TEA: Transforming Executable Alphabet — A Minimal Language Leveraging Sequence Transformer Chaining

JOSEPH W. LUTALO, Nuchwezi Research, Uganda

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. Programming languages happen to be just one aspect of software languages; others include domain specific languages, modeling languages, markup, design, specification, query and scripting languages among several other language categories worth noting. We introduce the Transforming Executable Alphabet (TEA); a compact, general-purpose programming language designed around a *sequence-transformer-chaining* paradigm and optimized for problem solving via text-processing. TEA's instruction set is intentionally minimal: the 26 Latin letters form the core commands, optionally qualified by a small set of symbols, producing a bounded global command space that enables concise programs, easy memorization, and powerful composition. TEA has been implemented as an interpreted language across platforms (using Python and JavaScript transpilers), supports embedded persistent storage, web client primitives, processing mathematics, and a rich text-processing standard library. We present the language design, formal semantics for instruction chaining, implementation strategies, representative applications (40+ sample programs), and an evaluation of its expressiveness, concision, and cross-platform performance. We conclude with a discussion of TEA's potential roles in education, low-resource mobile programming, domain-specific sequence analysis, and wrap-up with a brief review of other language research projects related to or which have motivated work on TEA.

CCS Concepts: • **Software and its engineering** → **Formal language definitions**; **Semantics**; **Compilers**; • **Theory of computation** → *Formal languages and automata theory*; • **Information systems** → *World Wide Web*; • **Applied computing** → Education.

Additional Key Words and Phrases: Language Engineering, Alphabet-based Languages, Language Semantics, Cross-Platform Development, Applied Transformatics

1 Introduction

Modern software engineering increasingly demands languages that are compact, portable, and expressive for text and sequence manipulation [6]. Designers of new languages are also encouraged to have users of the language in mind and especially to adopt a multidisciplinary approach to the design and evaluation of their new programming or software language [2]. We present TEA (Transforming Executable Alphabet), a language designed from first principles to make sequence transformation concise, composable, and accessible on constrained devices.

Table 1. What an AI Search might tell you about TEA (Transforming Executable Alphabet) right now

> TEA (Transforming Executable Alphabet) is based on the idea of an instruction set whose core is just the 26 letters of the Latin Alphabet, and which we can qualify or decorate using a specific set of symbols known as command qualifiers, `.`, `!`, `*` and `@`, so as to specify 'qualified commands'.

TEA began as a mobile-first text-processing prototype in 2021 and has evolved into a cross-platform language with interpreters in JavaScript (web, mobile-web) and Python (elsewhere), a growing standard library, and a documented instruction set (the TEA TAZ [8]). The language targets scenarios where compactness, offline capability, and text-centric transformations are primary concerns: mobile scripting, embedded text-processing, lightweight data engineering, and domain-specific sequence analysis and transformations among others.

Author's Contact Information: Joseph W. Lutalo, Nuchwezi Research, Uganda, jwl@nuchwezi.com.

In **Section 2** we shall look at the goals behind how TEA was designed, then look at its actual design in **Section 3**. **Section 4** then drills down into the language mechanics underlying TEA including how TEA relates to the mathematics of Transformatics. In **Section 5** we then consider the actual reference implementations of the language and also present the original design of how TEA program processing would work (the basis of current TEA transpilers). **Section 6** presents some highlights of what has been accomplished with TEA thus far, with **Section 8** focusing on a few specific noteworthy case studies, and we then dive into preliminary language evaluation thus far in **Section 7**. We shall also briefly discuss TEA's observed strengths and weaknesses in **Section 9**, and shall contrast the language to earlier and on-going research in software language engineering in **Section 10**. We conclude in **Section 11** by also highlighting what future directions work on TEA is likely to take. The extra material in **Appendix 1** has been added to help illuminate, but also motivate interested researchers towards understanding what Transformatics is all about as well as directions to take for those interested in picking up this new mathematics by the same inventor of TEA.

2 Design Goals and Rationale

The TEA design is driven by the following goals:

- **Minimal, memorable instruction set.** Use the 26 letters as primary command spaces to reduce cognitive load.
- **Composability.** Enable chaining of sequence transformers [19] to express complex pipelines succinctly.
- **Conciseness.** Allow [even complex] programs to be expressed as single strings when needed (minified form) — as *one-liners*.
- **Cross-platform portability.** Provide interpreters for web, mobile, desktop, and server environments.
- **Practical tooling.** Ship a standard library, sample programs, and a reference manual (TEA TAZ).

These goals reflect TEA's origin as a mobile-first language for low-resource environments and its subsequent maturation into a general-purpose text-processing language.

3 Language Overview

3.1 Lexical and Syntactic Conventions

A TEA instruction consists of a *key letter* (A–Z or a–z), optionally qualified by one of the qualifier symbols `. ! * @`, followed by a colon `:` and zero or more parameters separated by colons. Instructions may be placed on separate lines (sanitized form), combined on a single line separated by vertical bars `|` (minified form), or mixed with comments beginning with #.

```
i!:Hello World          # print `hello world'
n:1000:10               # RNG from [10,1000]
i!:{Name?}|i:|x:{Hi }   # name prompt + greet
```

Listing 1. Minimal TEA Program Examples

3.2 Command Space and Qualifiers

Under the language's grammar rules [8] we can form TEA commands using; one of 26 case-insensitive letters, qualifiers drawn from the symbol set $\{*, !, ., @\}$, each qualifier optional and used at most once, and when multiple qualifiers

appear they must follow the fixed precedence order `* ! . @`—each qualifier is either present or absent, giving $2^4 = 16$ qualifier forms per letter (including the empty form). Multiplying by the 26 letters yields a total of

$$26 \times 2^4 = 416$$

distinct TEA commands. This count includes the unqualified letter form and all combinations of one, two, three, or all four qualifiers written in the canonical order.

Table 2. Command-space combinatorics for TEA (fixed qualifier precedence).

Attribute	**Value**
Letters (case)	26 (case-insensitive)
Qualifier set	{∗, !, ., @}
Qualifier rule	Optional; no repetition; fixed precedence `* ! . @`
Qualifier forms per letter	$2^4 = 16$
Total distinct commands	**416**

Also, note that each letter denotes a *command space* (e.g., `I:` for I/O, `N:` for numeric utilities, `R:` for rewrite and math etc.) — see details in **Table 3**. Qualifiers modify semantics (for example, `r.:EXPR` is for evaluating math expressions, while `r:PAT:SUB` is for text rewriting [8]). However, despite the global maximum number of distinct qualified commands being bounded, and yet, in practice, not all of these are currently being utilized. Conventionally, unutilized or undefined TEA commands are labeled or considered **INERT** [8], generally have no effect in a TEA program, and might be flagged at runtime as part of dynamic program validation or debugging [12].

4 Formal Semantics

This section gives a concise, mathematically grounded semantics for TEA suitable for a SLE paper. We combine the language definition excerpted from the TEA TAZ [8] with a compact operational model based on *sequence-transformer chaining* [19].

4.1 Overview and notation

We model TEA programs as ordered sequences of *TEA instructions* (TIs). Let Σ denote the set of TEA strings (UTF-8 text). A TEA instruction t is a syntactic object formed from a *key letter* $\ell \in \mathcal{L}$ (where $\mathcal{L} = \{a, \ldots, z\}$ is case-insensitive) optionally followed by a qualifier pattern q drawn from the canonical qualifier symbol set $\mathcal{Q} = \langle *, !, ., @ \rangle$ and zero or more colon-separated parameters. A TEA program of length n is an ordered tuple

$$\mathbb{T}^n = (t_1, t_2, \ldots, t_n).$$

We write I for an arbitrary sequence value (the *TEA input/state*) and O for the program output/state.

4.2 Sequence-transformer chaining (informal)

Each instruction t_i denotes a *transformer* T_{t_i}, a (possibly partial) function from sequences to sequences:

$$T_{t_i} : \Sigma^* \rightharpoonup \Sigma^*.$$

Execution of a program $\mathbb{T}^n$ on initial input I_0 composes these transformers left-to-right to produce a final output O:

$$I_0 \xrightarrow{t_1} I_1 \xrightarrow{t_2} I_2 \cdots \xrightarrow{t_n} I_n = O,$$

equivalently $O = (T_{t_n} \circ \cdots \circ T_{t_2} \circ T_{t_1})(I_0)$. This presentation follows the sequence-transformer chaining paradigm and aligns with the Transformatics perspective on ordered sequence processing [19].

4.3 Small-step operational semantics

We present a small-step operational semantics that captures deterministic instruction evaluation and state threading. A *configuration* is a pair $(\mathbb{T}^n, I)$ where $\mathbb{T}^n$ is the remaining instruction sequence and I the current state. The one-step relation $\longrightarrow$ is defined by the rule below:

$$\frac{T_{t_1}(I) = I'}{(t_1, t_2, \ldots, t_n; I) \longrightarrow (t_2, \ldots, t_n; I')} \qquad \text{(STEP)}$$

If T_{t_1} is undefined on I (partial transformer failure or an *inert* TI), the configuration is stuck; practical interpreters may perhaps raise an error or perform a recovery action (logging, fallback transformer, etc.). In current implementations though, we generally merely skip or ignore specifically inert TI, passing on I_{i-1} to $T_{t_{i+1}}$ when T_{t_i} failed or was inert. Some commands though, also merely set $I' = \text{“”} = I_{i+1}$ in such a case; set current transformer output as the `EMPTY SPACE` upon encountering a runtime error [8]. Multi-step execution is the reflexive transitive closure $\longrightarrow^*$. Termination with output O is:

$$(t_1, \ldots, t_n; I_0) \longrightarrow^* (\,; O) \quad \text{iff} \quad O = (T_{t_n} \circ \cdots \circ T_{t_1})(I_0).$$

4.4 Big-step semantics (evaluation judgment)

For many proofs and reasoning tasks a big-step (evaluation) judgment is convenient. We write

$$\langle \mathbb{T}^n, I \rangle \Downarrow O$$

to mean that program $\mathbb{T}^n$ evaluates on input I to produce O. The big-step rules are:

$$\frac{}{\langle\, , I\rangle \Downarrow I} \qquad \text{(E-Empty)}$$

and

$$\frac{\langle (t_2, \ldots, t_n),\ I'\rangle \Downarrow O \quad T_{t_1}(I) = I'}{\langle (t_1, t_2, \ldots, t_n),\ I\rangle \Downarrow O} \qquad \text{(E-Step)}$$

These rules are equivalent to the small-step semantics under standard determinism assumptions for the transformers.

4.5 Transformer classification and side effects

Transformers fall into two broad classes:

- **Pure transformers:** T depends only on its input sequence and returns a new sequence; these are total or partial pure functions $\Sigma^* \rightharpoonup \Sigma^*$.
- **Effectful transformers:** T may perform I/O, persistent storage updates, or network actions. We model effectful transformers as functions that thread an explicit *world* or effect state W: $T : (\Sigma^*, W) \rightharpoonup (\Sigma^*, W)$. For the formal core we separate pure evaluation from effect handling; the interpreter semantics composes effectful steps in a deterministic scheduling policy.

4.6 Properties

Determinism. If every transformer T_t is deterministic on its domain and the interpreter enforces a fixed evaluation order, then program evaluation is deterministic: for given $\mathbb{T}^n$ and I_0 there is at most one O such that $\langle \mathbb{T}^n, I_0\rangle \Downarrow O$.

Compositionality. The semantics is compositional: the meaning of a program is the composition of the meanings of its instructions. Formally, if $\mathbb{T}^n = \mathbb{T}^k +\!\!+\, \mathbb{T}^{n-k}$ (concatenation), then for all I,

$$\langle \mathbb{T}^n, I\rangle \Downarrow O \quad \Longleftrightarrow \quad \exists I'.\ \langle \mathbb{T}^k, I\rangle \Downarrow I' \ \wedge\ \langle \mathbb{T}^{n-k}, I'\rangle \Downarrow O.$$

Reasoning about partiality and errors. Because many transformers are partial (e.g., parsing failures, network timeouts), program correctness proofs must account for stuck configurations or effectful recovery rules. We recommend using a totalization strategy (option types or explicit error values) in formal proofs and in critical interpreter components.

4.7 Relation to Transformatics

The TEA semantics directly implements the Transformatics view of ordered sequence processing: TEA instructions are first-class sequence transformers and program execution is their ordered composition. Also, for any sequence of TEA instructions specifying a particular program for some purpose, the entire sequence of instructions is considered a **higher-order sequence transformer**. This alignment provides a principled basis for reasoning about TEA programs using the mathematical tools developed in Transformatics [19].

4.8 Implementation notes

The reference interpreters (Python, JavaScript) implement the above semantics with a lexer/parser that normalizes sanitized, minified, and mixed forms into an internal instruction list that is essentially of sanitized form, an execution engine that applies transformers in left-to-right order, and an effect manager that sequences I/O and persistent storage operations deterministically. The repository and language manual provide the concrete transformer definitions and examples used in the semantics above [7, 8]. **Figure 1** captures and presents the gist of operating based on these semantics.

5 Implementation

5.1 Reference Interpreters

Reference interpreters[1] exist in Python (command-line) and JavaScript (web). The interpreters implement a modular pipeline:

(1) **Lexing/Parsing:** Tokenize instructions, qualifiers, parameters, and comments.
(2) **Normalization:** Convert sanitized/minified/mixed forms into an internal instruction list.
(3) **Execution Engine:** Evaluate transformers sequentially, manage state, I/O, and persistent storage.
(4) **Standard Library:** Implementations for each command space (A–Z) with documented semantics.

The TEA GitHub repository hosts the reference implementations, sample programs, and documentation. However, interested users and students might immediately access a live TEA language software operating environment (SOE) [24] (akin to a *language workbench* or integrated development environment (IDE)) via: https://tea.nuchwezi.com

5.2 TEA Language Processor

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 1**:

5.3 Persistent Storage and Web Primitives

TEA includes a lightweight embedded database ("TEA Database" — essentially a no-SQL key-value storage) for portable scripts, but also primitives for HTTP GET/POST with allowance for setting custom headers and advanced content and request parameter processing. These features enable TEA programs to act as tiny, but robust web clients, API glue, and offline-capable utilities with guaranteed device-specific persistence.

6 Tooling, Ecosystem, and Examples

TEA ships with:

[1]Actually, technically correct to be called "transpilers" for the case of how processing TEA is currently being implemented.

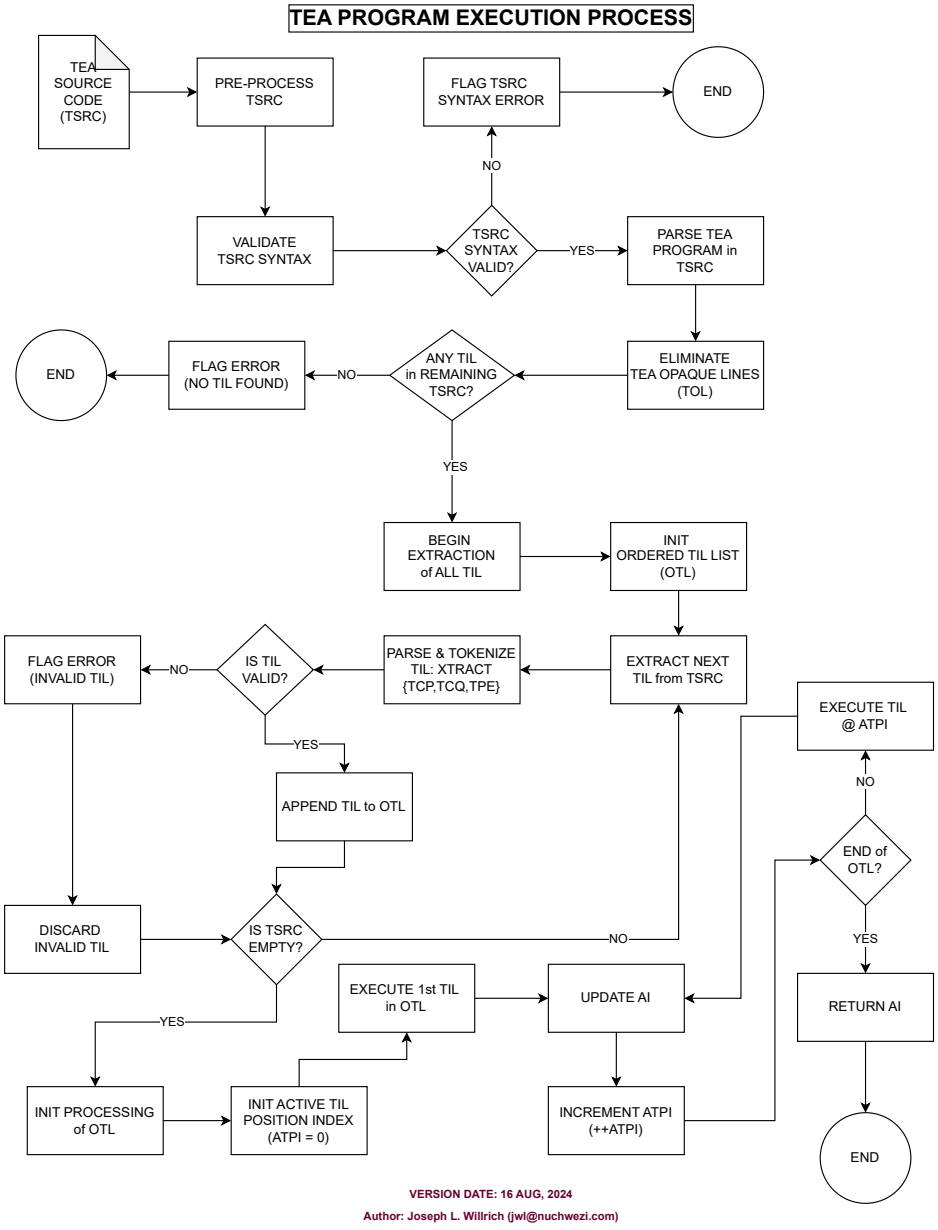

Fig. 1. The TEA Program Execution Process.

- **TEA TAZ:** The language manual and command-space specification [8].

Fig. 2. Latest **TEA MOBILE IDE**

- **40+ sample programs**[2]**:** Examples spanning calculators, data engineering, chatbots, games, cryptography, and a proof-of-concept TEA OS — refer to **Appendix 2**.
- **Command-line package (TTTT):** A Debian package[3] that adds the TEA interpreter as well as some TEA utilities (AMC, ZHA and Lexicon) to any Linux-compartible system for both developers and normal users to leverage TEA.
- **Mobile app (mobile TTTT):** A compact TEA IDE for Android enabling offline development and running or sharing TEA programs and their outputs. See **Figure 2**.

Minimal representative examples include:

- **Hello World:** `i!:Hello World`
- **Generate Random IP Address:** `n!:256:0:4:.`
- **Process Mathematics:** `r.:1/2*(1 + 5**(2**(-1)))`
- **Shuffle List of Words:** `a:{one two three go}`

[2]Readily accessible, usable via **TEA WEB IDE** but also via the *offline-capable* **TEA MOBILE IDE** https://bit.ly/teamobile_apks
[3]Quick, secure installation [on your server] via:
`curl -Ls https://bit.ly/installtea | bash`

And for a more involved minimal example, consider the following TEA program that fetches, preprocesses and presents **random jokes** from some WEB API:

```
i!:https://
x!:utility.nuchwezi.com
x!:/random/jokes
w: | h!:,
k:{setup|punchline}
h!:{:}
k!:{setup|punchline}
d.::. | r!:.\n:_ | d:.$ | h!:_ | d:_
```

Listing 2. EDUTAINMENT: Random Jokes v2.1

It essentially triggers an HTTP GET request, parses and cleans the returned JSON payload, and then produces neat outputs such as *"Why are skeletons so calm? Because nothing gets under their skin."*

7 Evaluation

We evaluate TEA along three axes: **concision**, **expressiveness**, and **performance**.

7.1 Concision

TEA programs are often extremely short in character count. The Hello World example demonstrates minimal syntactic overhead, and if we were to measure program length (characters) for a set of canonical tasks and compare TEA to both classical (COBOL, LISP, C, Sed,...) and modern languages (Python, JavaScript, Go, Ruby,...), one would generally find that, by metrics and cases such as the 8 covered in the DNAP SOE language evaluation suite [24], TEA would arguably outshine most of the competition. Consulting **Listing 3** should also help drive this home.

7.2 Expressiveness

Using the sequence-transformer chaining model, TEA expresses many text-processing pipelines succinctly. With the extra advantage that its instruction set almost exhaustively caters for most text-processing oriented problem solving via a suite of orthogonal transformers for arbitrary general problems, that one can then readily take any or most kinds of computational problems and model them using sequence transformers and their chaining, and thus readily harness TEA for realizing their specified solution. In this regard, TEA is arguably a very dependable and lightweight prototyping platform for personal, business, academic and scientific computation problems. In the TAZ, we present numerous case studies to demonstrate this power: a text-based adventure game ("Boobs 'N' Traps") [13], a q-AGI chatbot prototype (ZHA) [22], and a DNA/RNA sequence analysis pipeline (from the Applying Transformatics in Genetics work) [9], the LUMTAUTO bi-directional encryption [15] and more as one shall find in [8]. These case studies show TEA's ability to encode domain logic compactly.

7.2.1 TEA Instruction Set Expressiveness. Briefly, we shall look at why the minimal instruction design used in this language was chosen; ease of recall via a mnemonic naming of the inbuilt "instruction library", the desire to encourage

meaningful program composability via use of orthogonal instructions and of course, leveraging the sequence transformer chaining paradigm. With **Table 3**, we present the **26 command spaces** that define the TEA language instruction set; A: to Z: and their orthogonal purpose for authoritative reference purposes.

Table 3. TEA instruction set summary (A–Z): primitive, command space name, and its concise purpose within the language.

Primitive	Name	Purpose (distilled from TEA TAZ)
A:	`Anagrammatize`	Produce anagrams or shuffle symbols/lists for randomized reorderings.
B:	`Basify`	Extract a lexical base or canonical symbol sequence for keys and comparisons.
C:	`Clear`	Reset working memory or named vaults; initialize storage to empty.
D:	`Delete`	Remove matched content or filter out unwanted patterns/whitespace.
E:	`Evaluate`	Treat stored text as TEA code; execute or securely evaluate generated code.
F:	`Fork`	Perform conditional branching via pattern tests; enable logic about program state.
G:	`Glue`	Concatenate or bind tokens; remove whitespace/punctuation to form strings.
H:	`Hew`	Split or explode text by pattern; introduce line breaks and lossless splits.
I:	`Interact`	Set or prompt the Active Input (AI); force or conditionally assign AI.
J:	`Jump`	Static program jumps and label-based control flow (unconditional loops).
K:	`Keep`	Line-level filtering: retain only lines or segments matching user patterns.
L:	`Label`	Define named code blocks / labels to enable conditional branching and encapsulation.
M:	`Mirror`	Reverse or reflect text (characters, words, or multi-line inversion).
N:	`Number`	Produce numeric/entropy values; support randomness and stochastic modeling.
O:	`Order`	Sort tokens or sequences (lexical, numeric, or custom ordering).
P:	`Permutate`	Generate permutations or structured/random strings from a given alphabet.
Q:	`Quit`	Terminate execution (conditionally or unconditionally); support cut-points.
R:	`Replace/Rewrite`	Perform pattern-based substitution and general rewrite transformations.
S:	`Salt`	Randomly inject or delete characters/spaces or perturb text for variation.
T:	`Transform`	Apply formatting/structural transforms (trim, align, other string transforms).
U:	`Uniquefy`	Extract unique items and frequency-based projections (modal statistics).
V:	`Vault`	Named storage facility: read/write persistent or ephemeral data (variables).
W:	`Webify`	Network I/O: HTTP GET/POST and interactions with web APIs or remote endpoints.
X:	`Xenograft`	Affix prefixes/suffixes, replicate or reduce text to build parametric strings.
Y:	`Yank`	Read memory/vaults and access stored or initial input values.
Z:	`Zap`	External/effectful operations: non-TEA evaluation, case transforms, introspection.

As one shall find when consulting the definitive and authoritative formal reference concerning the language's instruction set and each command space semantics, usage notes, examples and quirks — essentially the TEA TAZ: [8] — for any particular TEA primitive, such as `V:` (**VAULT**), one finds well documented, an exhaustive breakdown of what each of the **active** (as opposed to *inert*) command signatures are which are currently supported for the primitive. We see an example illustrated succinctly as in **Table 4**. Such should help the [amateur] TEA programmer understand how to leverage or interpret the particular commands supported in that TEA primitive's arsenal so as to fulfill one or more of the purposes laid down for that command space[4].

So, for example, based on the semantics laid down in **Table 4**, we can then read and make sense of the following TEA program...

[4]In TEA, unlike many other languages perhaps, the rule of thumb is: **Read the M*F#@ Manual First!** As well as when in doubt. Otherwise, despite the apparent simplicity of the language's syntax, you'll without doubt readily get lost, throw a tantrum or pull out your hair... utterly confused; gazing at a TEA program written by someone else — such as the infamous **TEA OS**: http://bit.ly/run_teaos — or perhaps trying to debug a thorny case in your own TEA codes, and then go wondering (or even wandering)... with possibly the arcane ancient wyzard daemon bickering-on at the back of your mind... `Proco Proco! Est!` *Abandon all hope, ye thus entering here! Abandon all hope!* And well, of course, the smarter, seasoned engineer or hacker might sometimes decide to abandon hope looking for a solution via reading the [sometimes out-of-sync] manuals or TEA documentation, and instead directly go study the actual TEA transpiler source [7] — it's a commendable and undaunting route given how clean and well maintained that code generally is.

Table 4. Example Reference Semantics for the TEA `V:` (vault) instruction family.

Signature	Semantics (in summary)
`v:`	Store the Active Input (AI) into the default (unnamed) vault; if AI is unset, store the empty string. Returns AI.
`v:vNAME`	Store AI into named vault `vNAME`; returns AI.
`v:vNAME:VALUE`	Store literal `VALUE`(instead of AI) into vault `vNAME`; returns AI.
`v!:`	Return the length of the value in the default vault.
`v!:STR`	Return the length of literal string `STR`.
`v*:vNAME`	Force-write AI into vault `vNAME`, overriding existing content; returns AI.
`v*:vNAME:VALUE`	Force-write literal `VALUE` into vault `vNAME`; returns AI.
`v*!:vNAME`	Return the length of the value stored in vault `vNAME` (no name = same as `v!:`).
`v@:vNAME`	Persist AI under key `vNAME` in the TEA database (persistent vault); returns AI.
`v@:vNAME:VALUE`	Persist literal `VALUE` under key `vNAME` in the TEA database; returns AI.
`v!@:vNAME`	Return the length of the value stored in the TEA database under key `vNAME`.

```
####[ Basic Password Generator ]####
#####################################

#---[ START ]
#given some initial/external seed...
I:{SOMEVALUE} #if none, then use our own seed
G!:|A!: #glue and anagrammatize seed
V:vSRC|V*!:vSRC #store it + return its size
Q:^0$ #quit if seed was empty; returns 0

#---[ PROCESS ]
V:vPWD:{} #init password with empty string
L:lTEST #init named code-block; jump-to point
V*!:vPWD #get current length of password
#lPRODUCE if != 8chars long else goto lRETURN
F!:^8$:lPRODUCE:lRETURN
L:lPRODUCE #another labeled-block starts here
Y:vSRC #fetch the seed
A!:|D!:^. #shuffle seed + pick only 1st char
V:vCHOSEN #store picked letter in a vault
G*:{}:vPWD:vCHOSEN #glue it with current pwd
V:vPWD | V*!:vPWD #store as new pwd, rtn len
J:lTEST #[re]jump to password length test

#---[ FINISH ]
L:lRETURN | Y:vPWD #return final password
```

Listing 3. Basic 8-character Password Generator

7.3 Performance

Though detailed and rigorous benchmarking tests and data wont be presented here (for brevity's sake), preliminary informal benchmarks comparing the two Python and JavaScript-powered TEA interpreters on representative tasks has been conducted. Such tests spanned; string transformation throughput (such as in the *Ghost Messages protocol* encoding and decoding); numeric workloads (such as in the *Fibonacci Sequence generation* tasks) and HTTP client tasks. Results indicate how these interpreters are as competitive and impressively performant on tasks just like implementations in the host languages would be, but with better, simpler task expression and semantics. Of course, the language is still under active development as well as fine-tuning, and thus further optimization but also authoritative quantitative evaluations are left as future work.

8 Case Studies

8.1 Text Adventure: Boobs 'N' Traps

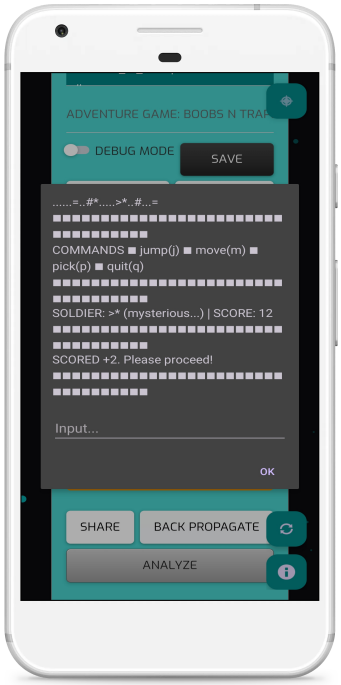

Fig. 3. BOOBS N TRAPS game session screen

Inspired by classic command-line text-adventure RPG game, **NetHack**, this game demonstrates creative multi-strategy and reward logic, a minimal UI that is performant on any device, robust state-machine encoding, command-driven progress and input-command parsing and processing. It is among the best examples of what could be accomplished

in pure TEA programming and the associated game state machine and design notes are documented as found in [13]. **Figure 3** demonstrates what a typical game session feels and looks like on mobile.

8.2 Sequence Analysis and Transformatics

TEA has been applied to sequence analysis workflows in genetics; the language's sequence transformer chaining maps naturally to common bioinformatics pipelines as demonstrated in both [23] and [9]. In particular, we find that TEA could be leveraged for problems such as:

(1) **Genetic Fingerprinting** — determining what the quantifiable similarities and differences between two or more genetic code sequences might be based on computing their modal sequences, sequence characteristics and for clusters or corpora, their population characteristic, etc.
(2) **Genome Expression** — by leveraging sequence transforms such as the mapping of DNA or RNA to decimal sequences and vice versa (such as in the proposed **Lu-Genome Expression System** [9]), one can readily apply algebraic and visual-spatial rendering techniques to the analysis and comparison of genetic sequences.

8.3 Chatbot and q-AGI Prototyping

ZHA is short for "Zee Hacker Assistant", a special chatbot with an interaction mode reminiscent of how hackers on IRC might ideally chat. Originally based off of earlier TEA-powered chatbots such as PA and TEAPA [22], but also inspired by earlier work in human support via chatbots such as with the VOSAC [25] Question-Answer Knowledge-Base driven voice chatbot, ZHA demonstrates TEA's suitability for rapid prototyping of conversational agents using compact transformation pipelines, offline-reliability via meaningful auto-generated content as well as the potential to leverage TEA's embedded persistence and neat text document generation capabilities.

An example output from the latest ZHA (v1.0.8) for a conversation between a fictitious character "Manuela" and the user (lines with no leading identifying handle) is shown in **Figure 4**.

9 Discussion

9.1 Strengths

- **Memorability:** 26-letter core reduces learning friction.
- **Compactness:** Minified programs fit in single strings, enabling embedding in constrained contexts.
- **Portability:** Interpreters for web and mobile enable broad deployment.

9.2 Limitations and Risks

- **Readability:** Extreme concision can reduce readability for complex programs; tooling (formatters, debuggers) is essential.
- **Type and correctness:** The dynamic, transformer-based model used in TEA requires careful testing and formal verification for safety-critical domains especially given that in TEA, there essentially is just one type of data; text or rather strings [8].
- **Ecosystem maturity:** TEA's ecosystem is nascent; adoption depends on documentation, libraries, and community tooling. However, the language has already been under-development and continuous active research since 2021, thus, it no longer is a toy nor mere academic curiosity.

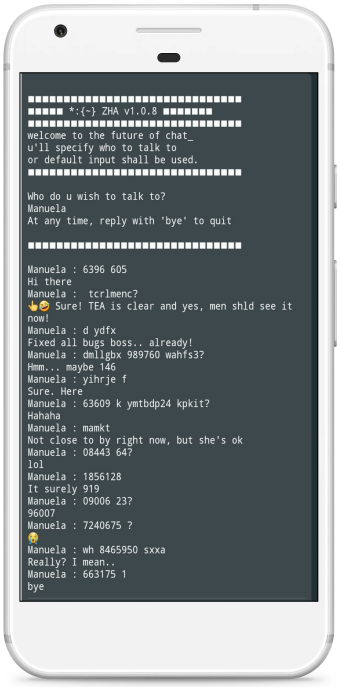

Fig. 4. A ZHA v1.0.8 session dump

10 Related Work

TEA intersects with minimal instruction-set languages, concatenative languages, and domain-specific text-processing languages [6]. It draws inspiration from prior work on language design[2], DSL design, and the theory that many programming tasks reduce to text/sequence processing (see Lutalo's, *Unifying Programming Language Theory* (UPLT) in [6]). Additional related projects include DNAP [4, 24] and other lightweight scripting languages.

Though not directly related to work in Computing nor Mathematics, TEA source code has been featured as part of the narrative in the 2025 creative fiction work from Uganda — the novel *ROCK 'N' DRAW* [17] — as one of the potent technologies being utilized by one military command engineer for cases such as automating remote reconnaissance, drone-auto-piloting and other wartime computational loads.

TEA as a language is founded on the idea of the sequence transformer chaining paradigm, and this idea has been well formalized and developed by the same person inventing TEA, but also the new subfield of mathematics being known as 'Transformatics' [9, 15, 18–21].

Coblenz et al. [2] argue for integrating user-centered design, empirical studies, and diverse evaluation methods into programming-language design; their framework supports combining formal, performance, and human-factors evaluations—an approach that complements but also inspired TEA's design and planned user studies.

Barash (ACM SLE '20) [1] proposes an example-driven approach to language definition and somewhat supports a commendable model for helping newbies master otherwise tricky [meta-] languages and new technologies via the concept of programming-by-example [26]. The paper argues that this lowers the barrier for beginner language engineers and outlines requirements for a web-based tool supporting illustrative syntax definition. However, especially for his presentation of arguments on why many modern languages need great language support tools — so-called language workbenches and IDEs being a core case he covered, it should be noted that research such as his did inspire a great deal of the work that went into designing the reliable, portable and beginer-friendly integrated development environment for the TEA language for both mobile and web users (See **Figure 2**). Also, his perspective complements TEA's emphasis on concise, example-rich documentation.

Concerning TEA's core focus on being a must-go-to language for arbitrary text-processing tasks, we had encountered the ideas presented in [26] relating to the MIT **STEPS** project, and even though theirs attempted to provide an exhaustive solution for text-editing and processing via AI-driven coding and a meta-language, and yet, we decided to instead focus on giving the human programmer a robust language with which they can directly, and manually think-up and resolve most thorny problems related to transforming, fixing and replicating text-processing solutions in relation to working with textual data or files.

Concerning the design and language implementation choice behind keeping the TEA language an interpreted and not a compiled language, note that our familiarity with the problems surrounding attempts to bootstrap, evolve, sustain and make portable/widely-accessible, a custom or customized compiler in the modern programming landscape, as depicted in the research and reviews conducted by Huberdeau et. al (ACM SLE '24) [3] in relation to their **PNUT** transpiler project was quite important. Further, we have chosen to maintain only two reference implementations of TEA (essentially, only two distinct interpreters [7]); one targeting the WEB environment; TEA to JavaScript; which we can likewise leverage for mobile users via methods such as exposing TEA functionality via an embedded web-browser (relates to work we did earlier-on with DNAP for mobile [4]); and the other catering for all non-web scenarios; command-line, server-side, desktop-etc via the TEA-to-Python interpreter. Somewhat similar to the intentions of PNUT, such as wanting to make C-like power available in a widely-accessible environment such as the POSIX shell, TEA was designed with the possibility of leveraging TEA programs from a POSIX shell — such as via the `#!/usr/bin/tttt -fc` facility — in mind, and indeed, and rather interesting, not only can TEA programs be run from a typical POSIX shell environment (Bash and WSL being among cases we have extensively tested already), but that TEA programs can readily be embedded within existing or new POSIX shell-compatible scripts (Bash, AWK, Python, etc.).

Last but not least, experienced legacy language connoisseur and compiler engineer Vadim Zaytsev did present (ACM SLE '20) ideas on his interesting legacy-like, multiple-language inspired academic language known as **BabyCOBOL** in [27], with the intention of demonstrating many of the problematic language design quirks and aspects that make processing, but also using or evolving legacy languages somewhat undeniably tricky if not outright difficult! Interestingly, and worth noting, we did not have Zaytsev's work in mind when designing and implementing the TEA language much as most of the work occurred after the publication of his paper, and yet, later, when reviewing his findings relating to tricky features spanning legacy languages like HLASM, COBOL, REXX, PL/1 and even later ones such as SQL and AppBuilder, we did find that, even though TEA wasn't designed to be difficult to parse or process, and yet, from our experience implementing its syntax but also semantics and comparing to what he documented, that TEA's design and implementation does indeed resolve many of the tricky issues he surfaced via not actually clever, but perhaps, accidentally well-designed features of the TEA language syntax and semantics; a better use of GOTO-like control-flow, multi-level and detachable looping constructs, case-insensitive keywords, etc.

Overall, and especially with regards to how TEA relates to, or is inspired by contemporary and earlier work in the software language engineering field (especially because, since the writing of the first language engineering review paper treating of TEA [6] in 2024), that much of the remaining work that went into the design and implementation of the language was actually based-upon or inspired by ideas we established while reviewing ACM SLE research since the inception of the conference in 2008! However, and for purposes of keeping this manuscript brief, we shall relegate the task of a complete and paper-by-paper comparison or review, vis-a-vis what is or has not been done in the TEA language project, to a future publication[5].

11 Conclusion and Future Work

We presented TEA, a compact, sequence-transformer-chaining paradigm general-purpose computer programming language that is text processing oriented and which was designed with broad portability and usability in mind. TEA's minimal instruction set, bounded command space, and cross-platform interpreters make it a promising utility for low-resource programming, rapid prototyping, and domain-specific sequence analysis and transformation tasks. Future work includes:

- Reviewing the language in light of the contemporary global programming language standards as well as ACM SLE-oriented cutting-edge theory and breakthroughs in language design, implementation and evaluation.
- Expanding the TEA standard library (numerous currently *inert* instruction signatures could find better use) based on further research into usability, feature completeness and community-feedback or demands. For example, based on ideas presented in [27], we intend to extend TEA with currently missing important facilities such as meaningful support for runtime exception handling as well as considering how to provide support for multithreaded programming.
- Performance optimizations based on standardized language benchmarks, but also via rigorous industrial testing and language adaptation case studies.
- User studies on learnability and productivity in low-resource environments — especially for scenarios such as in UGANDA where the language originated and where it was initially intended to be heavily utilized.

Acknowledgments

Thanks to the TEA project contributors and the Nuchwezi Research community for feedback and testing. Special Thanks are also due to the ACM SIGPLAN-M project and community, members of which have offered some mentorship to J. Willrich especially during the tough streak in 2025 while trying to secure a PhD position at Oxford. Among these, we must mention Dr. Abhiroop Sarkar of ETH Zurich (Switzerland), but also the encouragement and correspondences with Gordon McKay Professor of Computer Science at Harvard (USA), Stephen Chong.

Appendix 1: Quick Primer on Transformatics

Presented here based on the original 1-pager version [20] of [19], the essential foundations of the mathematics underlying TEA — basically **transformatics**, are as such:

[5] A total of 32 distinctive and high-impact papers from the ACM SLE conference have already been reviewed by the inventor of the TEA language — see https://t.me/bclectures — and so, it only remains to distill and compile the said [meta-]review, and we hopefully shall complete that in the near future with paper[s] and a book.

11.1 Result 1: The Empty Sequence[16]

$$\Theta = \langle\rangle \implies |\Theta| = 0 \quad \wedge \quad \Theta \equiv \emptyset$$

11.2 Result 2: A Symbol Set[5]

$$\psi_\beta^n = \langle \beta_{i\in[1,n]} \rangle : n \in \mathbb{N} : \varkappa(\beta_i \in \psi_\beta) = 1 \quad \forall \beta_i \in \psi_\beta$$

11.3 Result 3: A Sequence[16]

$$\Theta^n = \langle \theta_{i\in[1,n]} \rangle : \mathbb{N} \times \psi_\beta : n \geq 1$$

11.4 Result 4: Sequence Cardinality[14][5]

$$\forall \Theta^n = \langle \theta_{i\in[1,n]} \rangle : \mathbb{N} \times \psi_\beta \quad \implies \quad \varkappa(\Theta^n) = |\Theta^n| = n \in \mathbb{N}$$

11.5 Result 5: A Sequence Symbol Set[11][5]

$$\psi(\Theta^n) = \langle \theta_{i\in[1,k]} \rangle : k, n \in \mathbb{N} \wedge k \geq 1 : \varkappa(\theta_i \in \psi(\Theta^n)) = 1 \quad \forall \theta_i \in \Theta^n : k \leq n : \varkappa(\psi(\Theta^n)) = k$$

11.6 Result 6: A Sequence Transformation[18][16]

$$\Theta^n = \langle \theta_{i\in[1,n]} \rangle : \mathbb{N} \times \psi_\beta \rightarrow \Theta^* = \langle \theta^*_{i\in[1,k]} \rangle : \mathbb{N} \times \psi_* : \psi_\beta \neq \psi_* \vee \psi(\Theta^n) \neq \psi(\Theta^*) \vee \overset{>}{\psi}(\Theta^n) \neq \overset{>}{\psi}(\Theta^*)$$

11.7 Result 7: A Sequence Transformer[21][18]

$$\Theta^n = \langle \theta_{i\in[1,n]} \rangle : \mathbb{N} \times \psi_\beta \xrightarrow{f(\Theta)=\Theta^*} \Theta^* = \langle \theta^*_{i\in[1,k]} \rangle : \mathbb{N} \times \psi_*; \quad \psi_\beta \neq \psi_* \vee \psi(\Theta^n) \neq \psi(\Theta^*) \vee \overset{>}{\psi}_\beta \neq \overset{>}{\psi}_*$$

11.8 Result 8: A Sequence Filter[18]

$$\Theta^n = \langle \theta_{i\in[1,n]} \rangle : \mathbb{N} \times \psi_\beta \xrightarrow{f(\Theta,\psi_{\beta^*})=\Theta^*} \Theta^* = \langle \theta^*_{i\in[1,k]} \rangle : \mathbb{N} \times \psi_{\beta^*};$$
$$k \in \mathbb{N} : \psi_{\beta^*} \subseteq \psi_\beta \implies \varkappa(\psi_{\beta^*}) = k \leq \varkappa(\psi_\beta) \implies \varkappa(\Theta^*) \leq \varkappa(\Theta^n)$$

11.9 Result 9: A Sequence Generator[18][10]

$$\Theta^n|\emptyset \xrightarrow{f(\psi_\beta,k))=\Theta^*} \Theta^k = \langle \beta^*_{i\in[1,k]} \rangle : \mathbb{N} \times \psi_\beta : k \in \mathbb{N} : \forall \beta_i \in \Theta^k \implies k \geq 1 \quad \wedge \quad \beta_i \in \psi_\beta = \Psi^\infty \cup \Theta^n$$

11.10 Result 10: A Sequence Sampler[9]

$$\Theta^n \xrightarrow{f(\Theta^n,k))=\Theta^k} \Theta^k = \langle \theta_{i\in[1,k]} \rangle : \mathbb{N} \times \psi(\Theta^n) : k, n \in \mathbb{N} \wedge k \geq 1$$

For anyone entirely new to, and also those interested in understanding what has motivated the formalization and distinction of transformatics as a subfield of mathematics in its own right, note that there is an entire book chapter (**Appendix E**) presented in [9], titled *Situating TRANSFORMATICS (as a Theory or Field) within Mathematics* that goes to great length demonstrating how this mathematics relates to, but also how it differs from other, existing and earlier branches; Calculus, Set Theory, Linear Algebra, Statistics, and even some Computational Mathematics domains such as Machine Learning and Signal Processing. So, given it is the mathematics underlying the TEA programming language, one might surely find much to learn [and appreciate] beyond the computing and software engineering [covered in this work but also elsewhere], by taking a moment to familiarize themselves with that material as well.

Appendix 2: Introducing TEA v.1.5.2 and TEA OS v1.1.0

It is with pleasure that we must welcome all interested readers to try out not just the most stable TEA language release, but also its hottest application/use-case to date... the TEA OS (TEA Operating System). A preview of what has been accomplished so far can readily be accessed via the TEA WEB IDE: https://tea.nuchwezi.com, however, the specifics concerning this particular milestone are:

This is the latest official reference implementation of the TEA runtime and documentation; TEA at v1.5.2, and with the latest fixes to the runtime including ensuring the delete instruction works as documented for multi-parameter regular expression patterns. This version of TEA is very robustly tested and proven, being the one currently powering the final, minimal, feature-complete, self-documenting TEA Operating System (TEA OS) also currently being included as part of the official TTTT package; at v1.1.0 right now. So, not only can one use TEA for the 'usual' tasks, but with TEA OS, one has a complete TEA-powered file-system, a suite of basic utilities including ability to do math and test simple and complex minified TEA programs from the TOS shell terminal, and the power to write, store, and later invoke stored TEA programs as system commands directly from TEA OS!

Also, among great observations while working with TEA OS is that all work done and stored can not only be readily exported outside of the OS (per session), but that also that for; stored files, these can later be updated, renamed, deleted or exported outside of the TEA OS as is. Like all TEA programs, the complete TEA OS can fit inside a string, and one can carry it as a lightweight, easier-to-use alternative to Linux, Unix or Windows/MS-DOS in just any modern browser or via the TEA mobile IDE without installing anything, no virtual machines, no sophisticated configurations... it just works!

TO try TEA OS without installing anything, use the hot-load-and-run URL: https://tea.nuchwezi.com/?fc=https://gist.githubusercontent.com/mcnemesis/276b40c96b52f846bf11214419912e3f/raw/&i=Please+Click+Run+In+Case+It+Does+Not+Auto+Run&run

The latest documentation on both TEA and TEA OS can be found in [8], and as for what it feels like working inside of the TEA OS right now, the following illustrative session dump might be worth reviewing or referring to:

```
ooooooo_________________
___oo_____ooooo___ooooo__
___oo____oo____o_oo___oo_
___oo____ooooooo_oo___oo_
___oo____oo______oo___oo_
___oo_____ooooo___oooo_o_
_________________________
TEA OS v1.1.0 is Ready...

Please set a Username:
Fut. Prof. JWL
___________________

Hello, Fut. Prof. JWL. Welcome to TOS!
___________________
TOS/21:35:42/>
ls
.
datefile
info.text
info
nugreetings
```

```
run
the_greetings
permutate
helloworld
about.tea
____________________
TOS/21:35:49/>
help

=o==o==o==o==o==o==o=
TEA OS v1.1.0 HELP System Commands:
about
(help | man | whatis | info | about) [NAME]
help-file
help-help
help-prog
help-ui
help-util
=o==o==o==o==o==o==o=

____________________
TOS/21:36:02/>
about

=o==o==o==o==o==o==o=
ABOUT: TEA OS v1.1.0
#---[ABOUT TEA OS]

NAME: TEA Operating System
(also "TEA OS" or just "TOS")

ARCHITECT: J. Willrich Lutalo <jwl@nuchwezi.com>

SINCE: 9th APRIL, 2026

INTENT: To manifest a true general purpose, lightweight, minimal but powerful
     operating system that can essentially just fit inside a single string; to see
     what the limits of the mathematics of TRANSFORMATICS might be, and what we
     surely could accomplish with its proper use via modern portable computers.

ABOUT: TEA OS is implemented using the Transforming Executable Alphabet (TEA)
     programming language. It is based on a specification first laid out in the TEA
     TAZ. It is a project originating from UGANDA.

PROJECT HOME: Nuchwezi Research | https://tea.nuchwezi.com
=o==o==o==o==o==o==o=

____________________
TOS/21:36:15/>
help-file

=o==o==o==o==o==o==o=
TEA OS v1.1.0 FILE System Commands:
create FILENAME
init FILENAME
write FILENAME
```

```
(rename | rn | mv | move) FILENAME NEWNAME
(copy|clone) FILE1 FILE2
exists FILENAME
find PATTERN
(show | cat | read) FILENAME
export FILENAME
(del | delete | rm) FILENAME
(ls | list | dir | files) PATTERN
=o==o==o==o==o==o==o=

__________________
TOS/21:36:25/>
help-help

To use or access help on any topic in the system, use any of the commands presented
      in the following Menu.

=o==o==o==o==o==o==o=
TEA OS v1.1.0 HELP System Commands:
about
(help | man | whatis | info | about) [NAME]
help-file
help-help
help-prog
help-ui
help-util
=o==o==o==o==o==o==o=

Each of those commands leads you to further information about help on other commands.
       For example

> help help-util

Will explain what the help-util command is all about.
__________________
TOS/21:36:37/>
uilog
TOS Log-Interface Mode Activated
__________________
TOS/21:36:56/>
help uimini

=o==o==o==o==o==o==o=
uimini <~ Configure TEA OS interface to render using mini-terminal mode. Setting
      persists across logins until changed. This mode is suitable for environments
      such as using TOS on some Desktop Computer Systems.
=o==o==o==o==o==o==o=

__________________
TOS/21:37:05/>
uimini
TOS Mini-Interface Mode Activated
__________________
TOS/21:37:11/>
help-prog
```

```
=o==o==o==o==o==o==o=
TEA OS v1.1.0 PROGRAMMING System Commands:
(runnable | register | add-program) FILENAME
(isrunnable | canrun) FILENAME
(run | exec | execute) FILENAME
(del-prog | delete-program | unregister) FILENAME
(update | update-program) FILENAME
PROGRAM
(help | man | whatis | info | about) PROGRAM
=o==o==o==o==o==o==o=

__________________
TOS/21:37:18/>
whatis program

=o==o==o==o==o==o==o=
PROGRAM <~ Executes PROGRAM as a system command if it exists --- first, if it is any
      of the standard commands in TEA OS, otherwise if it is a valid filename "PROGRAM
      " that is already recognized/registered as a user-created program - this later
      case similar to calling "run PROGRAM", otherwise shows error message command-not
      -found.
=o==o==o==o==o==o==o=

__________________
TOS/21:37:34/>
okay
--COMMAND NOT FOUND | INVALID COMMAND--
__________________
TOS/21:37:42/>
help-utils

=o==o==o==o==o==o==o=
TEA OS v1.1.0 UTILITY System Commands:
clear | reset
date
echo MESSAGE
exit
export
(fetch | web) [URL] [FILENAME]
logout
math EXPRESSION
print
tea CODE
time
timestamp
username NAME
whoami | name
=o==o==o==o==o==o==o=

__________________
TOS/21:37:48/>
date
Monday, 18 May 2026
__________________
TOS/21:37:52/>
whoami
```

```
Fut. Prof. JWL
__________________
TOS/21:37:57/>
programs
nugreetings
the_greetings
about.tea
__________________
TOS/21:38:01/>
cat about.tea
i!:Hello World
__________________
TOS/21:38:16/>
run about.tea
Hello World
__________________
TOS/21:38:23/>
tea n!:100|v:vN|r@:{Here's a random number: #vN#}
Here's a random number: 94
__________________
TOS/21:39:55/>
echo n!:100|v:vN|r@:{Here's a random number: #vN#}
n!:100|v:vN|r@:{Here's a random number: #vN#}
__________________
TOS/21:40:11/>
write rng.tea
DATA WRITTEN to File: rng.tea
__________________
TOS/21:40:30/>
whatis rng.tea

=o==o==o==o==o==o==o=
That help topic [rng.tea] is not understood or supported. Try the 'help' command to
      see what help commands there are. For example 'help whatis'
=o==o==o==o==o==o==o=


__________________
TOS/21:40:38/>
programs
nugreetings
the_greetings
about.tea
__________________
TOS/21:40:42/>
files
.
datefile
info.text
info
nugreetings
run
the_greetings
permutate
helloworld
about.tea
rng.tea
```

```
__________________
TOS/21:40:48/>
add-program rng.tea
User PROGRAM [rng.tea] is now REGISTERED.
CAN BE INVOKED as SYSTEM COMMAND.
__________________
TOS/21:41:01/>
whatis rng.tea

=o==o==o==o==o==o==o=
rng.tea <~ Is a valid user-registered [TEA] program. You might run or execute it by
      directly invoking "rng.tea" or via "run rng.tea".
=o==o==o==o==o==o==o=

__________________
TOS/21:41:07/>
run rng.tea
Here's a random number: 23
__________________
TOS/21:41:16/>
about move

=o==o==o==o==o==o==o=
move FILENAME NEWNAME <~ Change the name of an existing file from FILENAME to NEWNAME.
       WILL ALSO update the registered [user] programs list if FILENAME was a
      registered program. Valid FILENAME should consist of ONLY letters a-zA-Z,
      numbers 0-9, underscore _ and/or a dot .
=o==o==o==o==o==o==o=

__________________
TOS/21:41:29/>
move rng.tea rng
Program File rng.tea RENAMED rng
With [WRITE] status: DATA WRITTEN to File: rng
And [REGISTER] status: User PROGRAM [rng] is now REGISTERED.
CAN BE INVOKED as SYSTEM COMMAND.
And [DEL] status: DELETED PROGRAM FILE: rng.tea
With Status: UNREGISTERED PROGRAM FILE: rng.tea
__________________
TOS/21:41:41/>
whatis rng

=o==o==o==o==o==o==o=
rng <~ Is a valid user-registered [TEA] program. You might run or execute it by
      directly invoking "rng" or via "run rng".
=o==o==o==o==o==o==o=

__________________
TOS/21:41:47/>
rang
--COMMAND NOT FOUND | INVALID COMMAND--
__________________
TOS/21:41:50/>
rng
Here's a random number: 60
__________________
```

```
TOS/21:41:53/>
rng
Here's a random number: 76
____________________
TOS/21:41:58/>
time
21:42:04
____________________
TOS/21:42:04/>
name
Fut. Prof. JWL
____________________
TOS/21:42:07/>
export
=o==o==o==o==o==o==o=
Thanks for Using TEA OS v1.1.0
```

Listing 4. TEA OS v1.1.0 Work Session

References

[1] Mikhail Barash. 2020. Example-Driven Software Language Engineering. In *Proceedings of the 13th International Conference on Software Language Engineering (SLE '20)*. ACM, Virtual, USA, 1–7. doi:10.1145/3426425.3426945

[2] Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2018. Interdisciplinary Programming Language Design. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18)*. ACM, Boston, MA, USA, 1–15. doi:10.1145/3276954.3276965

[3] Laurent Huberdeau, Cassandre Hamel, Stefan Monnier, and Marc Feeley. 2024. The Design of a Self-Compiling C Transpiler Targeting POSIX Shell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2024)*. ACM, Vancouver, BC, Canada, 70–83. doi:10.1145/3687997.3695639

[4] Joseph Willrich Lutalo. 2020. **Introducing the Dynamic Nuchwezi Architecture Platform**. *Preprints* (May 2020). doi:10.20944/preprints202005.0207.v2

[5] Joseph Willrich Lutalo. 2024. **A general theory of number cardinality**. *Academia.edu* (Jan 2024). https://www.academia.edu/43197243/.

[6] Joseph Willrich Lutalo. 2024. **Software Language Engineering-Text Processing Language Design, Implementation, Evaluation Methods**. *Preprints* (2024). doi:10.20944/preprints202410.0636.v2

[7] Joseph Willrich Lutalo. 2024. **TEA GitHub Project | The Reference Implementation of TEA (Transforming Executable Alphabet) computer programming language**. https://github.com/mcnemesis/cli_tttt/.

[8] Joseph Willrich Lutalo. 2024. **TEA TAZ - Transforming Executable Alphabet A: to Z: COMMAND SPACE SPECIFICATION**. *Academia.edu* (2024). https://www.academia.edu/122871672/

[9] Joseph Willrich Lutalo. 2025. **Applying TRANSFORMATICS in GENETICS**. *I*POW* (Aug 2025), 1–313. doi:10.5281/zenodo.20092893

[10] Joseph Willrich Lutalo. 2025. **Applying TRANSFORMATICS: Sequence Generators**. https://www.academia.edu/143116027/.

[11] Joseph Willrich Lutalo. 2025. **Concerning A Special Summation That Preserves The Base-10 Orthogonal Symbol Set Identity In Both Addends And The Sum**. *Academia* (2025). https://www.academia.edu/129011291/.

[12] Joseph Willrich Lutalo. 2025. **Concerning Debugging in TEA and the TEA Software Operating Environment**. *Computer Science and Mathematics: Software* (2025). doi:10.20944/preprints202502.1506.v2

[13] Joseph Willrich Lutalo. 2025. **GAME STATE MACHINE (BOOBS 'N' TRAPS) - a single-player text-adventure and strategy role-playing game (RPG)**. https://doi.org/10.5281/zenodo.19479082.

[14] Joseph Willrich Lutalo. 2025. **Introducing The Anagram Distance Statistic, Ã, A Quantifier of Lexical Proximity For Base-10 o-SSI And Any Arbitrary Length Ordered Sequences, Its Relevance And Proposed Applications in Computer Science, Engineering And Mathematical Statistics**. *Academia.edu* (2025). https://www.academia.edu/130159505.

[15] Joseph Willrich Lutalo. 2025. ***NOVUS MODERNUS GRIMOIRE LUMTAUTO MAGIA***. I*POW. *A Modern Grimoire of 5 Eternal Magickal Languages*. https://www.academia.edu/145086513.

[16] Joseph Willrich Lutalo. 2025. **Philosophical and Mathematical Foundations of A Number Generating System: The Lu-Number System**. *PrePrints* (2025). doi:10.20944/preprints202506.0763.v1

[17] Joseph Willrich Lutalo. 2025. ***Rock 'N' Draw***. I*POW. https://www.academia.edu/128601975/ ***ISBN*** *978-9913-624-71-8 (First Edition). A physical copy might be accessed via the National Library of Uganda (NLU), but also, an education-purposes-only edition is accessible freely via Telegram at* https://t.me/ipowriters/237.

[18] Joseph Willrich Lutalo. 2025. **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). https://www.academia.edu/136852057.

[19] Joseph Willrich Lutalo. 2025. **TRANSFORMATICS 101 - explained**. *Thesis* (October 2025), 20. https://www.academia.edu/144344109/

[20] Joseph Willrich Lutalo. 2025. **Transformatics 101 (1-Page Executive Summary)**. https://www.academia.edu/145185478/.

[21] Joseph Willrich Lutalo. 2025. **Transformatics for PSYCHOLOGY**. *PrePrints* (9 2025). doi:10.20944/preprints202510.0456.v1

[22] Joseph Willrich Lutalo. 2025. **Unraveling Mysteries of The ZHA q-AGI Chatbot: an Interview by ICC, of Fut. Prof. JWL and M*A*P Ade. Psymaz of Nuchwezi**. *Academia* (2025). https://www.academia.edu/129359195.

[23] Joseph Willrich Lutalo. 2026. **Applying TRANSFORMATICS in GENETICS: Sequence Analysis Using The Anagram Distance and the Modal Sequence Statistic**. In *Applying Transformatics in GENETICS*. Zenodo. doi:10.5281/zenodo.19995017

[24] Joseph Willrich Lutalo, Odongo Steven Eyobu, and Benjamin Kanagwa. 2023. **DNAP: Dynamic Nuchwezi Architecture Platform-A New Software Extension and Construction Technology**. *authorea* (2023). doi:10.36227/techrxiv.13176365.v1

[25] Joseph Willrich Lutalo and Tonny Oyana. 2024. **VOSA: A Reusable and Reconfigurable Voice Operated Support Assistant Chatbot Platform**. *Computer Science & Information Technology (CSIT)* 14, 8 (April 5 2024), 63–78. doi:10.5121/csit.2024.140803

[26] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam T. Kalai. 2013. A Colorful Approach to Text Processing by Example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, St. Andrews, United Kingdom, 349–358. doi:10.1145/2501988.2502040

[27] Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of the 13th International Conference on Software Language Engineering (SLE '20)*, Ralf Lämmel, Laurence Tratt, and Juan De Lara (Eds.). ACM, Virtual, USA, 72–85. doi:10.1145/3426425.3426933