Article

Not peer-reviewed version

Software Language Engineering - Text Processing Language Design, Implementation, Evaluation Methods

Joseph Willrich Lutalo *

Posted Date: 6 December 2024

doi: 10.20944/preprints202410.0636.v2

Keywords: Programming Language Engineering; Language Design; Language Implementation; Language Evaluation; Domain Specific Languages; Text Processing Languages; TEA

Article

Software Language Engineering—Text Processing Language Design, Implementation, Evaluation Methods

Joseph Willrich Lutalo

ICT Research Lab, Nuchwezi, Email: joewillrich@gmail.com

Abstract: Programming languages drive most if not all of modern problem-solving using computational methods and power. Research into new programming languages and techniques is essential because it makes the design, implementation, and application of automation to general or particular problem-solving ever easier, more accessible, and more performant. GPLs typically are designed to be purely domain agnostic—meaning they can be applied in any field, for any problem kind. However, this normally also makes them hard and difficult to use in problems where non-programmers or even experts with little or no GPL programming skills are required to leverage programmatic problem-solving capabilities, which is why DSLs come into play; they are generally more fine-tuned toward improving human productivity and performance than that of the machine, while making solving particular, domain-oriented problems simpler. In this paper, we review the literature concerning how to design and then fully implement a new DSL, with a special focus on a DSL for generic problem-solving leveraging Text Processing methods—a Text Processing Language (TPL). We consider leveraging the Design Research paradigm as a systematic framework for guiding research into the development of new TPLs. We present for the first time, a new unifying theory concerning general, but also TPL-specific language engineering theory and guiding frameworks—UPLT, PLEF & PLEf. With a re-introduction of the SOE framework, we consider quantitative and qualitative evaluation of software languages, with specific focus on programming languages. Finally, we highlight identified pending problems for future theoretical and pragmatic research into the field of language engineering, especially with a focus on TPLs.

Keywords: programming language engineering; language design; language implementation; language evaluation; Domain Specific Languages; Text Processing Languages; TEA

---

1. Introduction

We can deal with complexity by simplifying it[1]. The sole purpose of all computing is to simplify the complexities of reality, by creating hierarchies of useful abstractions—what Kain calls "illusions"[1], that hide away the intricacies of solving a complex problem by offering simpler [artificial] interfaces to otherwise unwieldy, but unavoidable [natural] complexity that must be dealt with so as to allow humans to manifest some solution or a series of them. Like how an architect designs and manipulates space and materials to create useful spatial illusions—in the form of buildings, boundaries and other physical interfaces [2,3]—such as what Field refers to when he says `Architecture is the one art that we cannot avoid... constantly before our eyes, indeed we live in its works, in the sense that all buildings are designed or planned`[2], so a computer architect, a computer engineer and for the sake of this paper, a language engineer or architect, designs and creates an abstraction in the form of a computer software language or rather an "engineered language"—to distinguish it from a *natural language*[4], with which otherwise complex problems become simple and more natural or intuitive to solve *inside of*.

Just like innovations and breakthroughs in computer hardware and physical electronics define or determine what we can accomplish with physical computation and engineering [5], innovations and advances in software help give life to that hardware as well as make some existing hardware obsolete or perhaps extend its use to solving arbitrary, old and new problems in simpler and better ways. Just like advances in semi-conductor chips, micro-controllers and materials power the evolution of computer hardware, advances in software languages—and for the rest of this paper, specifically, software programming languages (or just "programming languages"), libraries and Application Programming Interfaces (APIs) power progress in computer software.

The design and implementation of computer software is central to the application of computer science and software engineering to general problem solving using automation, while the design and implementation of programming languages underlies progress in computer science as a field [6][7], and is the stuff that makes software engineering possible at all.

The availability of tools typically determines what problems man can solve, as well as how those problems can be solved. It is important to note that much as every able human is endowed with the ability to speak and listen to natural languages, yet, only a trained human, equipped with special, formal languages such as mathematics can begin to reason about, think-through and solve certain kinds of problems—with ease especially. So, just like knowledge of mathematics makes it easier for an engineer to design, implement, validate and apply sophisticated constructs such as space-ships, inter-continental ballistic missiles or military code-breakers, likewise, knowledge of special computing languages such as Assembly, C, LaTex or BNF empower people to think about and solve certain otherwise difficult problems [in computing] via simple, straight-forward and systematic ways.

1.1. *The Relevance of Language Research*

First, we should appreciate that many programming languages—general or domain-specific, were first inspired by real-world, practical problems. For example, in Oliveira's 2009 paper treating of the advantages and disadvantages of, as well as the development methods used to create Domain Specific Languages (DSLs)[7]—sometimes also referred to as "specific purpose languages"[8], we see that typically, a Domain Specific Language (DSL) will help a domain expert better solve problems in their domain, better than if they had used a General-purpose Programming Language (GPL)[7]. So, it is not that there aren't already enough languages with which to solve problems, but, necessity and creativity will many times drive language engineers and computer scientists to come up with new or different ways to approach the solution of some problems, using new programming languages and methods.

Of course, it is easier for most researchers and/or problem solvers to merely take an existing language and apply it to solving their problems than it is to design, implement and come up with a new programming language. We should definitely note that in general, language development is hard, even for small or specialized languages, and it typically entails wide and deep knowledge of one or more special domains, as well as technical and theoretical knowledge of language engineering or development [9].

Also, merely developing a language is not enough. Many times, a new language needs to be evaluated and justified both empirically and conceptually or ideologically. However, concerning this, it is for example well known that tasks such as the quantitative validation of DSLs, in general, but also in particular cases is hard and an important open problem [9]. Concerning qualitative evaluation of a new language, it would help to give attention to language properties such as its syntax for example; contemporary work on language engineering does inform us that it is surely without doubt that in the task of learning new [programming] languages, the burden of having to learn and master a language's syntax/grammar before it can be well applied is mostly unavoidable and is "a major major barrier to novices learning programming languages, but also the first encountered" [4]. Thus, when looking at why it is important to study, research and evolve programming languages, such matters as how usable or learnable a language would be once implemented much necessitate extensive investment in this field.

Concerning why investment in developing a new language, especially a DSL might be important, note that many task automation programmers often spend much time and efforts on GPL programming tasks that are tedious and follow the same patterns. In such cases, the required code or solution could be better arrived at using automatic-code generation via such approaches as application generators—a

 doi:10.20944/preprints202410.0636.v2

kind of special compiler in a way, or via the use of an appropriate DSL [9]. In this vain, interesting contemporary methods for assisting non-experts in programmatic problem-solving using both GPL and DSL coding assistants include the use of artificial intelligence assistants in the form of programming [co-]assistants, such as the Github Copilot project [10], Microsoft's Copilot [11] and the more popular ChatGPT [12]

1.2. MLs vs GPLs vs DSLs

Concerning programming languages for problem solving, we basically have three major categories of languages;

- A: **Machine Languages**:
  - Operate at a very low abstraction level.
  - Express solutions in a style not meant for direct-human comprehension.
  - Typically are written in a numeric/binary or opcode/mnemonic or assembly-code syntax.
- B: **General-purpose [Programming] Languages**:
  - Employ a sufficient amount of abstraction above the machine/processor.
  - Express solutions in a style meant for easier human comprehension and are great at expressing human logic.
  - Typically are written in a humane syntax close to mathematics.
- C: **Domain Specific [Programming] Languages**:
  - Are typically very high in abstraction and very close to the problem domain in terms of their operation.
  - Express solutions in a style close to natural human language.
  - Their syntax is also very close to the problem domain.

We can see from this classification, that classes of programming languages in general vary depending on their level of abstraction above the machine/computer or rather physical processors upon which they operate [1][13] and their target [problem] domain [7]. In **Figure 1**, we visualize this variance in both abstraction and humanness—which helps one to easily appreciate the distance, even where merely conceptual or symbolic, between low-level languages meant to assist in say system programming, to high-level languages meant to simplify the production of solutions in a particular problem domain. In general, we have (A) Machine Languages (MLs), (B) General-purpose languages (GPLs) and then (C) Domain Specific Languages (DSLs). In this paper though, we shall focus on the relations between GPLs and DSLs, and an important intermediate class or category of TPLs—Text Processing Languages, that might sometimes fall in the GPL or DSL category.

It is formally understood that GPLs are designed from the ground-up without any particular domain, application area or problem in mind apart from creating a mechanism for humans to translate arbitrary solutions to any problem into algorithms and sets of instructions we call computer programs, that can then be presented to a [physical or abstract] computer, which then knows to parse, comprehend and execute the instructions in the program so as to solve some problem. DSLs on the other hand, are typically designed with a particular domain in mind, or at least with some systematic constraints imposed on the language so that it makes solving problems in a particular way and especially for a particular domain or problem space easier than it would typically be when using a GPL. In Essence, both DSLs and GPLs are computational problem solving tools, however, it is easy to see that whatever a DSL can do, a GPL could also have done, though, as one might find when exploring the literature about DSLs [7,9,14], DSLs are typically designed to make simpler, the process or method of solving certain problems which would otherwise have been very arduous to solve using a GPL. An example is attempting to produce a polished modern website using only the C programming language—which is not only a very mature, but also very popular GPL, compared to using specialized web-development, design and presentation languages such as CSS, JavaScript and HTML, or at least leveraging a generic text-processing language such as Python [15], Perl or PHP.

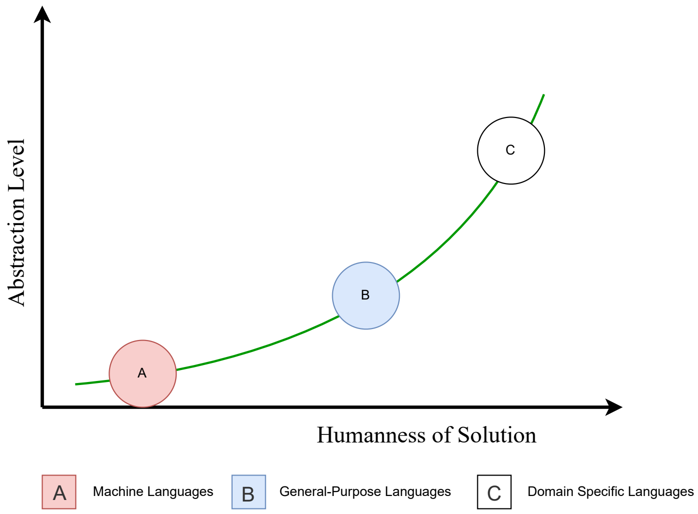

Figure 1. Visualizing Abstractness Vs Humanness of Programming Languages

In this paper, we shall undertake the task of exploring the design, implementation and then evaluation of any DSL. Also, we shall give special treatment to a particular, important sub-class of DSLs meant for text-processing—which we are to refer to as Text Processing Languages (TPLs). The importance of research into DSLs can readily be appreciated by the success and wide-spread use of both ancient and new DSLs such as CSS [16], Dot [17] or Latex [18]. We for example find that much as the language CSS was originally designed for styling markup documents such as HTML pages, it is currently also being used for animations, graphics and enabling non-visual alterations to web pages [19,20].

1.3. Why TPLs?

The Oxford dictionary of computing defines **text processing** thus:

> All forms of text manipulation including word processing [21].

Text Processing, which happens to be a major sub-category of most programmatic problem-solving leveraging computation [22–25], deserves and does get its rightful treatment across the programming language landscape since the earliest generations of computers [13,26], but is also not without special treatment even in the contemporary computer science and software engineering landscape [27], and even in an age where much of data processing is happening inside of/via AI models such as Large Language Models, Transformers, GANs and such, yet still, research shows that investment in careful text-processing—such as a pre-processing phase to advanced computations, is giving others an edge in performance compared to those merely using AI or machine learning without it [28].

First, note that text processing, tedious and error-prone even for programmers, remains one of the most important areas of research in applied computing [29]. We for example find that AWK, a traditional and popular TPL is thought to be problematic sometimes, because of being line-oriented, limited to regular expression patterns, and unable to use external parsers [30]. Other traditional tools such as Sed likewise come with their subtleties that could be limiting sometimes; for example, Sed is a non-interactive stream editor that will typically accept input from an existing file or perhaps redirected STDIN, and then strictly output to STDOUT [31]. Also, we find that in some cases, having to explicitly write a text processing program for solving some problem is considered wasteful or even infeasible. Thus have we seen the emergence of alternative approaches to text processing such as the STEPS project and language [29]—which allows a user to edit an example text by hand, and then have a machine-learning-powered system produce a program to perform the same or similar edits on other

 doi:10.20944/preprints202410.0636.v2

[similar] texts [29] automatically. Yessenov et al. argued that such Programming By Example (PBE) systems are easier to learn and yet they lack the arcane syntax of using a traditional TPL [29].

Also, there are some practical scenarios where use of traditional text processing tools—such as GREP, Sed or AWK, might be overly constrained, and we need new approaches or methods. For example, we find that STEPS is designed to handle hierarchical text that might not only span multiple lines in the input, but also need to span multiple lines and preserve hierarchy or structure in the output [29]. The newly developed TEA[1]—**T**ransforming **E**xecutable **A**lphabet computer programming language, a TPL by design, allows new approaches to solving critical, fundamental computing problems such as the generation of random numbers, alternating between visible and invisible text, random string generation, statistical analysis and data quantification among others, merely by leveraging simple transforms on pure text and no sophisticated mathematics or physics involved [27,32].

Also, we find that, unlike many other families of computing utilities, text-processing finds use in many, if not most of general high-level programming—arguably the domain of GPLs, so that, it is almost impossible to come across a serious GPL that does not come with some in-built library, module or set of instructions for performing some fundamental text-processing—these are usually the routines found in a language's standard string manipulation library, such as *string.h* for C [33], *string* for Python [34,35], *Strings* for Java [36], *String::Util* for Perl [37], etc. This means, for cases where one might not immediately have access to a [traditional] GPL, or where they wish to not use one, having access to a mature and/or robust TPL can make general problem solving much easier, and that is where languages like TEA are destined to shine [27].

2. Concerning the Design of DSLs

The decision to develop a DSL is often postponed indefinitely if considered at all, and most DSLs never get beyond the application library stage [9]. It is helpful and natural to think of a DSL in terms of a gradual scale, with very specialized DSLs such as BNF (itself used to design or implement both GPLs and DSLs) on the left, and GPLs such as C++ on the right [9]. We have also seen, in **Figure 1**—which somewhat generalizes this observation first made by Mernik, how the abstraction level and humanness of the solutions expressed using a language determine or hint at the class of language it belongs to. Clearly, this is mostly to do with a language's syntax, but also has a bearing on its semantics. Definitely, that classification should also help to guide DSL designers at a high-level, to ensure that the language they set out to design or implement, doesn't defeat these meaningful and helpful principles.

In the rest of this section, let us look at some of the quirks, ideas and principles underlying the design of most DSLs, TPL or not.

2.1. A Preamble to All Language Design

In this section, which in a way could be considered independent of much of the rest of this paper—not because the ideas here were developed last in the present work, but also because, it has been established by the author, they would help clarify much of the rest of this paper, plus also offer it the most meaningful theoretical basis for critical discussions. For the sake of generality, we shall refer to the ideas developed in the rest of this section as the Unifying Programming Language Theory (UPLT).

2.1.1. A Unifying Programming Language Theory

First, before we consider anything, let us start by revisiting the modern foundations of all computer science. Basically, let us recall that the generally agreed theoretical basis of all modern computing is the notion of the Turing Machine, which the Oxford dictionary of computing helpfully defines as such [21]:

---

[1] The Reference Implementation & TEA documentation at **https://github.com/mcnemesis/cli_tttt**

 doi:10.20944/preprints202410.0636.v2

Definition 1 (Turing Machine(TM)). *An imaginary computing machine defined as a mathematical abstraction by Alan Turing to make precise the notion of an effective procedure (i.e. an algorithm). There are many equivalent ways of dealing with this problem; among the first was Turing's abstract machine, published in 1936.*

A Turing machine is an automaton that includes a linear tape that is potentially infinite (in both directions), divided into boxes or cells, and read by a read-head that scans one cell at a time. Symbols written on the tape are drawn from a finite alphabet: $s_0, ..., s_p$

The control or processing unit of the machine can assume one of a finite number of distinct internal states: $q_0, ..., q_m$

The "program" for a given machine is assumed to be made up from a finite set of instructions that are quintuples of the form $q_i s_j s_k X q_j$ *where X is R,L, or N.*

The first symbol indicates that the machine is in state q_0*, while the second indicates that the head is reading* s_j *on the tape. In this state the machine will replace* s_j *by* s_k *and if X = R the head will move to the right; if X = L it will move to the left and if X = N it will remain where it is. To complete the sequence initiated by this triple the machine will go into state* q_j*.*

The machine calculates functions...

With that essential introduction then, let us take a moment to consider and reflect upon the following results the author puts forward:

Theorem 1 (Text is Everything). *All programming is text processing.*

Proof. Programming is the expressing of a solution for some task T, by an expressible algorithm P(T) in finite time. Assuming T* is some programmable task—a problem, then there exists some text expression S(P(T*)) with which P(T*) can be expressed. Then processing S(P(T*)) always produces the solution to T*. □

Lemma 1. *All programs are text.*

Proof. Follows from **Theorem 1**, and the fact that for any Turing Machine or computer for which a program P(T) solving task T can be expressed by some text S(P(T)) = A(T), A(T) is essentially equivalent to P(T). □

Figure 2. First Law of UPLT

Concerning the matter of programming Turing Machines, or rather, computers, let us not forget that a DSL, just like a GPL or any programming language for that matter, is meant to be used to express/write computer programs, and that these programs are generally nothing but mere code or rather source-code, and that **all source-code is nothing but mere text!** More precisely, a program in any language, DSL or not, is a kind of well structured or rather regularly structured piece of text—whether or not it is a combination of data and instructions doesn't matter, because, at the most abstract level, **a computer only does anything useful, upon reading some input**—fundamentally and generally so (refer to **Definition 1**), as text, and whether or not a piece of input text is instruction or data only depends on context and the nature of the computer processing the text. Much of this shall readily following from a basic understanding of some fundamental computer science concepts as Abstract Machines, Finite-State Machines and especially the Turing Machine, upon which most, if not all of the currently meaningful and practical computers and their languages are founded. Before we proceed, let us also take a moment to consider the post-Turing ideas of abstract machines and computers as shall help in the general appreciation of the theory and ideas we have embarked on introducing as well as developing. Kain, in their definitive book on Advanced Computer Architecture has this to say [1]:

> A machine with a separate program memory is often called a *Harvard Machine*, because the first computers built at Harvard in the 1940s used a separate paper tape for their programs; this tape was logically similar to a separate read-tape for their programs; this tape was logically similar to a separate read-only program memory. Machines that intermix

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

> programs and data in the same memory are called *von Neumann Machines*, or *Princeton Machines*, because the first machine built by von Neumman at Princeton placed the program and the data in the same memory unit. Sharing the same memory has an allocation advantage...

Thus, irrespective of what language, level or domain one sets out to conceptualize, design or implement a formal computer programming language for, keeping in mind that **the language itself is defined using text, is implemented using text and while being executed or run, is processed as text processing text**, shall help clear-up many illusions and confusion, as well as help establish the essential fundamentals for everyone, once and for all.

Next, considering that all programming occurs via the use of a programming language, we arrive at the following important result too:

Theorem 2 (The TPL Law). *All programming languages are text processing languages.*

Proof. First, a program can be a combination of both the data (including types) to be processed as well as the instructions to process them. Then, assuming a programming language operates on some type other than text, yet, whatever type that is, should likewise be expressible using text for it to be programmable or rather computable—follows from **Lemma 1** and the first argument. □

Figure 3. Second Law of UPLT

In general, we find that a computer is an abstraction for any construct capable of operating on some input following a set of instructions specified in a program, and then producing some output or effect. So, to sum this up, we also have the results depicted in **Figure 4**.

Theorem 3 (General Computation). *All computation is text processing.*

Proof. Follows from fact that all computable operations and tasks can be expressed by or are reducible to mere text and some processing on it. □

Lemma 2. *All computable things are text.*

Proof. Since **Theorem 3** is true, directly follows. □

Figure 4. Third Law of UPLT

As this is a work on language engineering, we should avoid the temptation to dive deep into philosophy or [computational] metaphysics, though, it doesn't hurt to digress a bit here—and usefully so, before we proceed with our main subject.

First, note that one might wonder... Concerning the matter of Human Computer Interaction (HCI), human computers and the rather strange but ancient [occult] metaphysics such as the Kaballistic idea that all reality is reducible to and controllable via numbers [38,39]. By extension, especially given numbers could be transformed to "text" so to say—and vice versa [38], interesting applications and problems of this current exploration in a new philosophy for computing could crop up, and justifiably so! However, just to motivate further interest in this topic and UPLT in general, let me request the interested reader to consider or ponder some of the mostly philosophical problems concerning text, text processing and reality in **Figure 5**

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

1. Assume humans are a kind of computer in the TM sense—even remotely so, could it be accurate to conclude that all their [mental] processing is a kind of text processing?
2. By extension, since everything such a computer can process is text or reducible to text, can it be correctly claimed that any knowable or rather, *computable* reality is a kind of [latent] text?
3. How might mere text be transformed into palpable concrete reality readily[a]?
4. Does there exist some text that if read by a human, would always cause them to die? go mad? freeze or loose consciousness even if momentarily?
5. Do human computers react systematically and predictably to any actionable text?

[a] Either during sleep or during waking states—so-called "Affectant Metaphysics"[40]

Figure 5. Some Philosophical Problems relating to UPLT

For now, consider that all sciences at the most general level, can be categorized under either **natural sciences**, **formal sciences** or **social sciences**[41]. Next, consider that in the present era, many such sciences either directly or indirectly leverage computing in their theories and applications—so-called *Computational Sciences*[42], we can then come to appreciate the true and fundamental importance of research on text processing, by considering how much this idea powers much of the computational sciences, and in particular, let us call out a few examples from the biological sciences, or more specifically from *computational biology*[43]; we shall call out one interesting example of how text processing is applied in this field, by looking at the relevance of this idea in the domain of genetic engineering and DNA-sequencing [44,45]. We see, in one authoritative teaching manual on Bioinformatics and Computational Biology [46]:

> **Sequence Analysis:**
> The application of sequence analysis determines those genes which encode regulatory sequences or peptides by using the information of sequencing. For sequence analysis, there are many powerful tools and computers which perform the duty of analyzing the genome of various organisms. These computers and tools also see the DNA mutations in an organism and also detect and identify those sequences which are related. Shotgun sequence techniques are also used for sequence analysis of numerous fragments of DNA. Special software is used to see the overlapping of fragments and their assembly.

That's on page 13 of that interesting manual, and then in a section on Proteins—page 46, we see that:

> **Protein:**
> Protein database maintains the text record for individual protein sequences, derived from many different resources such as NCBI Reference Sequence (RefSeq) project, GenbBank, PDB and UniProtKB/SWISS-Prot. Protein records are present in different formats including FASTA and XML and are linked to other NCBI resources. Protein provides the relevant data to the users such as genes, DNA/RNA sequences, biological pathways, expression and variation data and literature. It also provides the pre-determined sets of similar and identical proteins for each sequence as computed by the BLAST. The Structure database of NCBI contains 3D coordinate sets for experimentally-determined structures in PDB that are imported by NCBI. The Conserved Domain database (CDD) of protein contains sequence profiles that characterize highly conserved domains within protein sequences. It also has records from external resources like SMART and Pfam. There is another database in protein known as Protein Clusters database which contains sets of proteins sequences that are clustered according to the maximum alignments between the individual sequences as calculated by BLAST.

With that in mind, and back to our original discourse concerning language engineering, since the motivations for the above theoretical explorations stemmed from our interests on the analysis, design and implementation of programming languages, let us complete this section on the UPLT by studying

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

and keeping in mind the important schematic depicted in **Figure** 6, that sums up all sorts of language engineering work, with a single model or framework depicted in a single picture, and which we shall call “The Programming Language Evolution Framework"—PLEF.

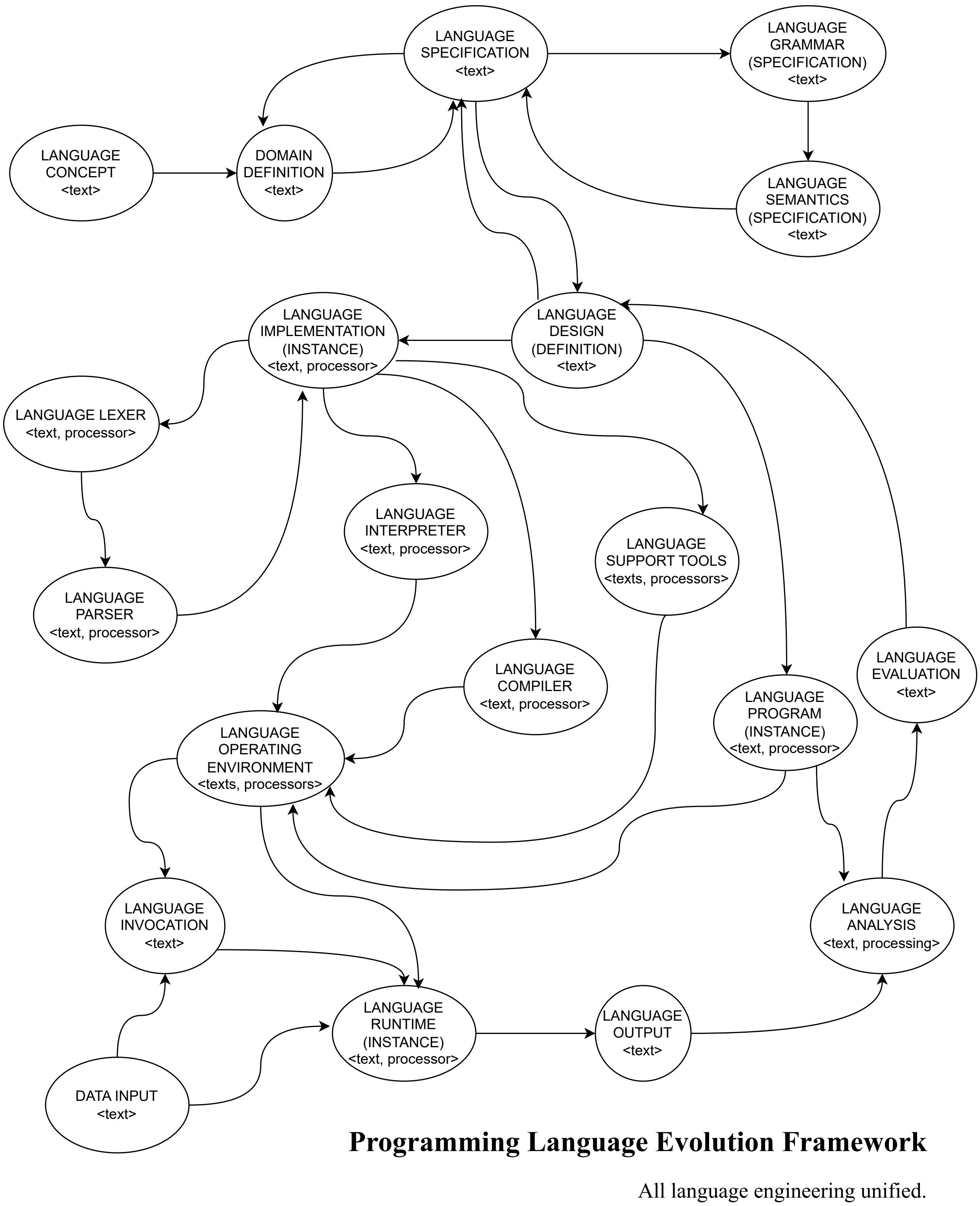

Figure 6. The PLEF as a State Diagram

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

2.2. Designing Any DSL

Concerning methods of bringing a DSL to life, it is important to note that in combination with a relevant application library, any GPL can implement or act as a DSL [9]. For example, the TMIL (Text Manipulation Imaging Language) by Hamburger et al. is said to essentially simplify what could have been achieved merely by using the Java Paint2D or C++ GD2 libraries[8]. Of course, this doesn't mean that every DSL out there is merely some off-shoot of a GPL, or that any DSL can simply be reduced to or obsoleted by a GPL even if such might be possible or conceivable with sufficient effort and general programming dexterity. Also, unlike GPLs, DSLs need not be executable [9]. We for example find that one of the more popular DSLs used in the domain of systems analysis and design, the Unified Modelling Language (UML)[47], is generally not readily executable, and is not really meant to be executable, much as it is meant to be used to design or describe executable systems!

Otherwise, many DSLs come to life via traditional language development methods and principles, most of which merely help one to either specify, design or implement the language's formal/regular structure. This regular structure shall usually be expressed via a formal/regular language, and this expression is what determines the grammar and syntax of the language, and from this the semantics of the programs expressed in the language are then derived. Programs written using the language then, shall merely be formal contracts, declarations, specifications or definitions of some sort, that conform to the language's grammatical, syntactic and semantic rules so as to solve some problem[2]. This is essentially the core, gist of it all—refer to the **PLEF** in **Figure 6** for a general overview of this process.

Concerning traditional methods for such language development, and especially considering some of the popular methods used, tools and technologies such as Lex [48] and Yacc [49]; the former for building lexers, the later for implementing parsers [14] are quite well-known and well-documented with regards to helping verify or process regular language expressions. This shouldn't come as a big surprise, because, since the early generations of modern computing, it became clear that instead of leaving every new language design and development project to its own methods and toolset, the design and implementation of some generic, re-usable language engineering tools—Make [50], Yacc, Lex, ANTLR, BNF [51] and UML [47] being great examples that would help solve a very fundamental problem in the field of language research and implementation. In their famous "**Dragon Book**" on general programming language development, and particularly traditional compiler design[52], Aho et al. tell us that:

> Students work... create and implement a little language of their own design... student-created languages have covered diverse application domains including quantum computation, music synthesis, computer graphics, gaming, matrix operations and many other areas. Students use compiler-component generators such as ANTLR, Lex, and Yacc and the syntax directed translation techniques... to build their compilers.

In general though, with or without helper tools, we find that the systematic approach to implementing a new DSL involves the following key steps [14]:

1. Defining the Domain.
2. Designing a DSL that accurately captures the domain semantics.
3. Building Software Tools or Software Components to support or realize the DSL.
4. Developing applications (domain instances) using the newly developed DSL infrastructure so as to verify and evolve the DSL.

In terms of implementation approach, we find that a DSL is better implemented as an interpretable language than as a compilable one [14]. Concerning this, it should be interesting to note that DSLs exist that are capable of being interpreted, but also which can be compiled—many times, not into an ML as

[2] Defines a computer program!

 doi:10.20944/preprints202410.0636.v2

would be the case of a GPL, but into some other DSL or some Intermediate Language(IL)—examples in this special category include SQL[53] and YAML[54]—the language F#[55] would also fall into this queer category, but is a GPL not a DSL.

Arguably, whether or not a language is best designed/implemented as compilable or interpretable might depend on several critical factors such as:

1. Whether the language is to be used in a stand-alone context.
2. Whether the language is to be embedable in other programs of other languages.
3. Whether the language is meant to operate in user-space–such as for GUI systems, or is for low-level/system-level programming tasks—such as boot-loaders, system configuration or part of complex automation tasks such as are implemented with tools like Gradle[56] (which uses languages Groovy[57] or Kotlin[58]) or Ansible [59] (which uses YAML).

However, especially while still prototyping or evaluating a language, it might make most sense to first approach it as an interpretable language, and this has its compelling advantages even for mature languages—interactivity, such as we see for languages like Python[35] via its shell or for LISPS via a REPL(Read Eval Print Loop) interface being some of the great examples[60] when such is feasible.

2.3. The Case of Designing TPLs

Still concerning the design of DSLs, let us take a moment to give special thought to the class of DSLs meant for text-processing. We for example find that, because a typical TPL is created for the sake of operating on data, in particular, text or rather, the string data type, the methods and principles behind the design of such languages have much to do with the data they are meant to operate on and/or how they are meant to operate on it.

TEA for example, is a generic DSL for text manipulation and/or transformation[27], but the more correct classification given the existing literature is to consider it a Text Processing Language (TPL) since text processing encompasses operations on the appearance of text, but also on the structure of text [61][62].

We find that many modern languages and tools for performing automated text processing employ methods and concepts such as regular expressions, string manipulation primitives, parsers and/or generators [29]. These approaches are to be found in most of the traditional text processing utilities/languages such as R, AWK, Sed, Perl and Python [23,31,63–65], and could thus be considered fundamental, if not essential primitives in any serious TPL.

It should be interesting to note that Text Processing on its own is such an important problem that some traditional GPLs such as Perl (Practical Extraction and Report Language) originally started out as a mere text processing utility [65]. This should defintley hint at the interesting fact that, a well designed TPL could many times end up becoming all one needs to perform tasks which would otherwise be generally relegated to a GPL. For this matter, and for completeness's sake, we could summarize the key attributes any robust TPL should possess with the following list of key TPL attributes:

1. A mechanism to read text into the program—if not during runtime, at least at program initiation or invocation.
2. A mechanism to output or write text from the program—this could be merely writing to standard output (such as onto the screen, printer or over the network—e.g. to a networked projector), but also to more generic data-sinks such as files on the local or a remote/network file system.
3. A mechanism to search for patterns in a text or generally, a string.
4. A mechanism to replace or overwrite sections of or the entirety of a string.
5. A mechanism to produce new text from other text—such as by the combination of multiple strings into one.
6. A mechanism to fragment or split up strings—with or without explicit patterns.
7. A mechanism to quantify strings—essentially, mapping a string to some numerical quantity, a number, such as by computing its length, or some other identifying metric such as a string's unique hashcode.

 doi:10.20944/preprints202410.0636.v2

8. A mechanism to compare or contrast two or more strings—such as the ability to test strings for equivalence based on exact contents or by some regular pattern as is possible with regular expressions.
9. A mechanism to reduce a large string to a smaller one—for example, by eliminating trailing white-space (typically known as stripping or trimming), eliminating certain sub-strings such as punctuation, redundancies, or perhaps automatically summarizing a long text etc.
10. Performing some standard transform on a string, such as toggling an entire string to uppercase or title-case, etc.

The above list of TPL attributes, though probably not exhaustive, can be backed-up by the design and implementation characteristics of several mature and/or main-stream TPLs such as Perl, Sed or Awk, but also by the text-processing facilities in many GPLs such as C, Python, JavaScript, C#, etc. (refer to **Section 1.3**). Interesting to note, the newly developed TEA language fully and exhaustively implements that list of TPL attributes, and for the sake of evaluating pure TPLs, could be among the best living examples at the moment.

3. Implementation of a DSL

In the introduction of this paper, we've already seen that generally, language development is hard. We have also seen that it is easier for most researchers and language-users to merely take an existing language and apply it to solving their problems than it is to design, implement and come up with a whole new language. In all cases where the design or development of a language is involved—DSL or not, we might consider it to be a case of *language-engineering*, to distinguish it from the more general software engineering[7] for which it is a sub-domain, albeit a more fundamental one since all software-engineering in some way relies on the results of language-engineering.

In the next section, let us dive a bit deeper than general language development, and instead focus more on what is known concerning the implementation of DSLs.

3.1. Theory on DSL Implementation

3.1.1. It Is Generally Complex

First, concerning the complexity of implementing a DSL; it can't readily be said if the same challenges face the implementation of a particular DSL for a particular domain, than it might be for another. However, as Mernik has stressed with regards to DSL implementation in general[9], it is hard, however, though he doesn't offer explicit arguments why this is so—though his work gives that claim some authority, yet, in a 1997 review paper on DSLs[14], Paul Hudak tells us that it can be fairly difficult to design and implement a programming language from scratch, and not only that, but that it is not uncommon for such undertakings to span anywhere between 2 to 5 years [14]—which perhaps many can't afford.

However, away from what those researchers say concerning this challenge, we also find that, upon closer inspection of what actually goes on in real-world problem-solving using programming languages—such as the experience the author has acquired while operating and leading a research lab exploring and implementing several small and large academic and industry projects relating to or driven by computing[66]—several of which involved some sort of language engineering—at **Nuchwezi**[3][67], the general failure to see many DSLs come up and/or mature past concept phase, is because, if for no other reason at least, merely by virtue of there existing many capable GPLs with which one might approach the solution of a problem for which a DSL was first envisioned, might steer one away from actually diving in and implementing a new language, instead choosing to adapt or adopt some existing GPL or a DSL, so as to manifest the solution (without innovation). And also, even

[3] see https://nuchwezi.com

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

where one might go ahead and attempt to implement a DSL, such attempts might only need go as far as the implementation of the DSL in the form of a library or some API over another DSL or GPL and nothing further than that—such DSLs being technically referred to as "embedded languages" to differentiate them from *external languages*[7] or we might refer to them as *Domain Specific Embedded Languages* (DSELs)[9].

Further, we should note that such an alternative to actually implementing an Independent DSL (IDSL), might make sense especially if merely implementing a Dependent DSL (DDSL) offers the solution to the original problem, or allows for the most economical solution given real-life constraints on time, resources and the freedom to conduct fundamental scientific research necessary to really manifest a robust and wholly independent computer programming language such as a GPL or an IDSL.

Finally, it is interesting to note that typically, the value or worth of developing a new DSL might not be clear or obvious, until substantial investment in its development (using a GPL) has been made [9]. Mernik's paper further tells us that in such cases, the development of the DSL becomes a key aspect, or plays a fundamental role in the evolution or re-engineering of (existing) software [9]. Overall, it is worth noting that just like general software, computer programming languages do (and need) to evolve, in someways, just like human natural languages might. However, for these engineered languages, this evolution might for example be driven by progress [or lack thereof] in other [existing] programming languages—such as when a new/newer language offers shorthands for expressing common idioms that were originally more verbose in some other language[4].

3.1.2. The DSL Implementation Method

Concerning the actual implementation of a DSL, it should be noted that the language research and engineering community does offer some useful hints as to the general underlying principles and theory in several works [7,9,14], and we shall here attempt to distill the most important general aspects of these methods.

First, building upon what we have already seen in **Section 2.2**, the systematic development, or rather, evolution of a new DSL will typically consist of the following general steps:

1. Defining the domain
2. Specifying the Requirements of the DSL
3. Designing the DSL
4. Constructing the DSL
5. Supporting the DSL—basically, constructing tools to support the DSL
6. Applying the DSL—which is about constructing applications or rather solutions in the domain, leveraging the DSL and the DSL's support tools—the DSL infrastructure, platform or ecosystem—such as the so-called Software Operating Environment (SOE) for the language[68].
7. Evaluating the DSL

Defining the Domain: First, we note that before a DSL can be implemented, a domain needs to be defined for it—it is actually helpful to do so, as we shall soon see. This process of defining the domain isn't just haphazard, nor is it merely a matter of imagination, but is actually better driven by some kind of systematic analysis. "Domain Analysis", which is the name Mernik gives to this process[9], entails conducting some sort of *Knowledge Engineering*—a field he notes to be relatively new and largely unexplored, during which process, focus is given to the systematic capturing of knowledge about or in the domain of interest, then to its systematic representation, and finally to the development of some ontology representative of that domain.

DSL Specification: Once we have the domain defined, next we must specify the DSL we intend to implement for it. This makes sense, since, the implementation of a DSL—for programming in particular, likewise relates to the implementation of software, and as per rigors of software engineering in general, starting with a specification is better than not[13]. For language engineering in relation to a DSL though, we also get the recommendation to utilize the result of the previous step, in the form of a “Feature Model" for the domain[9], from which one or more DSLs can then be developed. Such a feature model for example might be captured or represented in the form of a concept-map or mind-map diagram, but in Mernik’s paper[9] we see this depicted using a feature diagram instead.

DSL Design: Once a specification for the DSL is in place, then we can approach the design. This process of designing a DSL can be classified using the following DSL Design Patterns;

1. **Language Exploitation**—in which the DSL is implemented (partially or wholly) using an existing GPL or another DSL. This pattern is further broken down into; *Piggyback*: in which an existing language is only partially used, *Specialization*: in which an existing language is merely restricted or constrained, and finally, *Extension*: in which an existing language is merely extended.
2. **Language Invention**—this involves the design of the DSL entirely from scratch, with no commonality between the new DSL and any existing languages.
3. **Informal Design**—which refers to cases where the DSL to be implemented is only described or specified informally, such as with natural languages or domain terms, but with no strict or formal/regular structures or properties being explicitly defined.
4. **Formal Design**—in which case, the new DSL is explicitly, and wholly, rigorously specified, typically using an existing syntax and semantics definition method such as attribute grammars, re-write rules or an abstract state machine.

In all cases, one wants to ensure that they give special treatment to the design phase of a DSL, because this greatly enhances the quality and effectiveness of the DSL implementation, and just like with the design of GPLs, such an approach could also give special attention to known GPL-design criteria such as readability, simplicity and orthogonality among others[9][14]. Oliveira’s paper also delves into some technologies that one might use in the DSL design proces, and these include use of Backus-Naur Form (BNF) or its extended variant, EBNF [7].

Constructing the DSL: The actual implementation of the DSL then follows, and for the case of executable DSLs[9], we note that, typically, their implementation, like the implementation of software in general, likewise leverages or utilizes existing software tools. For the case of DSLs, these might include use of generic, re-usable code-generators such as traditional Lex—for building lexers/code-syntax-readers/verifiers and Yacc—for the construction of code parsers[14]. Interestingly, both Lex and Yacc are themselves kinds of DSLs! However, typically, or more commonly, most DSLs are implemented using a kind of interpreter rather than a compiler, and so, might not need use the same language-engineering tools as GPL implementation would call for—thus, for a DSL, one might fore-go the need for leveraging an existing lexer generator or parser generator such as Lex and Yacc respectively. However, for DSLs in particular, we see several tools called out for their implementation, based on the DSL implementation strategy, and these we can summarize as such [7]: *Translation Grammar Tools*—such as JavaCC or SableCC; *Attribute Grammar Tools*—such as LISA, ANTLR and JastAdd; and for the case of DSLs via compilers or specialized intepreters—DRACO, ASF+SDF, Kephera, Kodiyak and InfoWiz—with the DSLs constructed using these last methods being considered “external languages" because they don’t depend on any pre-existing language directly. Finally, it should be noted that any DSL can be implemented using a suitable GPL, in which case the GPL thus used is referred to as the “base language" for the DSL[7], and among popular base languages for the construction of DSLs are Ruby, Python, Haskell, Java, C++ and Boo among others. For the case of leveraging a GPL to construct a DSL, two methods are called out—*Embedding*: in which case the DSL is created without having to first create a new/custom compiler or interpreter, since that of the base-language is used, and then *Extension*: in which case the DSL is created by taking an existing GPL’s compiler or interpreter and merely extending or adapting it to process the DSL [7].

DSL support tools: After a DSL is implemented, it definitely is supposed to be used. This for example means, someone should be able to write new programs leveraging the DSL’s syntax, and then be able to have this written code somehow translated into an executable or be processed somehow to solve a practical problem. In this regard, merely having the DSL’s compiler or interpreter is sometimes not enough, and so, helper tools such as an Integrated Development Environment (IDE)

 doi:10.20944/preprints202410.0636.v2

for the DSL[29], similar to the case for traditional programming, Specialized Editors, (syntax-aware) Pretty Printers, Consistency Checkers, (code) Analyzers and (code) Visualizers are also important[9].

Applying the DSL: Even for non-exectutable DSLs, the only natural and straigh-forward way to evaluate and evolve the language is via its practical application. Since we are talking about languages for solving computational problems especially, the meaningful way to apply them is to use them to write programs—computer programs to be precise. However, the special case of non-executable DSLs has been treated of by Mernik's paper[9], and we see emphasis being placed on the nature of their programs, which, unlike the programs of executable DSLs, are classified under the categories of "Specifications", "Definitions" or "Descriptions", to differentiate them from typical executable computer programs such as are the result of GPL and executable DSL programming.

Evaluating the DSL: Through applying the DSL, it then becomes easy and straight-forward to evaluate the language based on some quantitative or qualitative criteria. One might for example start to look at things like the language's runtime performance (measured in speed or space/memory consumption for example), its usability, generality, completeness, consistency, expressiveness, abstractness, concreteness and finally its computational power [7,29]. Mernik though, stresses that quantitative evaluation or validation of a DSL in both general and particular cases is hard and an open problem[9].

Note that by the above introduced classifications, the new TEA TPL[27] is a formally designed executable DSL whose current reference implementation[32] makes it a DDSL or rather a DSEL exploiting the **Python 3**[69] GPL as its base language via the specialization and extension patterns.

Figure 7. A Formal Description of the TEA TPL DSL

Finally, before leaving the matter of implementing DSLs, let us take a brief moment to appreciate the intricacies behind manifesting an effective and practical DSL, by considering the case of the TMIL language[8].

3.1.3. Lessons About DSL Implementation from the TMIL Project

TMIL is the "Text Manipulation Imaging Language", and was first formalized/introduced in a 2007 paper by Hamburger et al [8]. This language is a high-level DSL meant for the manipulation of text on an image as well as the drawing of text onto images. The language is meant to simplify what one could have done using GPL capabilities in Java or C++, concerning graphics programming, however, it simplifies such tasks by developing a more specialized, simpler programming interface for the task while still retaining much of the syntactical characteristics of those GPLs.

The TMIL language is syntactically similar to Java and C++, and like them, supports such common GPL characteristics such as support for special lexical conventions; for example, having strict rules for the naming of identifiers, support for single-line and multi-line comments, reserved words, special characters, support for constants and in-built operators among others[8]. Like most GPLs, TMIL supports lexical and semantic scoping, and its scoping style is very similar to that found in C or C++. TMIL was designed to be cross-platform, and has been implemented for Linux, Windows and MacOS for example.

In terms of how TMIL works under-the-hood, it is worth noting that TMIL was mostly implemented using ANTLR (Another Tool for Language Recognition) [70], and that basically, TMIL source-code compiles to C++, so that TMIL code should compile and run anywhere C++ code would [8]. Specifically, ANTLR was used to help build the compiler components for the TMIL compiler (TMIL Lexer, TMIL Parser and TMIL tree-walker) thus;

- TMIL Lexer: breaks TMIL source-code down into a series of tokens for the TMIL Parser.
- TMIL Parser: checks those TMIL tokens to ensure the TMIL syntax is obeyed and correct, then generates a TMIL AST (Abstract Syntax Tree) based on these, for the TMIL source-code that was provided.

 doi:10.20944/preprints202410.0636.v2

- TMIL Tree-Walker: operates on the generated TMIL AST, checks for semantic errors, then generates target C++ code.

In summary, we see that the process for processing TMIL source-code is:
Valid TMIL Code → TMIL Compiler → C++ Code Generated → C++ Compiler → Target Platform Executable

3.2. What to Consider When Implementing a TPL

In **Section** 2.2 we have introduced much of the essential theory in designing a DSL, and have also given special treatment to the design of TPLs in **Section** 2.3. The laws and general principles laid out in **Section** 2.1 shouldn't be taken for granted either, and shall help guide any implementation of any kind of TPL for that matter. It shall be found that much of what one needs to consider before actually implementing a TPL follows directly from what we have laid out in those earlier sections. Specifically concerning implementation, **Section** 3.1.2 covers most of the ground we need for TPL implementation, and much need not be repeated here. However, it should be noted that for the case of TPLs, at least based on the author's own experience while implementing the TPL TEA[32], the following observations could simplify a TPL implementor's life further:

1. Spend more time working on, studying and understanding the TPL's design and specification before actually implementing the TPL. Essentially, avoid directly jumping into the coding. Building extensive documentation about the specification and design of the language ahead of its implementation shall help streamline much of the actual implementation/coding phases to follow. Definitely, as with any software, it is wise to not fall into the trap of over-engineering, however, as language engineering isn't just any kind of software engineering, clear, and careful attention to a *Clean-Room Process* shall greatly contribute to the robustness, correctness and efficiency of the language implementation.
2. Ensure to give some considerable time to trying out the TPL conceptually—say, using pseudo-code or thought-experiments, before actually waiting to implement the language and then test or try it out. This, especially with some realistic problems in the domain the language is expected to solve, so as to identify and fix any conceptual, semantic or syntactic flaws with the planned language or its design before much effort is poured into its actual implementation. This phase can also readily help catch critical omissions in the language design, as well as eliminate unnecessary redundancies early-on. This then gives us a clean and robust language specification and design.
3. Once the TPL is well designed—and this need not be done all at once or in one-sitting, then look around for any existing languages—especially those related to the planned language—by domain, grammar or syntax, and spend some time studying them or gleaning useful cues about how they were designed and implemented, and what makes them successful. Then adopt some of this knowledge for the implementation of your own TPL.
4. Decide on whether the TPL is to be an interpreted or compiled language, as this will greatly determine how to proceed from its design to the implementation. Initial focus should be on realizing or manifesting a proof-of-concept, a prototype of the TPL and nothing more. For example, if the TPL is to be interpreted, then, merely deciding on which target environment, platform or operating system to build a proof-of-concept for, shall greatly narrow down much of the intimidating aspects of the language implementation—for example, it shall then be clear, what choice of technologies to leverage to implement the language, because, not every available language development tool might lend itself readily for any potential target operating environment.
5. Start coding. Construct. Implement the damn language, and all the while, occasionally return to and consult the specification and design documents for the language, and if necessary, either evolve them or evolve the language implementation itself.
6. Test! Don't wait to fully implement the language before beginning to test it. Also, where the language development tools allow you to—important to consider this early on, ensure to have

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

enough insights into what is actually going on inside your language's run-time—the interpreter or compiler you are building should help with this, so that it is simple to catch implementation problems and tell where they originate from—let's call it *language-debugging*—for example, a run-time test might fail, not because the language design or semantics are wrong, but because the implementation has a flaw. But also, it could be that the test itself is the problem and that time shouldn't be wasted trying to fix the language implementation or design without knowing clearly, unambiguously, what each written test should produce as its result or output with or without the actual language implementation! So, do lots of things in the head! It also helps.
7. Iterate!

3.3. *Leveraging Design Research in Language Engineering*

When it comes to how to actually go about exploring, albeit practically, and systematically, matters concerning the implementation of a new programming language—GPL, DSL or TPL doesn't matter, there is not much in terms of a one-size-fits-all methodology to be precise. However, basing on the field and laboratory experience and success of the author while working on three projects touching on programming language engineering since 2019; first, with DNAP[68]—in which, an approach was sought to allow non-programmers to be able to design, publish and apply web and mobile apps leveraging a Low-or-No-Code (LNC) paradigm, via a kind of visual programming in the Persona IDE that under-the-hood defines "mini-programs" leveraging a dialect based on JSON Schema, and which mini-language[4] was named "Cwa Script"; then while working on the Voice Operated Support Assistant (VOSA) platform[71]—a kind of generic, re-usable and re-configurable voice-controlled artificial personal assistant that allows users to command/control it using voice-commands or rather a not-so-well-defined, but practical voice-based/speech-to-text command-and-control language with an interesting natural-language-based instruction set; then while working on TEA, the **T**ransforming **E**xecutable **A**lphabet, the new programming language meant to standardize text-processing as a first-class paradigm in solving arbitrary problems programmatically[27]; in each of these language engineering projects, it has been established that the Design Science Research (DSR) paradigm, or to be more general, the Design Research paradigm[72] also brought up in Lutalo's VOSA thesis technical report[5] [73] serves well to systematize research on the design, implementation and evaluation of a new programming language.

DSR has the advantage that it doesn't really matter what kind of computer system is under consideration, as long as the research project involves some sort of need to systematize a kind of innovative endeavor involving conceptualizing, designing and implementing some kind of information processing system, or rather a software system[74]—for which, considering our interests here, all, if not most practical computer programming languages are. The paradigm supports well, practical research of an experimental kind in ICT and computer science generally, and is to be considered very plausible for guiding research into the implementation of a TPL for that matter.

3.3.1. Example Results from TEA DSR Work

Because the DSR paradigm calls for the production of useful artefacts resulting from the undertaken research—for example, applied to a language engineering project, this might be the tools, example source-code, new knowledge, documentation, literature and other kinds of technical and creative outputs resulting from the research work. For the case of the TEA project as an example of how successful this paradigm can be, we can call out a few interesting results that are noteworthy:

Thus far, some compelling results working on the TEA language has given us include:

4 Concept of a "mini-language" might be likened to Mernik's concept of *little languages*[9] which are essentially small DSLs that do not include many features found in GPLs

5 Not to be confused with the later, shorter and related VOSA paper[71], the VOSA technical report is accesible via https://doi.org/10.6084/m9.figshare.25138541

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

1. A useful programming language specification and design example depicted in the TEA "TAZ" manuscript[27]
2. A fully functional reference implementation of a *pure TPL* in the form of a working and ready to install TEA language Unix/Linux package named **tttt**—currently at version 1.0.5[6][32]
3. Useful documentation concerning getting started with how to program in the TEA language—comes with the above mentioned **tttt** package, as well as more in the language's official *living manual*[27]
4. New knowledge and ideas such as we've come across in this very paper.
5. A collection of example source-code and TEA programs included in the project's official repository[32]

Concerning the last item on that list of results, we shall briefly look at some interesting example programs of applying the implemented TEA language to some basic, but important problems applying text processing.

The first is an example for how a TEA program can be used to transform raw text into more presentable formats such as by presenting it drawn inside of a neat textbox, and this, on the command-line/inside a typical Linux terminal, without using any special graphics processing but just mere text manipulation. We see the example source-code for this case in **Listing 1**, and an example of this TEA program being applied in **Figure 8**.

Listing 1. Basic Text Processing with Graphics in TEA

```
# Given some text, shall return a text box drawn around it
f!:^$:lPROCESS # don't prompt if there's already some input
i!:Enter some text: |i:
l:lPROCESS
v:vIN
r*!:vIN:.:-
x:--|x!:--
v:vBTOP
v:vSTART:
v:vBLR:{|}
g*:{ }:vBLR:vIN:vBLR
v:vIN
g*:{_ }:vSTART:vBTOP:vIN:vBTOP
h!:_
r!:_:|
```

[6] see https://github.com/mcnemesis/cli_tttt

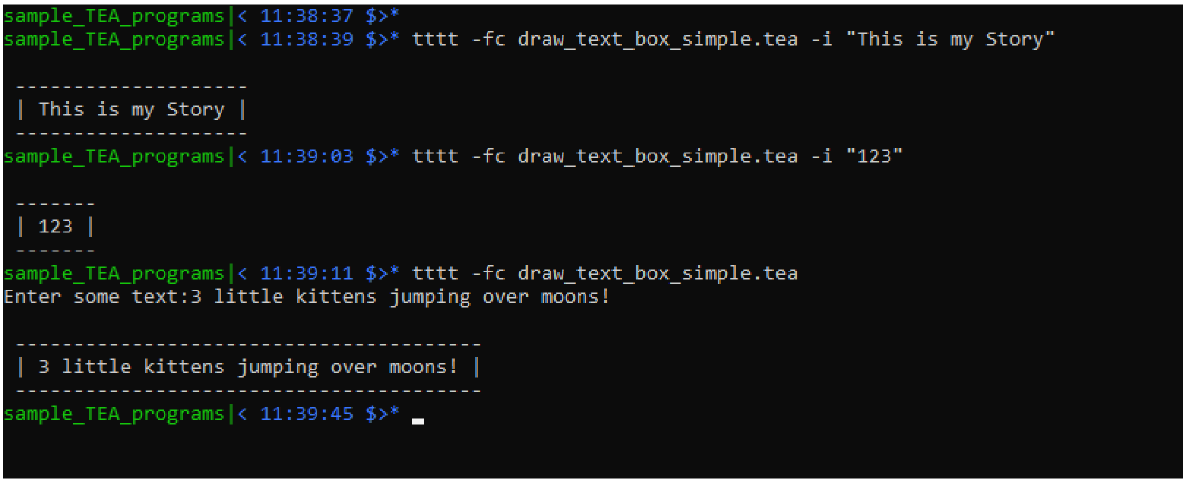

Figure 8. Drawing Text-Boxes on the Linux Commandline using TEA

The second example merely highlights an interactive “Hello World" program in TEA—which, instead of merely displaying “Hello World", first prompts for a name from the user, then uses it to greet them. We see the example source-code for this case in **Listing** 2, and an example of this TEA program being applied in **Figure** 9.

Listing 2. An Interactive Hello World program in TEA

```
i:{What is your name please? }|i:|x:{Hello }
```

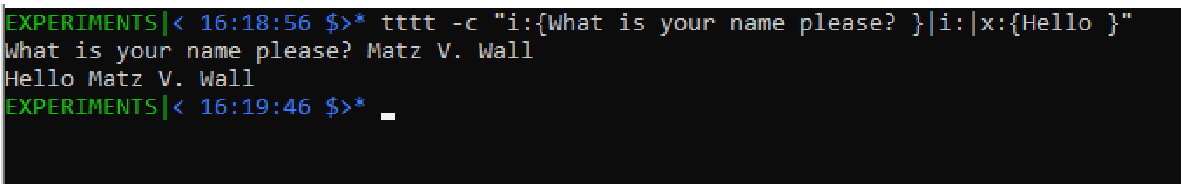

Figure 9. Running the Interactive TEA Hello World

The third example demonstrates how, starting with a basic Random Number Generator (RNG) implementation in TEA—lines #1-6 in **Listing** 3, a simple, but compelling dynamic art generator (in the form of ASCII art) is realized, and which can then be used to create interesting artworks that might inspire more complex work or which can directly be used as is. The source for this program, taken from the official collection of examples in the command-line TEA implementation—TTTT, the “TEA Text Transformer Terminal"[32] is shown in **Listing** 3, and two examples of the kind of artworks this simple TEA program can generate are shown in **Figure** 10 and **Figure** 11.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

Listing 3. TEA program Implementing an ART Generator

```
i:123456789 0
a!:
r:[2357]:0 987654321
a!:
d:[ ].*$
d:^0+
r!:[0]:* * * * * * * * * *
r!:[19]:=========
r!:[28]:<><><><><><><><>
r!:[37]:      ======
#r!:[46]:<><>*<><>
r!:[4]:<><>*<><>
r!:5:#*
r!:6:+>
t!:
a!:
```

Figure 10. Example Random Art using TEA

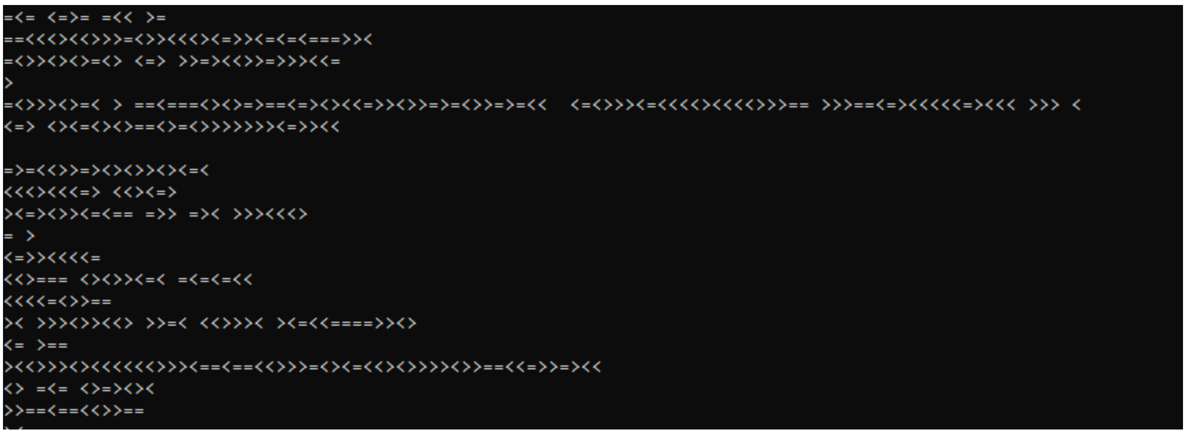

Figure 11. Architectural ASCII Art from the modified "rCHURCHY City SKYLINE Example"

Finally, we also look at one example that demonstrates how systematic statistical analysis might be attempted via TEA; basically, with the "WordGraph.tea" TEA program shown in Listing **Listing 4**. Two example invocations of this program via the command-line are shown in **Figure 12**—one visualizing mere numbers via direct user-input, the other an alphanumeric command-line parameter. This basic example, simplified to merely map numbers and/or words to their visual projections based

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

on relative positioning in the Base-36 Symbol Set [38], can help one appreciate how complex scientific problems might be solved, and how such solutions could be designed using a text-processing language such as TEA.

Listing 4. Interactive Word-to-Graph TEA program

```
#!/usr/bin/tttt -fc
#----------------+
# WordGraph.tea
#----------------+
# The following program
# when given a number, word
# or text, shall print
# its visual projection.
f!:^$:lPREPROCESS
i:{Enter a value: }|i:
l:lPREPROCESS
v:vTEXT
r!:[ ]*:{ }|#reduce sparseness
g!:_|#eliminate punctuation
h!:|#place each char on a line
z:|#lowercase everything
l:lMAP
r!:[0_]: | r!:1:= | r!:2:== | r!:3:=== | r!:4:==== | r!:5:=====
r!:6:====== | r!:7:======= | r!:8:======== | r!:9:=========
r!:a:----------+ | r!:b:-----------+ | r!:c:------------+
r!:d:-------------+ | r!:e:--------------+ | r!:f:---------------+
r!:g:----------------+ | r!:h:-----------------+ | r!:i:------------------+
r!:j:-------------------+ | r!:k:--------------------+ | r!:l:---------------------+
r!:m:----------------------+|r!:n:-----------------------+
r!:o:------------------------+|r!:p:-------------------------+
r!:q:--------------------------+|r!:r:---------------------------+
r!:s:----------------------------+|r!:t:-----------------------------+
r!:u:------------------------------+|r!:v:-------------------------------+
r!:w:--------------------------------+|r!:x:---------------------------------+
r!:y:----------------------------------+|r!:z:-----------------------------------+
# prints the text as a graph
```

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

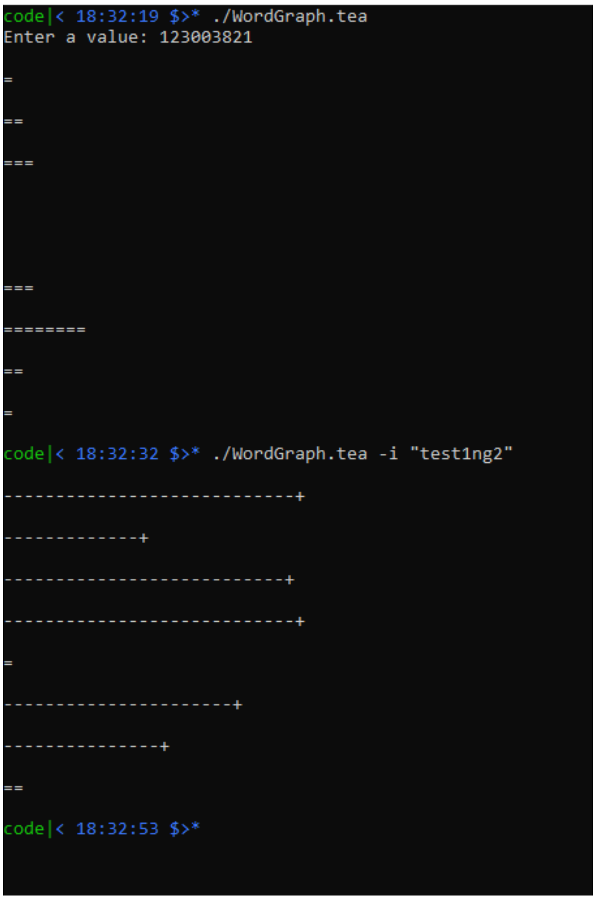

Figure 12. Word and Number Graphs drawn using TEA

4. Systematic Evaluation of Programming Languages

In **Section** 3.1.2 we have briefly looked at some concepts useful in the systematic evaluation of a TPL. That basis could serve well to bring into context the matter of how to systematically evaluate any programming language, and we are to build upon it in this section. Essentially, we wish to arrive at some guiding framework for the proper, perhaps exhaustive evaluation of any programming language, and this then can readily be used to evaluate any language category—GPL, DSL or TPL.

For starters, consider the attempt presented in **Figure** 13, which we are going to refer to as the Programming Language Evaluation (PLEf[7]) framework. It is an attempt at creating an ontology with which both qualitative and quantitative evaluation of any programming language can be approached. It also helps to put the already mentioned SOE evaluation framework into a general context for language evaluation.

---

[7] The special acronym **PLEf** chosen so as to differentiate it from **PLEF**—the Programming Language *Evolution* Framework first defined in **Figure** 6

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 December 2024 doi:10.20944/preprints202410.0636.v2

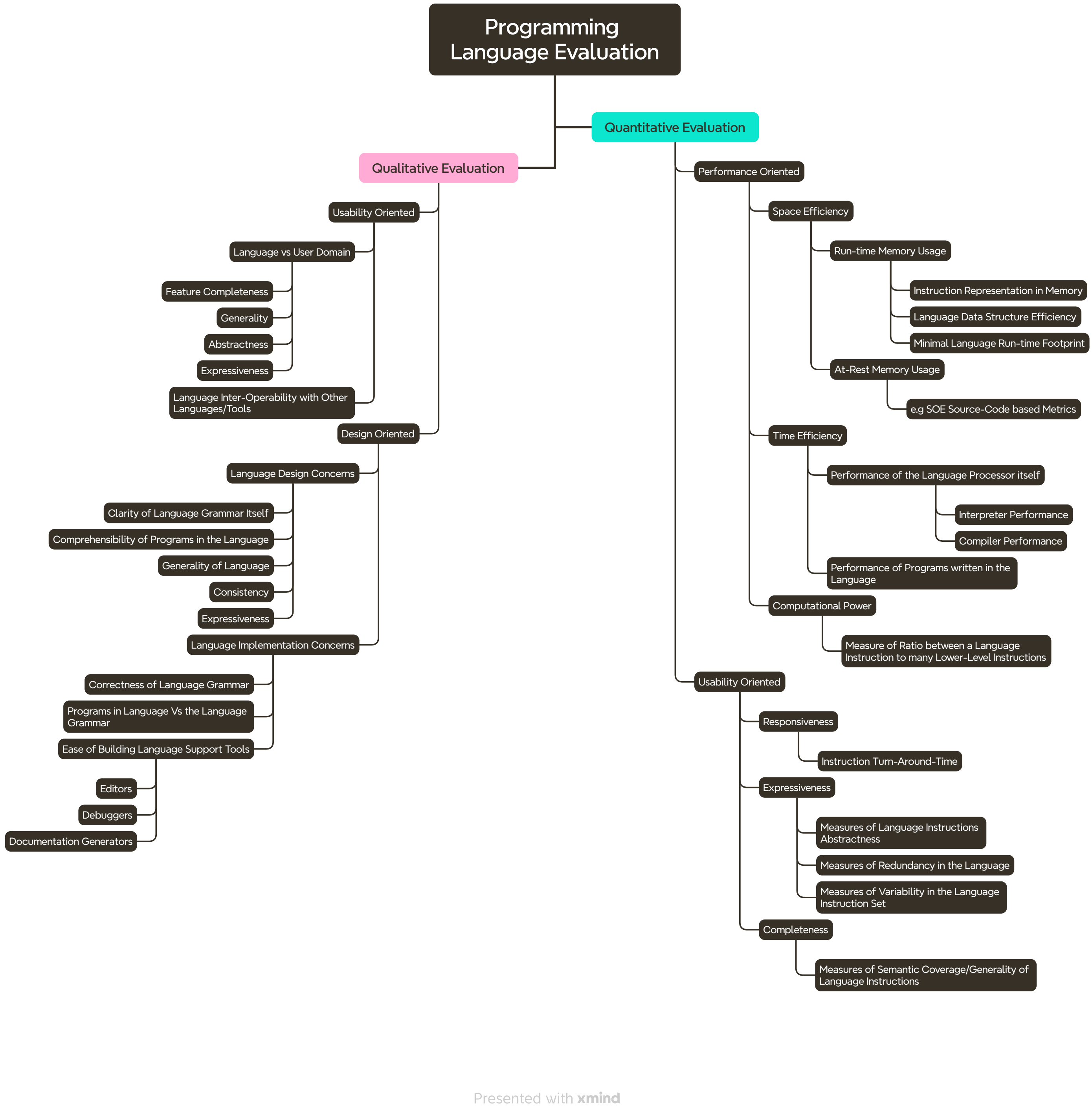

Figure 13. The PLEf framework as a Concept Map

Much could be said about language evaluation methods using this framework alone, however, for the sake of keeping this present paper brief, we suggest to treat of the full PLEf in a future work. The following three sections though, touching on language evaluation ideas already existing in the literature, shall help sum-up our present discussion concerning language evaluation.

4.1. DSLs vs GPLs, Humanness Vs Abstractness of Computer Languages

For a moment, we return to the ideas introduced in **Section 1.2**, and which are illustrated in **Figure 1**. First, notice that, because the typical digital computer basically speaks or understands only binary (1s & 0s), humans generally employ abstractions to express instructions to such a computer in a human-friendly language, but which can then be readily translated to binary for the computer to understand and execute [7]. On the spectrum of abstraction, DSLs are typically more abstract than GPLs for this basic reason.

Also, GPLs are tailored to be used to solve any kind of problem, irrespective of domain [7]—thus them being "general-purpose", while a good DSL operates at a high abstraction level that is closer to the human's way of thinking about or solving problems in a particular and/or target domain; essentially, by increasing the distance between a human's problem domain and a machine's way of solving problems [in any domain].

We also know that DSLs offer substantial gains in expressiveness and ease of use compared to GPLs in their domain[9]. Further, we know that programs written in a DSL are generally more concise, allow for faster development or iteration, are easier to maintain, easier to reason about, and can typically be written well by non-programmers (who typically are experts in the domain for which the DSL was designed than in general programming)[14].

However, it should be noted that where run-time performance or efficiency is a problem or a critical matter, and a low-level language (such as many GPLs are), isn't able to solve it, then, creating a DSL to improve the solution might not be the best route to solving the performance problem [14]. Thus, we can note that where performance is key or critical, prefer a low-level general-purpose language to a higher-level language—especially where several layers already exist between the underlying machine language (which in reality is the *actual* problem solver) and the programming language of choice.

4.2. The SOE Framework & Evaluating Any Set of Programming Languages by Their Syntax Properties

Because a programming language is meant to be written so as to be applied or executed (refer to UPLT in **Section 2.1**, especially the First Law of UPLT) it should make sense to consider programming language evaluation both from the perspective of writing programs in it, and then exeucting programs with it—the former deals with applying the language "at rest" (relates to the "At-Rest Memory Usage" concept presented in the PLEf framework in **Figure 13**), while the later deals with the language while in active use such as during the running of a real program written in the language, while it is being processed by the language's runtime.

The idea of evaluating a language "at rest" mostly deals with the characteristics of the language's structure and style/syntax while being applied to express some solution in the form of a computer program for example. It deals with the expressions of the language itself, and thus is a factor of the language's syntax and grammar to be precise. Nothing about the actual language's implementation or processing at this level or in this context, because such evaluations have more to do with the actual run-time environment or platform upon which the language's programs are executed—makes sense, because, even for the same language standard, for example, for the HTML[75] or CSS standard[76], the same exact source-code when looked at from the context of a run-time, such as when such web source-code is rendered in different web browsers—some of which might not fully or accurately implement the standard, it might be found that differences exist in how fast a page is fully rendered—something that has little to do with the syntax or properties of the code itself, and more to do with the rendering or code-processing engine. This should help clarify the importance of this evaluation approach.

The SOE framework [68] is helpful in this context because it only focuses on metrics that measure or compare programming languages at source/syntax level. For example, it looks at how basic/fundamental programs such as the popular "Hello World" program would be expressed across the set of languages under consideration, thus helps to bring out such subtleties as unnecessary verbosity and/or overhead in expressing basic programming idioms such as merely printing to standard output the string "Hello World".

The SOE also looks at other interesting aspects such as the minimum amount of code required to prompt for and then output a string in a given programming language—something not to be taken lightly, because much of useful programming is underpinned by this basic functionality.

4.3. The Case of Evaluating TPLs

The evaluation of TPLs should of course base upon the same principles and ideas of language evaluation as have already been introduced and discussed in the earlier sections of this paper (the PLEf framework in **Figure 13** should come in handy), however, especially because TPLs are a kind of DSL focused on text processing, special metrics and evaluation methods focused on just text processing capabilities and approaches might help here.

From a program development context—meaning, expressing a particular solution leveraging a particular language, we might want to consider how much a TPL simplifies the most typical text processing tasks—in the same vein as how the SOE treats of a GPL's evaluation via the analysis of general program categories such as the Most Basic Output Program (MBOP), Minimum Basic Input Program (MBIP), Hello World program (HW), and more[68]. For a TPL then, we might similarly want to think of such generic text processing program cases as:

- **Most Basic String Concatenation Program (MBSCP)—**a case in which, for example, two small strings such as "Hello" and "World" are combined into one larger string, to form "HelloWorld".
- **Most Basic String Filtering Program (MBSFP)—**in which, for example, one string, such as "Hello World" has every instance of another string such as "o"—typically referred to as the *search pattern*, used to eliminate contents from the first string—producing "HellWrld" in this example.
- **Most Basic String Quantification Program (MBSQP)—**for example, given "Hello World", to compute and return its length, 11.

Especially by measuring the LOC metric—one of *Lines of Code*(meaningful for evaluating multi-line programs) or *Length of Code*(better for short, single-line or one-liner programs such as can be easily expressed on the command-line) for each of the above cases of special TPL programs, we might then readily arrive at a meaningful, non-ambiguous and very telling quantitative comparison of two or more TPLs. Definitely, and rather interestingly, these same metrics could lend themselves readily to the evaluation and comparison of any programming language, GPLs especially, merely by focusing on evaluation of their text processing capabilities, and thus, should not be dismissed when considering arbitrary language evaluation (should make sense, given the implications of UPLT).

Also, this fits well into the overall language evaluation ontology already introduced in the PLEf framework. The DNAP paper [68] should help with how to go about building a concrete language evaluation case for any TPLs leveraging the above suggested general text processing scenarios. For example, building on this idea, we can attempt to compare TPLs such as TEA, Sed, Awk and Python on the MBSQP case, and several such evaluations would help form an example of Text Processing Language Evaluation Framework (another TPLEF!) applications, stuff we aren't going to delve into here, but which we shall want to give detailed treatment of in a future work.

5. Epilogue

Having come to the conclusion that all programming is text processing—see **Section 2.1**, it also implies that all programming languages, GPL, DSL or not, are basically TPLs—refer to **Theorem 2**. For example, for a GPL, it might be argued that their ability to process numbers, boolean logic or boolean expressions, simple and complex data structures, objects and arbitrary types, is all merely a result of processing mere text with but higher-than-mere-string abstractions imposed on it. These ideas are well illustrated in the PLEF—refer to **Figure 6**, which is underpinned by the fundamental research we have conducted while reviewing the literature on language engineering as well as ground-breaking work on the TEA computer programming language. The most important results, summed up in three laws which together form the basis of a unified theory we have called the "Unifying Programming Language Theory", **UPLT**, is expected to guide much of all future work in this field of language engineering.

Further, these observations and results should further deepen the argument that research into text processing and TPLs in general, lies at the core of all useful research and development into problem solving using computation and essentially computer programming languages.

This paper has brought together many important ideas concerning the design, implementation and evaluation of programming languages, with special focus on DSLs and TPLs. Also, we have looked at the important characteristics that distinguish any programming language from another, based on its abstraction level and proximity to the human problem domain it is meant to provide solutions for—refer to **Sections 1.2 and 4.1**. We have seen that, at the lowest level, machine languages operate closest to the physical and logical processors that operationalize the [physical] machines that make computation possible, while, on the highest-end, domain specific languages serve to abstract away most of the [low and higher-level] complexity and intricacies of a computing machine—abstract or not, and instead focus more on simplifying problem solving in a language and at a level closest to the human domain for which practical solutions are required. DSLs in particular, fill an important category of programming languages, and more importantly, we saw that TPLs, which might sometimes be realized using just a GPL, are a class of DSLs that warrant special treatment and attention given their relevance even in the case of realizing any useful GPL[8]. As most of the problem solving possible with computers starts with some sort of reading and writing of data—which, for all practical cases, is typically merely some kind of text, we argue that research into, and development of better TPLs can help support and drive better innovations across the entire landscape of computing and software engineering—thus the **PLEF**. We looked at the theory behind the design, implementation and evaluation of DSLs, but also of TPLs as a special case. The Transforming Executable Alphabet (**TEA**) programming language, which is still new, and which especially was designed by the author as a text processing language meant to aid in general problem solving, does warrant some special attention, and working on this language has helped arrive at many of the interesting results presented in this paper, including the useful ideas concerning the design of any TPL (see **Section 3.2**), and a framework for the systematic evaluation of any programming language—see **Section 4**. More work remains to be done in relation to many of these newly introduced ideas—especially concerning the frameworks that should guide much of future language engineering. However, with regards to TEA, which first inspired this work, future research could dive deeper into the distinction between TPLs and GPLs given TPLs like TEA have been found to readily solve some problems traditionally only best left for a *typical GPL*, but also, research into how to design and implement better TPLs or how to implement a GPL based on a *pure TPL* could make much sense for the language research community and industry at large.

Acknowledgements

I would like to thank my children, Karungi S. Marlyn, Theo R. Willrich and Arora J.J. Muntu, who, despite their tender age, have psychologically supported me so much, so I could stay focused and push-on with active research at Nuchwezi ICT Research Lab—from where much of this work has been conducted since 2014. I can't take for granted those many moments they would share with me a smile, call me "daddy" and not bother me concerning why I chose to turn their home into a research facility. I trust, and promise, they shall get back their home!

---

8 Note that there might seem to be some circularity here... a kind of Chicken-Egg problem concerning whether a GPL or a DSL brings to life a TPL—which itself might in some contexts be considered a GPL! This confusion is the motivation for developing the UPLT in **Section 2.1**, and should motivate further research into this fundamental problem

 doi:10.20944/preprints202410.0636.v2

About the Author

Joseph Willrich Lutalo C.M.R.W. is a Graduate Researcher (Final Year) at Makerere University. In 2011, he received his B.Sc. degree (Hons) in Computer Science with a minor in Mathematics from Makerere University, Kampala, Uganda. From 2014, he has operated and administered Nuchwezi, a multifaceted organization based in Uganda that is best known for its various technological and educational projects. He is currently its Internet President. He has written 3 Books, 8 Papers, 1 Talk and 2 Drafts as part of his major academic contributions thus far, and was awarded a Certificate of Excellency for presenting work on VOSA at the SE2024 conference in Copenhagen, Denmark. His research interests include business automation, intelligent mobile systems, programming languages and software platforms.

Data Availability Statements

The data that support the findings of this study are available from the corresponding author, Joseph W. Lutalo, upon reasonable request. Otherwise, the primary data and materials used in this research are openly available in the GitHub repository at https://github.com/mcnemesis/cli_tttt, as well as specifics about work on the TEA language in the **TAZ Manual** [27]. The repository contains much of the essential data related to the author's present research, including source code, documentation, and supplementary materials.

Code Availability

The source code—especially relating to the author's work on the TEA programming language [27,32], the earlier DNAP [68] data-engineering technical platform as well as the VOSA [73] artificial assistant platform is available from the corresponding author by request.

Conflict of Interest

The author declares that they have no conflict of interest.

References

1. Kain, R. Advanced computer architecture: a system design approach. Proceedings of the 1996 workshop on Computer architecture education, 1996, pp. 12,116–117.
2. Field, D. *The world's greatest architecture: Past and present*; Chartwell, 2007.
3. Nawangwe, B. The evolution of the Kibuga into Kampala's city centre-analysis of the transformation of an African city **2010**.
4. Gordon, C.S. The Linguistics of Programming **2024**.
5. Landauer, R. Computation: A fundamental physical view. *Physica Scripta* **1987**, *35*, 88.
6. Erwig, M.; Walkingshaw, E. Semantics first! Rethinking the language design process. International Conference on Software Language Engineering. Springer, 2011, pp. 243–262.
7. Oliveira, N.; Pereira, M.J.; Henriques, P.R.; Cruz, D. Domain specific languages: A theoretical survey. *INForum'09-Simpósio de Informática* **2009**.
8. Hamburger, E.; Merler, M.; Wei, J.; Yang, L. Text Manipulation Imaging Language **2007**.
9. Mernik, M. Domain-specific languages: A systematic mapping study. International Conference on Current Trends in Theory and Practice of Informatics. Springer, 2017, pp. 464–472.
10. Howard, G.D. Github copilot: Copyright, fair use, creativity, transformativity, and algorithms **2021**.

11. Stratton, J. An Introduction to Microsoft Copilot. In *Copilot for Microsoft 365: Harness the Power of Generative AI in the Microsoft Apps You Use Every Day*; Springer, 2024; pp. 19–35.
12. Idrisov, B.; Schlippe, T. Program Code Generation with Generative AIs. *Algorithms* **2024**, *17*, 62.
13. Sommerville, I., 1.2 The software process; Addison-Wesley, 1995; p. 7–7.
14. Hudak, P. Domain-specific languages. *Handbook of programming languages* **1997**, *3*, 21.
15. van Rossum, G.; Warsaw, B.; Coghlan, N. PEP 8 – Style Guide for Python Code, 2001. Accessed: 2024-09-25.
16. (W3C), W.W.W.C. CSS: Cascading Style Sheets, 2024. Accessed: September 21, 2024.
17. Team, G.D. DOT Language - Graphviz, 2024. Accessed: September 21, 2024.
18. Urban, M. *An introduction to LATEX*; TEX users group, 1986.
19. Lovrencic, A.; Konecki, M.; Orehovački, T. 1957-2007: 50 Years of Higher Order Programming Languages. *Journal of Information and Organizational Sciences* **2009**, *33*, 79–150.
20. Weyl, E. *Transitions and animations in CSS: adding motion with CSS*; "O'Reilly Media, Inc.", 2016.
21. *Dictionary of computing*, 4 ed.; Oxford University Press, 1996.
22. Wienold, G. Some basic aspects of text processing. *Poetics Today* **1981**, *2*, 97–109.
23. Mertz, D. *Text processing in Python*; Addison-Wesley Professional, 2003.
24. Levine, J. *Flex & Bison: Text Processing Tools*; "O'Reilly Media, Inc.", 2009.
25. Navarro, G.; Mariano, C. String Processing and Information Retrieval, 2005.
26. Tucker, A. *Text Processing: Algorithms, Languages, and Applications*; Bristol-Myers Squibb Cancer Symposia, Academic Press, 1979.
27. Lutalo, J.W. TEA TAZ -Transforming Executable Alphabet A: to Z: COMMAND SPACE SPECIFICATION **2024**. doi:10.6084/m9.figshare.26661328.v7.
28. Siino, M.; Tinnirello, I.; La Cascia, M. Is text preprocessing still worth the time? A comparative survey on the influence of popular preprocessing methods on Transformers and traditional classifiers. *Information Systems* **2024**, *121*, 102342. doi:https://doi.org/10.1016/j.is.2023.102342.
29. Yessenov, K.; Tulsiani, S.; Menon, A.; Miller, R.C.; Gulwani, S.; Lampson, B.; Kalai, A. A colorful approach to text processing by example. Proceedings of the 26th annual ACM symposium on User interface software and technology, 2013, pp. 495–504.
30. Miller, R.C.; Myers, B.A.; others. Lightweight Structured Text Processing. USENIX Annual Technical Conference, General Track, 1999, pp. 131–144.
31. Siever, E.; Weber, A.; Figgins, S.; Love, R.; Robbins, A. *Linux in a Nutshell*; "O'Reilly Media, Inc.", 2005.
32. mcnemesis. cli_tttt: Command Line Interface for TTTT, 2024. Accessed: 2024-09-21.
33. TutorialsPoint. C Standard Library - <string.h>, 2024. Accessed: September 21, 2024.
34. Foundation, P.S. Python Standard Library - string, 2024. Accessed: September 21, 2024.
35. Zelle, J.M. *Python Programming: An Introduction to Computer Science*, 2nd ed.; Franklin, Beedle & Associates Inc., 2010.
36. Corporation, O. Java Platform SE 8 - String Class, 2024. Accessed: September 21, 2024.
37. Baker, S. String::Util - String processing utility functions, 2024. Accessed: September 21, 2024.
38. Lutalo, J.W. Numbers from Arbitrary Text: Mapping Human Readable Text to Numbers in Base-36 **2024**.
39. Dan, J. *Kabbalah: A Very Short Introduction*; Oxford University Press, 2007.
40. Lutalo, J.W. Explorations in Probabilistic Metaphysics **2023**.
41. contributors, W. Branches of Science, 2024. Accessed: 2024-09-25.
42. contributors, W. Computational Science, 2024. Accessed: 2024-09-25.
43. Tiwary, B.K. Bioinformatics and Computational Biology: A Primer for Biologists. *SpringerLink* **2022**. Accessed: 2024-09-25.
44. MacBeath, J.R.; Harvey, S.S.; Oldroyd, N.J. Automated fluorescent DNA sequencing on the ABI PRISM 377. *DNA sequencing protocols* **2001**, pp. 119–152.
45. Kugonza, D.R.; Kiwuwa, G.; Mpairwe, D.; Jianlin, H.; Nabasirye, M.; Okeyo, A.; Hanotte, O. Accuracy of pastoralists' memory-based kinship assignment of Ankole cattle: a microsatellite DNA analysis. *Journal of Animal Breeding and Genetics* **2012**, *129*, 30–40.
46. of Science, S.I.; Technology. Course Material: SBB1609, 2024. Accessed: 2024-09-25.
47. (OMG), O.M.G. Unified Modeling Language (UML) Specification Version 2.5.1, 2024. Accessed: September 21, 2024.
48. Schmidt, E.; Lesk, M. Lex - A Lexical Analyzer Generator, 2024. Accessed: September 21, 2024.

 doi:10.20944/preprints202410.0636.v2

49. Johnson, S.C. Yacc - Yet Another Compiler Compiler, 2024. Accessed: September 21, 2024.
50. Stallman, R.M.; McGrath, R.; Smith, P.D. *GNU Make: A Program for Directing Recompilation*, 4.3 ed.; Free Software Foundation, 2021. Accessed: 2024-09-25.
51. Backus, J.; Naur, P.; others. *Report on the Algorithmic Language ALGOL 60*; Springer, 1960. Introduced the BNF notation.
52. Aho, A.V.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools*, 2nd ed.; Addison-Wesley, 1986.
53. ISO/IEC. SQL: Structured Query Language, 2024. Accessed: September 21, 2024.
54. Ben-Kiki, O.; Evans, C.; döt Net, I. YAML Ain't Markup Language (YAML) Version 1.2, 2024. Accessed: September 21, 2024.
55. Microsoft. F# Language Reference, 2024. Accessed: September 21, 2024.
56. Muschko, B. *Gradle in Action*; Manning Publications, 2014.
57. Champeau, C.; Koenig, D.; D'Arcy, H.; King, P.; Laforge, G.; Pragt, E.; Skeet, J. *Groovy in Action*; Manning Publications, 2015.
58. Akhin, M.; Belyaev, M.; others. Kotlin language specification: Kotlin/Core, 2020.
59. Sahoo, P.; Pujar, S.; Nalawade, G.; Gebhardt, R.; Mandel, L.; Buratti, L. Ansible Lightspeed: A Code Generation Service for IT Automation. *arXiv preprint arXiv:2402.17442* **2024**.
60. contributors, R. LearnPython: Discussion on Python Programming, 2024. Accessed: 2024-09-25.
61. Salton, G. Automatic text processing: The transformation, analysis, and retrieval of. *Reading: Addison-Wesley* **1989**, *169*.
62. Sundaram, S.; Narayanan, S. AN EMPIRICAL TEXT TRANFORMATION METHOD FOR SPONTANEOUS SPEECH SYNTHESIZERS.
63. Kathuria, A.; Gupta, A.; Singla, R. A review of tools and techniques for preprocessing of textual data. *Computational Methods and Data Engineering: Proceedings of ICMDE 2020, Volume 1* **2021**, pp. 407–422.
64. Robbins, A. *Sed and awk Pocket Reference: Text Processing with Regular Expressions*; "O'Reilly Media, Inc.", 2002.
65. Christiansen, T.; Wall, L.; Orwant, J.; others. *Programming Perl: Unmatched power for text processing and scripting*; "O'Reilly Media, Inc.", 2012.
66. Lutalo, J.W. Combined Latest Resume — JWL **2024**. doi:10.6084/m9.figshare.27074407.v1.
67. Nuchwezi. Nuchwezi: Exploring the Intersection of Technology and Society, 2024.
68. Lutalo, J.W.; Eyobu, O.S.; Kanagwa, B. DNAP: Dynamic Nuchwezi Architecture Platform-A New Software Extension and Construction Technology **2020**.
69. Team, P.C. *Python: A dynamic, open source programming language*. Python Software Foundation, 2019. Python version 3.x.
70. Parr, T. *The Definitive ANTLR 4 Reference*, 2nd ed.; The Pragmatic Bookshelf: Raleigh, North Carolina, 2013.
71. Lutalo, J.W.; Oyana, T. VOSA: A Reusable and Reconfigurable Voice Operated Support Assistant Chatbot Platform. *Available at SSRN 4810799* **2024**.
72. Johannesson, P.; Perjons, E. *An introduction to design science*; Vol. 10, Springer, 2014.
73. Lutalo, J.W. VOSA: A Reusable and Reconfigurable Voice Operated Support Assistant Chatbot Platform, 2024. doi:10.6084/m9.figshare.25138541.
74. Hevner, A.; Chatterjee, S.; Iivari, J. Twelve Theses on Design Science Research in Information Systems. *Design research in information systems: Theory and practice* **2010**, pp. 43–62.
75. HTML: HyperText Markup Language - The Official Standard, 2024. Accessed: 2024-09-25.
76. Cascading Style Sheets (CSS) - The Official Standard, 2024. Accessed: 2024-09-25.