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**](mailto: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.

# 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.

## 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

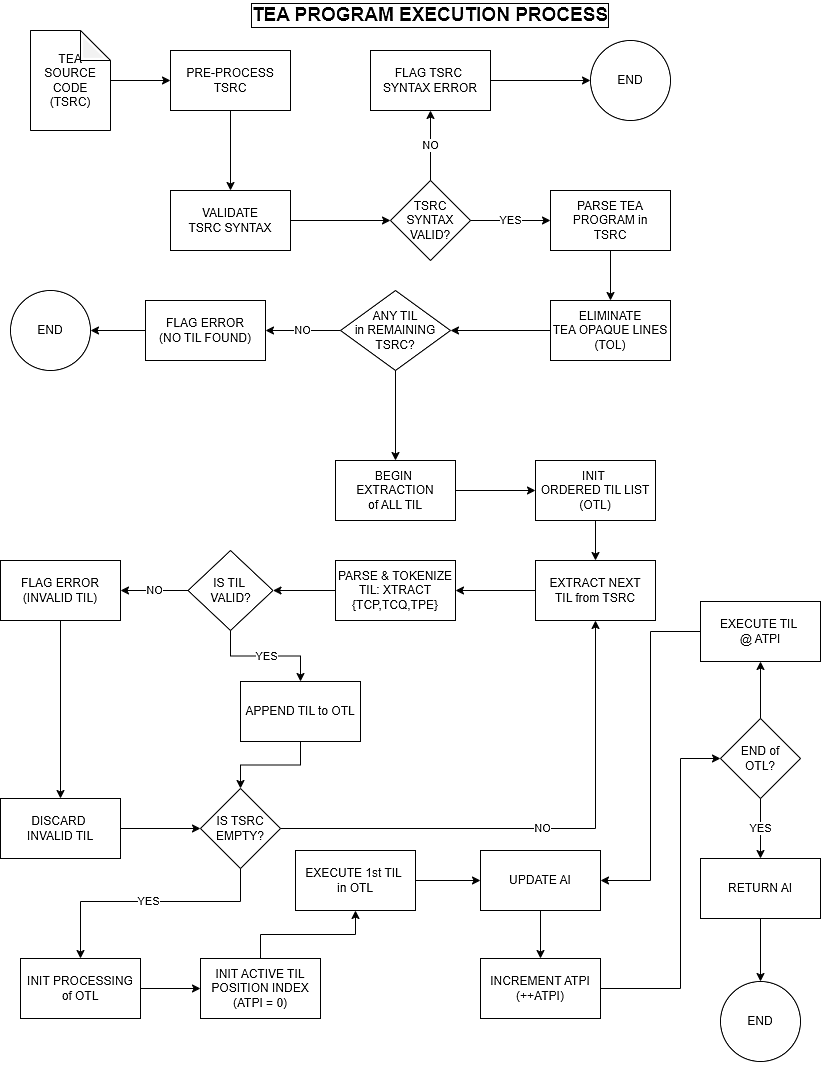
l:C | q!:

#(=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.
   1. 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
   1. 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.
   2. 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:



# 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.

# 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.

# 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.