!: and w!: commands, from making access to original user-provided input impossible when needed. The Y-Inverse commands, like V!, also allow for a straightforward way to determine the size of strings in TEA – for Y, only possible with strings already held inside some vault, while v!:STR also can help determine the length of explicit, inline string values.

invoked, uses it as the URL and performs an HTTP GET to that endpoint, returning the result as IO

On the commandline/with CLI TEA/tttt, we see this work out as such:

```
tttt -c "i!:https://freeipapi.com/api/json/|w:"
{"ipVersion":4,"ipAddress":"41.210.143.35","latitude":-0.07722000000000001,"longitude":31.4567,"countryName":"Uganda","countryCode":"UG","capital":"Kampala","phoneCodes":[256],"timeZones":["Africa\/Kampala"],"zipCode":"-","cityName":"Sembabule","regionName":"Central Region","continent":"Africa","continentCode":"AF","currencies":["UGX"],"languages":["en","sw"],"asn":"20294","asnOrganization":"MTN Uganda","isProxy":false}
```

TEA PRIMITIVE Z:

SEMANTICS

NAME	Zap
PURPOSE	Do things in TEA using external power
SYNTAX & SEMANTICS	**z:** Return AI transformed to ALL LOWERCASE **z:CMD** Invoke the system command CMD passing AI as only input, set the result as IO iff the command successfully executed, otherwise return EMPTY STRING **z!:** Return AI transformed to ALL UPPERCASE **z!:CMD** Same as z:CMD, but will return an Error Message from the System command execution if it didn't execute or if there were errors. **z*:** Return AI transformed to TITLE CASE **z*:vCMD** **z*!:vCMD** Same as z:CMD and z!:CMD, but using the command string stored in the vault with name vCMD

NOTES

The Z-command space is for bringing external (think Unix, System or non-TEA) powers into a TEA program… But also, much as its outputs might be harder to predict or determine up-front, and yet, among the TEA primitives, it is one of the simplest to use and recall, because of the simplicity of its API.

Also, because the Z: primitive allows a TEA program to set the value of the ACTIVE input to whatever that command returns from executing an external system command, it then becomes a potential way to read external values into the TEA program, and also might allow writing the ACTIVE input to some external memory store. Other input commands such as **i:** and **w:** can be likened to **z:** in this regard, with the exception that any potential data or values that might be introduced into a TEA program using the **i:** primitive is explicitly stated in the TEA program code.

The earlier sections of this specification already contain several example TEA programs using the Z-primitive. But, perhaps, it should be noted, that, just like the network accessing commands in the W-command space might not do what they are expected to do when the program is being run without access to a network – such as when a program meant to post data to a server runs while the host device has no data connection, also, in some operating environments – such as when a TEA program is being run in an environment that doesn't expose the system – such as on un-rooted mobile devices – they might be running on Linux underneath, but this space might not accessible to TEA programs by default. While, in other environments like Windows or LINUX, such access is guaranteed to be possible most times. Thus, Z-primitives should be used cautiously, and perhaps only minimally or not at all if possible, so as to ensure that one's TEA programs shall do what they are expected to do portably or across all environments. Z-command space should mostly be kept for advanced, non-trivial cases that are expected to only be required in special environments one is sure about.

Z: basically makes it possible to perform some basic system programming in TEA. For a simple example demonstrating the power of the Zap facility, here is a program that would generate random numbers from words:

```
i:abcdefghijklmnopqrstuvwxy z
a!:
d:[ ].*$
r!:[aeiou]:0
r!:[bcdfghjklmnpqrstvwxyz]:1
x:{ibase=2; }
x:{echo '}
x!:{' | bc}
v:vCMD
c:
z*:vCMD
```

Where we assume the system or shell command `bc` exists. Also, this example shows how we can use TEA to construct custom or special system commands, and then use them to solve problems based on user-provided data. For example this program constructs a call such as

```
echo 'ibase=2; 1100011011111101111' | bc
```

Which, when passed to the Z-command returns “407535” on a system where the “bc” command is accessible. A slightly modified version of this program is shown below, which would take any user input and return it as a number. It can for example show that the name “Jesus Christ” is equal to the number 1403!

```
# program converts provided word to a number
i:test
g!: # first eliminate all whitespace
z: # then turn everything to lowercase
r!:[aeiou]:0 | r!:[bcdfghjklmnpqrstvwxyz]:1
x:{ibase=2; } | x:{echo '} | x!:{' | bc}
v:vCMD | c: | z*:vCMD
# Magic!
```

An otherwise simple, and likely potentially widely supported system command is the “pwd” command for returning the current working directory. We can see Z: working with an example invoking this command via TEA:

```
EXPERIMENTS|< 14:59:29 $>*
EXPERIMENTS|< 14:59:55 $>* pwd
/mnt/e/LAB/EXPERIMENTS
EXPERIMENTS|< 14:59:58 $>* tttt -c "z:pwd"
/mnt/e/LAB/EXPERIMENTS
EXPERIMENTS|< 15:00:15 $>*
```

Another useful demonstration is with combining use of TEA vaults with the Z-utility, such as in this example:

```
i:{+'%A %d, %b %Y'}|v:vCMD:{date}|z*:vCMD
```

Which should return something like "Friday 23, Aug 2024"

5 BONUS: CONCERNING HOW TO ACCESS AND RUN TEA PROGRAMS

As of this writing, September 2025, TEA has already come a long way since first inception in 2021 when the language was but a toy experiment with a few instructions (a:, aa:, t:, rt:, s:, etc.) and that it was restricted to just an environment within the origin TTTT android mobile app[5]. Right now, TEA is not only a fully comprehensive language spanning 26 basic primitive instructions a: to z:, but because most of these can also be decorated or qualified (with any of the two standard instruction qualifiers, "!" and "*" or both "*!"), so as to spawn other, distinct instructions, the entire instruction set could be close to approximately 26 x 4 (essentially more than 100) unique, basic TEA instructions! Thus, TEA is no more just a toy or a laboratory thing, but is ready to be applied and exploited in virtually any domains of computer programming, software engineering and computational problem solving in any scientific, mathematical and creative discipline.

As of this moment, with the latest version of command-line TEA at **v1.0.8**[3], while the WEB version of TEA[10][3][11] – meant to closely adhere to, and base its implementation on the CLI reference implementation[3], is also in line with that latest version, and for the foreseeable future, both versions, despite one being implemented in Python, and the other in JavaScript, are meant to, and expected to move in sync, so that we shall have one major version of TEA as a language, and all implementations must reflect the latest, reference standard of the language.

Concerning how to run TEA programs on any supported operating system via the official Linux/Debian packages:

1. If your system supports it, or is adequately prepared (has a Shell, Python3, curl and can run/install *.deb packages), merely run the command: **curl -Ls https://bit.ly/installtea | bash**
2. Alternatively, head over to the TEA Github Project [3], and check the guide on the README, under the how-to-install section, for tips and latest info concerning installation. You might also merely clone the repository and run the TEA executable directly from your system if you know what you are doing.

For those who don't wish to install anything or who just wish to try out TEA without first installing anything, the simplest approaches are two:

1. Head over to the **TEA WEB IDE**[11] and just type/paste in your TEA source code, and hit "RUN". The essential url is https://tea.nuchwezi.com
2. The other option is to use the TEA WEB API. This still uses the above linked-to tool (unless one has installed or deployed WEB TEA on their own server or local environment), and essentially, you craft a special URL which contains both the point to the WEB TEA compiler/interpreter, and then the code and/or input data you wish to run or load, and this API likewise supports directly/automatically running any such provided data/code to see results – including ability to DEBUG such payloads. An example would be the following executable URL: https://tea.nuchwezi.com/?i=PARACETAMOL&c=t:|h:|a!:|r!:&run
    a. In case you wish to explore this further, head over the the WEB TEA IDE, and then click the "More" > "SHOW WEB IDE DOCS" and then read the guidelines displayed in the section on that interface in the "TEA Input".

6 REFERENCES

1. Lutalo, J. W. (2024) ***[notes][v1.1] Thoughts & Ideas Behind Design of TEA language, Nuchwezi Research.*** Accessible via: https://www.academia.edu/122314564/_notes_v1_1_Thoughts_and_Ideas_Behind_Design_of_TEA_language
2. Lutalo, J. W. (2024) ***The TEA language; Design, Implementation and Justification of a new Generic Text Processing Programming Language***, *Academia.edu*. Accessible via: https://www.academia.edu/121280553/The_TEA_language_Design_Implementation_and_Justification_of_a_new_Generic_Text_Processing_Programming_Language
3. Mcnemesis. (2024). ***TEA GitHub Project | The Reference Implementation of TEA (Transforming Executable Alphabet) computer programming language***. Retrieved from https://github.com/mcnemesis/cli_tttt
4. Lutalo, J.W. ***TEA: Transforming executable alphabet mini-lectures***, YouTube. Accessible via: https://youtube.com/playlist?list=PL9nqA7nxEPgulGKWw1L9xEyqgCdEezaKB&feature=shared
5. Nuchwezi (2019) ***TTTextTransformer, The Reference Implementation of the TEA (TExt Alternator) language's Interpreter for Android Operating System***, *Bitbucket*. Accessible via: https://bitbucket.org/nuchwezilabs/tttexttransformer/
6. Free Software Foundation. (2023). GNU Bash (Version 5.1) [Software]. Accessible via https://www.gnu.org/software/bash/
7. Lutalo, J. W. (2020). ***DNAP: Dynamic Nuchwezi Architecture Platform -A New Software Extension and Construction Technology***. TechRxiv. Accessible via https://doi.org/10.36227/techrxiv.13176365.v1
8. Farrar, Janet, and Stewart Farrar. "*Spells and how they work.*" (1990). pp 77-78
9. Joseph Willrich Lutalo. ***Unraveling mysteries of the zha q-agi chatbot: an interview by icc, of fut. prof. jwl and m*a*p ade. psymaz of nuchwezi***. 2025. Accessible via https://doi.org/10.6084/M9.FIGSHARE.29064671
10. Lutalo, J. W. ***TEA RESEARCH: TEA ON THE WEB A High-Level Web Software Operating Environment Specification For The TEA Programming Language: Web TEA Architecture***. https://doi.org/10.6084/M9.FIGSHARE.29591687
11. Joseph, Willrich L. "WEB TEA IDE." ***Web TEA Integrated Development Environment***, Nuchwezi Research, 15 July 2025, tea.nuchwezi.com/. *Accessed 20 Sept. 2025. The only official reference implementation of TEA for any operating system, and which is accessible via any web browser, allowing reading, editing, debugging, sanitization, execution and sharing of TEA programs using direct code or mere URLs.* https://tea.nuchwezi.com
12. Lutalo, J. W. (2025). ***The Theory of Sequence Transformers & their Statistics***. Nuchwezi Research, 1(1), 37. https://doi.org/10.6084/M9.FIGSHARE.29505824.V3

A: Anagrammatize

B: Basify

C: Clear

D: Delete

E: Evaluate

F: Fork

G: Glue

H: Hew

I: Interact

J: Jump

K: Keep

L: Label

Since 2024

REF:
bit.ly/tazfile

ARCHITECT:
jwl@nuchwezi.com

TEA: Transforming Executable