Normal Forms for Perl-Compatible Regular Expressions

Oleg Efimov
St. Lawrence University
Canton, NY USA
oefim22@stlawu.edu

Tommy Tracy II
University of Virginia
Charlottesville, VA USA
tjt7a@virginia.edu

Whiting Zheng
University of Virginia
Charlottesville, VA USA
ryu9cx@virginia.edu

Kevin Skadron
University of Virginia
Charlottesville, VA USA
skadron@virginia.edu

Kevin Angstadt
St. Lawrence University
Canton, NY USA
kangstadt@stlawu.edu

May 15, 2026

Abstract

Regular Expressions (regexes) and Perl-Compatible Regular Expressions (PCRE), their more expressive counterparts, have wide applicability in software applications, but are often difficult to write and debug. To improve readability of PCRE and mitigate the risk of unintentional performance degradation, we propose a set of four normal forms (1NF–4NF) for PCRE. We formally define these normal forms and empirically evaluate their effects on readability and performance on 36 real-world patterns. We demonstrate that our normal forms reduce expression length by 14% on average, and improve runtime performance by 2.3–3.5x on average across two popular Python regex engines.

Keywords: Regular expression, Perl-compatible regular expression, Normal form

1 Introduction

Regular expressions (regexes) are used for parsing, matching, and filtering in almost all areas of modern software engineering, with applications including data scraping, bioinformatics, and log filtering [4, 19, 21, 31]. However, regexes can be

particularly difficult to write correctly, especially for novices [23,27]. Extensions to regex syntax and semantics, such as Perl-Compatible Regular Expressions (PCRE), are commonly supported by programming languages and used by developers. CPU-based processing algorithms for PCRE are often implemented using backtracking algorithms that have exponential worst-case runtimes (cf. polynomial for theoretical regex). Something as innocuous as grouping parts of an expression (e.g. `(.)*`) can result in additional computation and data storage overhead. We refer to this phenomenon as *unintentional performance degradation*, which can either be a well-intentioned mistake or a malicious attack (e.g., regex denial of service [6, 18]).

Other domains have benefited from normalization rules for improving clarity and performance, such as database systems [14, 24]. A notion of normal forms has also been used in the context of regular expressions; however, the focus has been on non-developer-facing optimizations, such as reducing the search space in theoretical regular expressions for synthesis tools [15], or non-semantics-preserving transformations to optimize the detection of REDoS vulnerabilities [16]. Given the prevalence of PCRE and the associated development challenges, there is a need for a set of rules and restrictions to normalize the construction of PCRE patterns that favor writing correct and efficient regexes by construction.

In this paper, we propose a set of four normal forms for PCRE to reduce possible human errors, performance issues, and provide a more uniform and readable format. Each normal form eliminates redundant components of the pattern or restricts syntax that can result in unintentional performance degradation. The first two normal forms focus on improving the correctness and clarity of the PCRE, while the third and fourth normal forms focus on improving the performance of the regex. Because of the complexity of PCRE syntax, there are often many ways to write a regex that matches the same language, but with different performance characteristics. While individual properties of each normal form are independent, the normal forms themselves are *hierarchical*: a PCRE that satisfies the requirements of a higher normal form also satisfies the requirements of all lower normal forms.

We evaluate these normal forms over a random sample of RegExLib indicative benchmarks, evaluating runtime performance on Python's built-in `re` module and the third-party `regex` library [1], two common regular expression engines supporting different features of the PCRE. Our empirical evaluation of 36 benchmarks and 180 distinct PCRE patterns demonstrates that our normal forms have an average speed-up of 2.3x for `re` and 3.5x for `regex` while also reducing the number of characters in the patterns by an average of 14%.

2 Background and Related Work

In this section, we provide a brief background on regular expressions and PCRE. We also discuss related and complementary efforts to improve the readability, performance, and security of regular expressions.

2.1 Theoretical and Perl-Compatible Regular Expressions

Regular expressions are a common formalism for describing regular languages that are recognized by finite automata. A regular expression, R, is described over a finite alphabet, Σ, and is constructed recursively using exact matches, concatenation (R_1R_2), alternation ($R_1|R_2$), and Kleene-$*$ (R^*). Regexes are often extended with additional syntax, such as R^+ (one or more matches of R), $R^?$ (zero or one match of R), and character classes ($[abc]$), which do not increase expressive power. The *language* of a regex, $\mathcal{L}(R)$, is the set of all strings that match the pattern described by R. We refer the reader to standard texts for a more thorough discussion of regexes and their properties [25].

Theoretical regexes have significant limitations in their expressive power, and thus are not sufficient for many practical applications. For example, they cannot express patterns that require counting (e.g., matching a string with an equal number of a's and b's), or patterns that require backreferences (e.g., matching a string that contains the same substring twice). Programming language support for regular expressions have thus *extended* the theoretical expression syntax with additional features to support a broader set of supported patterns.

One of the most common extensions to regex is Perl-Compatible Regular Expressions (PCRE), which is supported by many programming languages and libraries [11]. PCRE includes a number of additional syntactic and semantic features,[1] such as escape sequences for character classes (e.g., `\d` for digits), non-capturing groups (e.g., `(?:...)`), lookahead and lookbehind assertions (e.g., `(?=...)` and `(?<=...)`), and atomic groups (e.g., `(?>...)`).

2.2 Performance and Normalization of Regular Expressions

Runtime, bottlenecks, and degradation of regex engines have been extensively studied [2,7,29,30]. In the worst case, a carefully crafted input can stall a PCRE engine, creating a Regular Expression Denial of Service (REDoS) [6]. Recent efforts to mitigate REDoS vulnerabilities include hybrid engine selection [28] and automated *localize-and-fix* repair of vulnerable regexes [17]. Deterministic regex engines, such as Hyperscan [12] or RE2 [9], avoid certain performance issues by eliminating support for some PCRE features.

Several normal forms have been proposed to simplify *theoretical* regular expressions, including Brüggemann-Klein's *star normal form* [3], Gruber and Gulan's *strong star normal form* [10], and Kim et al.'s *concise normal form* [15]. Kahrs and Runciman explore several simplifying transformations for theoretical regexes aimed at reducing their size [13]. In Section 3.2, we extend some of these transformations to PCRE, and across Section 3 introduce additional transformations that are specific to PCRE syntax and semantics.

[1] PCRE are therefore no longer describing regular languages [22].

3 PCRE Normal Forms

In this section, we present a hierarchy of four normal forms for Perl-compatible regular expressions. These normal forms restrict the syntax of PCRE patterns to improve and regularize the readability of patterns while also reducing the risk of unanticipated performance degradation. For a regex to be in a given normal form, it must satisfy all the requirements of that normal form. We number these normal forms from one to four (i.e., 1NF, 2NF, 3NF, and 4NF), with "lower" normal forms being more focused on readability and "higher" normal forms being more focused on performance. Each subsequent normal form requires more complex changes to the regex to satisfy the normalization rules. For novice regex writers, applying the first two normal forms (1NF and 2NF) can be sufficient to improve readability and remove common mistakes and unnecessary complexity to patterns. As we demonstrate in our evaluation in Section 5, these normal forms significantly reduce pattern length and have the ability to improve the runtime performance of regex engines. The remaining normal forms (3NF and 4NF) reintroduce some syntactic complexity while further restricting regex syntax to improve runtime performance and mitigate the risk for unintentional performance degradation.

3.1 Worked Example

For the remainder of this section, we will introduce each normal form and its requirements. We will also apply these rules to a running, worked example. This example is simplified from a real regex pattern[2] submitted to RegExLib, a popular online repository of regex patterns [26]. This pattern matches a subset of valid website URLs using PCRE-specific syntax.

Worked Example, Original Pattern

```
(https?://)?((?:(\w+-)*\w+)\.)+(?:com|org)(\/?\w?-?=?_?\??&?)+[\.]?
```

3.2 First Normal Form (1NF)

Before defining the first normal form, we need to introduce a notion of structural simplification. We focus on simplification that preserves the use of PCRE syntax, but eliminates redundant and unnecessary syntax that can make the regex more difficult to read and understand, and can also lead to performance degradation.

Definition 1. *For the purposes of this subsection, we will say that a PCRE, P, is* structurally simplified *if it satisfies:*

Non-Ambiguity: *There are no nested quantifiers on single character patterns $(a^{\circ})^{\bullet}$ where $\circ, \bullet \in \{+, *, ?\}$. Such forms are collapsed (e.g.,* `(a+)*` $\rightarrow$ `a*`*).*

[2] `https://regexlib.com/REDetails.aspx?regexp_id=2508`

Repetition Collapse: *Sequences of identical elements are collapsed into a single quantified atom $R\{n,m\}$ (e.g.,* `\d\d\d?` $\rightarrow$ `\d{2,3}`*).*

Disjointness of Alternation: *For any alternation not including backreferences, recursive patterns, or subroutine calls, $R_1|R_2|\ldots|R_n$, the languages are pairwise disjoint: $\forall i \neq j, \mathcal{L}(R_i) \cap \mathcal{L}(R_j) = \emptyset$ (e.g.,* `\w|\d` $\rightarrow$ `\w`*).*

Kleene-Class Conversion: *All groups quantified by a Kleene closure of optional single character patterns, $(a?b?\ldots)^\circ$ where $\circ \in \{+,*\}$, are reduced to a Kleene-`*` closure of a single character class (e.g.,* `(a?b?c?d?e?)*` $\rightarrow$ `[a-e]*` *).*

Non-ambiguity and repetition collapse are trivial simplifications that are equivalent by construction. We provide proofs for the other two simplifications.

Lemma 1. *Let $R = R_1 \mid R_2 \mid \cdots \mid R_n$ be a PCRE alternation. If R does not include backreferences, recursive patterns, or subroutine calls, there exists an equivalent expression $R' = R'_1 \mid R'_2 \mid \cdots \mid R'_n$ such that $\mathcal{L}(R) = \mathcal{L}(R')$ and the sub-languages are pairwise disjoint, i.e., $\forall i, j \in \{1,\ldots,n\}, i \neq j \implies \mathcal{L}(R'_i) \cap \mathcal{L}(R'_j) = \emptyset$.*

Proof. We construct a sequence of expressions that partition the language. Let $R'_1 = R_1$. For $k \in \{2,\ldots,n\}$, we define R'_k such that its language satisfies:

$$\mathcal{L}(R'_k) = \mathcal{L}(R_k) \setminus \left(\bigcup_{j=1}^{k-1} \mathcal{L}(R'_j) \right).$$

Since the class of regular languages is closed under union and set difference, such a representation exists. By construction, any string $w \in \mathcal{L}(R)$ is contained in exactly one $\mathcal{L}(R'_k)$, specifically the smallest k such that $w \in \mathcal{L}(R_k)$. This transformation preserves the semantics of the first-match policy common in PCRE engines while eliminating redundant backtracking paths caused by overlapping sub-languages. □

Lemma 2. *Let $R = (a_1?a_2?\ldots a_n?)^\circ$ where $\circ \in \{+,*\}$ and a_i are single character patterns. R is equivalent to $R' = [a_1 a_2 \ldots a_n]^*$.*

Proof. This is a generalization of the *fusion* rule described by Kahrs and Runciman [13]. Observe that the original expression R can match any string that consists of zero or more occurrences of the characters $a_1, a_2, \ldots, a_n$ in any order, including the empty string. In the case where $\circ = +$, the optionality of every a_i ensures that $\varepsilon \in \mathcal{L}(R)$, and thus $(a_1?a_2?\ldots a_n?)^+ \equiv (a_1?a_2?\ldots a_n?)^* \equiv [a_1 a_2 \ldots a_n]^*$. □

Definition 2 (First Normal Form, 1NF)**.** *A PCRE, P, is in* First Normal Form *if it is structurally simplified and satisfies the following properties:*

Standard Quantifiers: *Standard operators $\{+,*,?\}$ are used instead of explicit range quantifiers $\{n,m\}$ or $\{n,\}$ whenever the range is equivalent to $\{1,\}$, $\{0,\}$, or $\{0,1\}$, respectively (e.g.,* `a{1}b{0,1}` $\rightarrow$ `a+b?`*).*

Collapsed Alternation of Characters: *Alternations of single character patterns, $a_1|a_2|\ldots|a_n$, are collapsed to the character class of their elements $[a_1a_2\ldots a_n]$.*

Character Class Simplification: *Canonical PCRE categories (e.g.,* `\w`*,* `\d`*, etc.) are used instead of character classes when they are equivalent. When character classes are necessary, characters are listed in lexicographic order and ranges are used when possible (e.g.,* `[abcAoE]` $\rightarrow$ `[AEa-co]`*).*

Character Class Elimination: *Character classes with a single element are replaced by that element and escaped as needed (e.g.,* `[a]` $\rightarrow$ `a`*).*

No Unnecessary Groups: *Parentheses are removed if they do not change the precedence of operators or serve as a required capture group for backreferencing (e.g.,* `(ab+)cd(\w)+` $\rightarrow$ `ab+cd\w+`*).*

We illustrate the application of the 1NF transformation rules to the regex from Section 3.1. The transformation, $\mathcal{T}_{1\text{NF}}(R_{\text{base}})$, (i) applies *No Unnecessary Groups* to non-functional parentheses, (ii) uses *Character Class Elimination* on character classes like `[\.]`, and (iii) employs *Kleene-Class Conversion* to collapse groups of optional characters as shown below:

Worked Example, 1NF

```
(https?://)?((\w+-)*\w+\.)+(?:com|org)[\/\w=\?&-]*\.?
```

3.3 Second Normal Form

Our next normal form further eliminates unnecessary syntax and deals with composition of sub-patterns of proper subsets. In many cases, one sub-pattern can *subsume* the other, especially when composed using quantifiers, which can lead to ambiguous matching and performance degradation.

Definition 3. *A PCRE, P, is* quantifier-subsumption-free (QS-free) *if it does not contain adjacent sub-expressions A and B such that $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, and either of the following conditions hold:*

Adjacency Subsumption: *P contains the sequence $A^{\bullet}B^{\circ}$, where $\circ$ is an unbounded quantifier $\{n,\}$ (including $*$ and $+$) and $\bullet$ is an arbitrary quantifier $\{n,m\}$ or $\{n,\}$. The surplus or optional repetitions of A beyond the minimum n are subsumed by the unbound closure B° (e.g.,* `[ab]+[a-d]*` $\rightarrow$ `[ab][a-d]*`*).*

Nested Subsumption: *P contains a quantified group $(AB^{\circ})^{\bullet}$ where $\circ$ is an unbounded quantifier $\{n,\}$ and $\bullet$ is either an unbounded quantifier $\{n,\}$ or a bounded quantifier $\{n,m\}$ with $n < m$ and $m > 1$. All but the lower bound of the outer quantifier $\bullet$ is subsumed by the inner closure*

B°, *as any subsequent* A *can be matched by the* B° *of the preceding iteration (e.g.,* `([ab][a-d]*)+` $\rightarrow$ `[ab][a-d]*` *and* `([ab][a-d]*){2,5}` $\rightarrow$ `([ab][a-d]*){2}`*).*

Lemma 3. *If a PCRE,* P*, is QS-free, the number of successful paths in the backtracking search tree for a given string* w *is strictly less than or equal to that of its QS-afflicted counterpart.*

Proof. Consider the QS-afflicted pattern P (i.e., P contains either adjacency or nested subsumption) and its QS-free counterpart P' such that $\mathcal{L}(P) = \mathcal{L}(P')$. We consider the branching factor of the backtracking search tree during the matching process.

For *Adjacency Subsumption*, any substring $s \in \mathcal{L}(A) \cap \mathcal{L}(B)$ introduces a choice point in the afflicted pattern $A^\bullet B^\circ$. The engine must decide whether to match s using a repetition of A or to transition to B°. In contrast, the QS-free pattern $A^n B^\circ$ eliminates this ambiguity by enforcing a minimum number of repetitions of A before transitioning to B°, thus reducing the number of paths.

In the pattern $(AB^\circ)^\bullet$ for *Nested Subsumption*, the engine faces a nondeterministic choice at the end of each B° iteration, where it can either start a new iteration of the group (which requires matching A again) or continue matching with B°. The QS-free pattern collapses this structure, eliminating the redundant iterations and thus reducing the branching factor.

In both cases, the QS-free pattern P' has a more deterministic structure that reduces the number of successful paths in the backtracking search tree compared to the QS-afflicted pattern P. □

Lemma 4. *Let* P *be a PCRE exhibiting quantifier subsumption. The transformation to a QS-free pattern* P' *preserves the language, such that* $\mathcal{L}(P) = \mathcal{L}(P')$*.*

Proof. For **Adjacency Subsumption**, let $P = A^\bullet B^\circ$ with $\mathcal{L}(A) \subseteq \mathcal{L}(B)$. Any string in $\mathcal{L}(A^\bullet)$ consists of k repetitions of some string in $\mathcal{L}(A)$, where $k \geqslant n$. Since $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, any repetitions beyond the minimum n can be equivalently matched by the unbound closure B° (recall $\mathcal{L}(B^+) \subset \mathcal{L}(B^*)$). Thus, $A^n A^{k-n} B^\circ \equiv A^n B^\circ$.

For ease of considering **Nested Subsumption**, we will handle Kleene-$*$ separately from all other quantifiers. First, let $P = (AB^\circ)^\bullet$ where $\bullet = \{n, \}, n > 0$. Since $\mathcal{L}(A) \subseteq \mathcal{L}(B)$, it follows that $\mathcal{L}(AB^\circ) \subseteq \mathcal{L}(BB^\circ) \subseteq \mathcal{L}(B^\circ)$. The outer Kleene closure may be collapsed: $(B^\circ)^\bullet \equiv (B^\circ)^n$. While the first n iterations of the closure are required to satisfy the lower bound of $\bullet$, any iterations $k > n$ are subsumed by the inner closure B°. Further, the leading A remains necessary to satisfy the pattern restrictions for the first n iterations of the group, but subsequent iterations are subsumed by the inner closure B°. The same argument holds if $\bullet = \{n, m\}$ with $m > 1$, since the first iteration of the group requires A, but subsequent iterations can be matched by B°.

Finally, assume instead $\bullet = *$. In this case, $\mathcal{L}(P) = \mathcal{L}((AB^\circ)^*)$. Since we established that $\mathcal{L}((AB^\circ)\{k, \}) \subseteq \mathcal{L}(AB^\circ)$ for all $k \geqslant 1$ via the subset relationship, the Kleene closure reduces to $\mathcal{L}(P) = \{\varepsilon\} \cup \mathcal{L}(AB^\circ)$. This is the

exact language of $(AB^{\circ})?$. Because $? \equiv \{0, 1\}$, the resulting pattern has a maximum repetition $m = 1$, satisfying the QS-free property. □

Definition 4 (Second Normal Form, 2NF)**.** *A PCRE, P, is in* Second Normal Form (2NF) *if it meets the requirements of 1NF, is* QS-free*, and satisfies:*

Minimal Escaping: *Escape characters are utilized only when they are strictly necessary to alter the matched string. Over-escaping is avoided to prevent pattern bloat (e.g.,* `\$` → `$` *or* `[\&\^]` → `[&^]`*).*

Feature Necessity: *Advanced PCRE features, including atomic groups, possessive quantifiers, and lazy quantifiers, are used only when they functionally change the match results or provide documented performance gains (e.g.,* `(?>[a-z])` → `[a-z]`*).*

We illustrate the application of the 2NF transformation rules to the worked example in 1NF from Section 3.2. The transformation, $\mathcal{T}_{\text{2NF}}(R_{\text{1NF}})$, applies the *Minimal Escaping* rule to remove unnecessary escape characters. In this example the 1NF pattern is already QS-free (because adjacent sub-patterns such as `\w+-` and `\w+\.` do not satisfy the $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ condition), so no transformations are applied for that rule, and the *Feature Necessity* rule is not applicable since there are no unnecessary advanced PCRE features used in the pattern.

Worked Example, 2NF

```
(https?://)?((\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?
```

3.4 Third Normal Form (3NF)

While 1NF and 2NF improve clarity and structural integrity, the Third Normal Form (3NF) optimizes for *runtime operational efficiency*. In many PCRE-compliant engines, parentheses serve a dual role as both grouping constructs and capture groups. By default, these engines allocate memory to store sub-match offsets to facilitate backreferencing; this implicit allocation increases memory pressure and degrades performance. The data structures used to track captured strings may require frequent reallocation to accommodate a large volume of sub-matches, leading to significant performance degradation with small, frequently captured groups (e.g., `(.)*`) and during states of catastrophic backtracking, where capture states must be saved and restored on the backtracking stack.

Definition 5 (Third Normal Form, 3NF)**.** *A PCRE, P, is in* Third Normal Form (3NF) *if it meets the requirements of 2NF and is* capture-minimal*: all grouping constructs use the non-capturing syntax* `(?:...)`*, unless the group is explicitly required for a backreference or is an intended output of the match (e.g.,* `(abc)+` → `(?:abc)+`*).*

In some regular expression engines (e.g., `C#`), the programmer can disable automatic capturing of groups, which can help mitigate the performance degradation caused by unnecessary capture groups. Because this is language-specific, and many PCRE engines do not have this option (such as Python's `re`), we propose manually transforming all unnecessary capture groups into non-capturing groups when using other regex engines.

The transformation of our running example from Section 3.3 into 3NF, $\mathcal{T}_{3\text{NF}}(R_{2\text{NF}})$, requires converting all parentheses into non-capturing groups.

Worked Example, 3NF

```
(?:https?://)?(?:(?:\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?
```

3.5 Fourth Normal Form (4NF)

Like 3NF, the Fourth Normal Form (4NF) also focuses on mitigating unanticipated performance degradation. A frequent source of inefficiency in PCRE-based systems is the presence of adjacent quantified sub-patterns that match overlapping character sets. This is a more general form of the quantifier subsumption issue addressed in 2NF, but without the requirement of a strict subset relationship. In such cases, the backtracking engine may explore an exponential number of ways to partition a string, leading to significant runtime spikes even for relatively short inputs. 4NF addresses this by using *atomic groups* `(?>...)` and, when available, *possessive quantifiers* (e.g., `*+`). These constructs commit the engine to a greedy match, effectively pruning the search tree by preventing the re-evaluation of atomized sub-paths. 4NF aims to prune the backtracking search space by *hardening* the pattern against unanticipated performance degradation. The goal is to maximize the use of atomic constructs while strictly preserving the semantics of the original expression.

Definition 6. *A PCRE, P, is* atomic-optimal *if it satisfies the requirements of 3NF and the following structural criteria:*

Maximal Atomization: *For every quantified sub-expression within P, any prefix that can be atomized without altering the overall language, $\mathcal{L}(P)$, is contained within an atomic group or governed by a possessive quantifier.*

Lazy-Atomic Reduction: *P contains no sub-expressions of the form* `(?>A??)` *or* `(?>A*?)`*, as the atomization of a lazy match is functionally redundant and reduces to a fixed-width match.*

Possessive Canonicalization: *All atomic closures in P are expressed using possessive quantifiers (e.g.,* `++`*,* `*+`*) rather than atomic groups* `(?>...)` *whenever the two are functionally equivalent.*

Remark 1. While atomic groups are highly effective for performance hardening, their implementation varies across environments. For example, the Python `re`

library only introduced support for these features in version 3.11, while the third-party `regex` library has supported them for much longer.

Definition 7 (Fourth Normal Form, 4NF)**.** *A PCRE, P, is in* Fourth Normal Form (4NF) *if it is* atomic-optimal.

Lemma 5. *The transformation, $\mathcal{T}_{4NF}$, preserves $\mathcal{L}(P)$ provided that the atomized sub-expression is* possessive-safe*: the greedy commitment of the atomized prefix does not prevent the discovery of a valid match that would have been reachable via standard backtracking.*

Proof. Let P' be the atomic-optimal version of possessive-safe pattern P. The inclusion $\mathcal{L}(P') \subseteq \mathcal{L}(P)$ follows from the fact that atomic constructs only restrict the backtracking behavior of the engine without expanding the set of matched strings. Conversely, the inclusion $\mathcal{L}(P) \subseteq \mathcal{L}(P')$ holds because the possessive-safety condition ensures that a greedy commitment does not preclude the discovery of a valid global match. Since PCRE engines explore paths in a greedy-first order, atomization merely *locks* the initial path the engine would have already prioritized, preventing only the redundant re-evaluation of unsuccessful alternative branches. Consequently, any string $w \in \mathcal{L}(P)$ remains reachable in P', ensuring that $\mathcal{L}(P) = \mathcal{L}(P')$. □

We illustrate the application of 4NF to the regex from Section 3.4. In the sub-pattern `(?:(?:\w+-)*\w+.)+`, the nesting of quantifiers creates a potential for excessive backtracking. By applying *Possessive Canonicalization* to the inner repetition, we ensure that the engine does not revisit the `\w+-` sequences in the same iteration, thereby stabilizing the runtime performance against unanticipated degradation:

Worked Example, 4NF

```
(?:https?://)?(?:(?:\w+-)*+\w+\.)+(?:com|org)[/\w=?&-]*\.?
```

4 Experimental Methodology

In this section, we describe our methodology for evaluating the impact of our proposed normal forms on the readability and runtime performance of PCRE.

4.1 Processing Engines and Hardware

In our evaluation, we measure performance across two popular Python PCRE libraries: the standard library `re` and version 2026.2.28 of the third-party `regex` library. More than 30% of public python modules use at least one regex [7], and because inefficient backtracking can cause severe operational failures, performance improvements at this scale are highly significant. In one example, recent

research demonstrated that unoptimized regex patterns can induce 0.6–1.2 seconds of matching delay per execution, which exhausts engine matching limits to bypass security detections entirely [18]

All tests were run on an 8-core Intel Xeon E5-1620 v4 CPU with 64GB of RAM running Ubuntu 22.04.5, Python 3.14.3, and Linux kernel 5.15.0-161.

4.2 Benchmark Selection

We perform an empirical evaluation of real-world regex patterns to assess the practical impact of our normal forms. We randomly sampled 40 regex patterns from RegExLib [26], a widely used repository, and removed any patterns that failed to parse with both `re` and `regex` resulting in a final set of 36 patterns. One of the authors then manually inspected each regex and applied the normalization transformations to create the 1NF, 2NF, 3NF, and 4NF versions of each pattern, resulting in a total of 180 distinct regex patterns (36 original + 144 normalized).

4.3 Test Input Generation

To evaluate the runtime performance of the original and normalized regex patterns, we need a large corpus of test strings. Unfortunately, RegExLib does not provide this, and regex pattern generators often focus on theoretical syntax or a small subset of features. However, we do not require a perfect set of test strings; rather, we needed a sufficiently large and diverse set of strings that exercise the matching behavior of the patterns. Inputs that are immediately rejected by a pattern are not useful, as much of the matching behavior is left unexercised.

To utilize existing input generation tools, we first strip PCRE-specific features from our patterns to create simplified variants that these generators can process. These inputs are sufficiently close to positive examples for the purpose of exercising the matching behavior of the patterns consistently across the original and normalized versions. For each benchmark pattern, we use `rstr` [20] to generate up to 150 positive examples.[3] Then, for each positive example, we generate 30 mutations by randomly removing, inserting, or replacing a character to create non-trivial negative examples. Because these negative examples are similar to positive matches, the PCRE engine is likely to attempt to match the input against the pattern instead of rejecting the sample immediately. Thus, for each of the 36 original patterns, we generate up to 4,650 test strings (150 positive + 30 mutations per positive). We then shuffle all of these generated test strings and merged them into a single input file to use across all benchmarks. In total, this test input file is 5.4 GB.

[3] In some cases, the simplified pattern may have a finite language or a very small number of matching strings, in which case we generate the maximum number of possible positive examples.

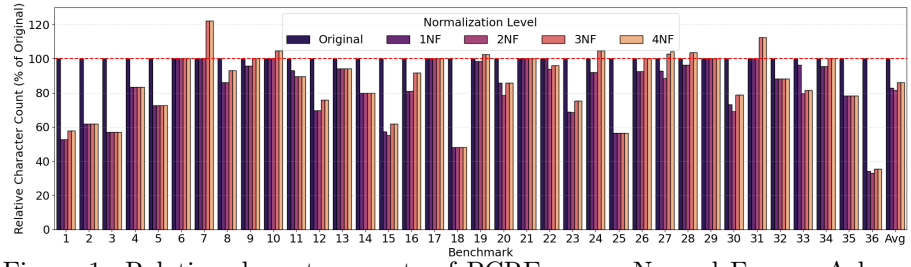


Figure 1: Relative character counts of PCRE across Normal Forms. A lower value represents a reduction in the length of the regex. In most cases, the character count decreases with average normalized expressions 86% the length of the original.

5 Evaluation

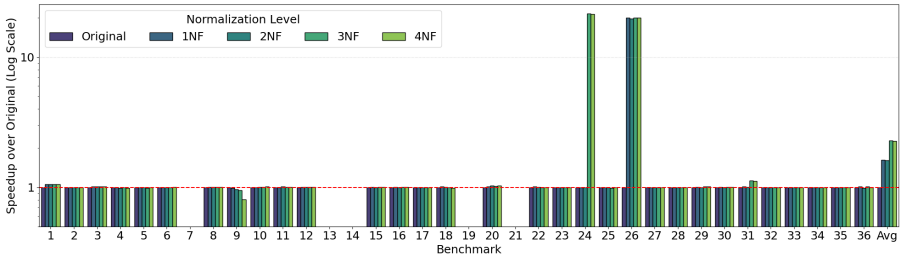
Our empirical evaluation of normal forms is primarily concerned with the following two research questions: (i) how do the proposed normal forms affect the length of regex patterns, and (ii) do the proposed normal forms have a net positive effect on the runtime performance of regex pattern matching?

5.1 Normalization and Readability

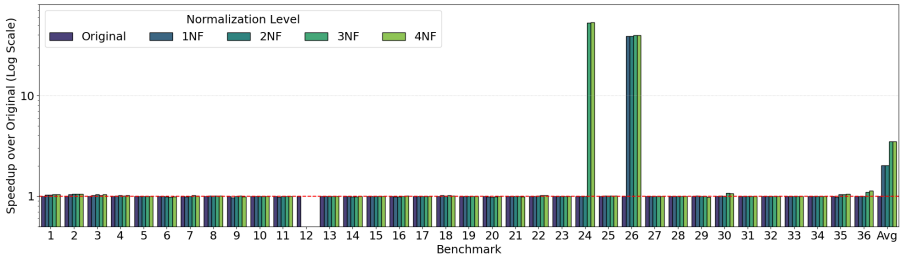
As noted in Section 3, both 1NF and 2NF are designed to improve the readability of regex patterns by eliminating unnecessary syntax and structural issues. A comprehensive human study of readability falls outside the scope of this paper, but we can use the pattern length as a proxy for readability. Chapman et al. observed a relationship between expression length and comprehension, for example [5]. Figure 1 shows the relative character counts of the original and normalized patterns across all 36 benchmarks. Overall, we see a significant reduction in character count from the original patterns to the 1NF and 2NF versions, with an average character reduction of 18.4%. The 3NF and 4NF versions show a slight increase in character count compared to 2NF, but still maintain a significant reduction compared to the original patterns, with an average character reduction of 14%. Because 1NF and 2NF remove unnecessary syntax, duplication, and redundancy, they tend to reduce the character count of the patterns, which can be a proxy for improved readability.

5.2 Runtime Performance

We next compare the runtime execution of normal forms to the original patterns. As described in Section 4.1, we evaluate runtime using two popular Python modules, `re` and `regex`. For each regex, we time the execution to process the test input file and calculate the average runtime across 10 runs for each pattern and its normalized versions. Because our benchmarks may contain unintentional



(a) Speedup using the `re` library.



(b) Speedup using the `regex` library.

Figure 2: Runtime comparison of normal forms. Higher values indicate improved runtime performance across the `re` (a) and `regex` (b) libraries. Note the log scale on the y-axis. On average, the normalized patterns show a speedup of 2.3x in `re` and 3.5x in `regex` compared to the original patterns.

performance degradation (REDoS in the worst case), we follow best practices (see Davis et al. [8]) and cut off execution of any one trial at 25 minutes.

Figure 2a and Figure 2b show the relative speedup of the normalized patterns compared to the original patterns for the `re` and `regex` libraries, respectively. For most benchmarks, we observe roughly similar performance across the original and normalized patterns, meaning that the normalization does not have a negative impact on runtime performance.

The built-in `re` library does not support certain PCRE features and therefore fails to parse benchmarks 7, 13, 14, 19, and 21. Further, `re` times out for benchmarks 2, 17, 27, 28, 32, 33, 34, and 35 (and is thus reported as a 1x speedup in the figure). For the `regex` library, there are no failures for any of the benchmarks, but benchmarks 10, 13, 15, 17, 19, 23, 27, 28, 31,32, 33, and 34 time out.

For both libraries, we observe a significant improvement in runtime performance for the 3NF and 4NF versions in benchmarks 24 and 26. Indeed, the original patterns for these benchmarks timed out in both libraries but the 3NF and 4NF versions were able to process the input in less than two minutes. When manually inspecting these patterns, we noted that they contained significant use of parenthetical groups that did not need to be captured.

Benchmark 12 unexpectedly times out in the `regex` library following the 1NF

transformation of `[0-9]` to `\d`. We attribute this to an implementation-specific quirk in the handling of Unicode-aware shorthands, as the same transformation remained performant in the `re` library and across other benchmarks.

For benchmarks that did not time out, we see a modest improvement in runtime performance for the normalized patterns, and a significant improvement in some cases. This demonstrates that our normal forms can help mitigate unanticipated performance degradation. On average, 4NF benchmarks have a 2.3x speedup in `re` and a 3.5x speedup in `regex` compared to the original patterns.

6 Conclusions and Future Work

We propose four normal forms for PCRE patterns that balance structural clarity (1NF–2NF) with operational efficiency (3NF–4NF). These normal forms motivate future development of automated refactoring and linting algorithms, validating performance gains across languages, and conducting human-subject studies to measure the qualitative impact of normalization on developer comprehension. Ultimately, our empirical evaluation of 36 regex patterns from RegExLib validates the effectiveness of our normal forms, yielding an average 14% reduction in pattern length and a 2.3–3.5x improvement in runtime performance, on average. These results demonstrate that systematic normalization is a potent, low-cost strategy for hardening software against unanticipated performance degradation while simultaneously removing unnecessary complexity from patterns.

Acknowledgments

This work was supported in part by the National Science Foundation (Award CCF-2211751), and PRISM, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA. Claud, Copilot, ChatGPT, and Gemini were used in the preparation of this document for editing, revising, and grammar checking. Generative feedback was fully reviewed and edited by the authors, and all substantive content, including the design of the normal forms, the experimental methodology, and the analysis of results are the original work and responsibility of the authors.

References

[1] Barnett, M.: regex: Alternative regular expression module, to replace re. `https://pypi.org/project/regex/`, accessed: 2026-03-26

[2] Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems. p. 145–154.

ANCS '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1323548.1323573

[3] Brüggemann-Klein, A.: Regular expressions into finite automata. In: Simon, I. (ed.) LATIN '92. pp. 87–98. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)

[4] Chapman, C., Stolee, K.T.: Exploring regular expression usage and context in python. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. p. 282–293. ISSTA 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2931037.2931073

[5] Chapman, C., Wang, P., Stolee, K.T.: Exploring regular expression comprehension. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. p. 405–416. ASE '17, IEEE Press (2017)

[6] Crosby, S.: Denial of service through regular expressions. In: 12th USENIX Security Symposium (USENIX Security 03). USENIX Association, Washington, D.C. (Aug 2003)

[7] Davis, J.C., Coghlan, C.A., Servant, F., Lee, D.: The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 246–256. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236027

[8] Davis, J.C., Williamson, E.R., Lee, D.: A sense of time for JavaScript and node.js: First-Class timeouts as a cure for event handler poisoning. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 343–359. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/davis

[9] Google: Re2. https://github.com/google/re2, accessed 2026-03-26

[10] Gruber, H., Gulan, S.: Simplifying regular expressions: a quantitative perspective. In: Proceedings of the 4th International Conference on Language and Automata Theory and Applications. p. 285–296. LATA'10, Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13089-2_24

[11] Hazel, P.: PCRE - Perl compatible regular expressions. https://www.pcre.org (2026), accessed: March 17, 2026

[12] Intel: Hyperscan. https://github.com/intel/hyperscan (2023), accessed 2026-03-26

[13] Kahrs, S., Runciman, C.: Simplifying regular expressions further. Journal of Symbolic Computation **109**, 124–143 (2022). https://doi.org/https://doi.org/10.1016/j.jsc.2021.08.003, https://www.sciencedirect.com/science/article/pii/S0747717121000572

[14] Kent, W.: A simple guide to five normal forms in relational database theory. Commun. ACM **26**(2), 120–125 (Feb 1983). https://doi.org/10.1145/358024.358054

[15] Kim, S.H., Im, H., Ko, S.K.: Efficient enumeration of regular expressions for faster regular expression synthesis. In: Maneth, S. (ed.) Implementation and Application of Automata. pp. 65–76. Springer International Publishing, Cham (2021)

[16] Li, Y., Chen, Z., Cao, J., Xu, Z., Peng, Q., Chen, H., Chen, L., Cheung, S.C.: Redoshunter: A combined static and dynamic approach for regular expression dos detection. In: USENIX Security Symposium (2021), https://api.semanticscholar.org/CorpusID:237258630

[17] Li, Y., Sun, Y., Xu, Z., Cao, J., Li, Y., Li, R., Chen, H., Cheung, S.C., Liu, Y., Xiao, Y.: RegexScalpel: Regular expression denial of service (ReDoS) defense by Localize-and-Fix. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4183–4200. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/li-yeting

[18] Liu, Y., Çakar, B., Agrawal, A., Seo, M., Davis, J.C., Lee, D.: Regular expression denial of service induced by backreferences. ArXiv **abs/2602.21459** (2026), https://api.semanticscholar.org/CorpusID:286011585

[19] Maududie, A., Retnani, W.E.Y., Rohim, M.A.: An approach of web scraping on news website based on regular expression. In: 2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT). pp. 203–207 (2018). https://doi.org/10.1109/EIConCIT.2018.8878550

[20] McCollam, B.: rstr = random strings in python. https://pypi.org/project/rstr/, accessed: 2026-03-26

[21] Mulder, M., Nezlek, G.: Creating protein sequence patterns using efficient regular expressions in bioinformatics research. In: 28th International Conference on Information Technology Interfaces, 2006. pp. 207–212 (2006). https://doi.org/10.1109/ITI.2006.1708479

[22] Nogami, T., Terauchi, T.: On the Expressive Power of Regular Expressions with Backreferences. In: Leroux, J., Lombardy, S., Peleg, D. (eds.) 48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 272, pp. 71:1–71:15. Schloss

Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.MFCS.2023.71

[23] Okuboyejo, O.Y., Ewert, S., Sanders, I.: Goofs in the class: Students' errors and misconceptions when learning regular expressions. In: Wells, G., Nxozi, M., Tait, B. (eds.) ICT Education. pp. 57–71. Springer International Publishing, Cham (2021)

[24] Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, Seventh Edition. McGraw-Hill Book Company (2020), `https://www.db-book.com/`

[25] Sipser, M.: Introduction to the Theory of Computation. Thomson Course Technology, 2nd edn. (2006)

[26] Smith, S.: RegExLib.com: Regular expression library. `http://regexlib.com` (2026), accessed: 2026-03-26

[27] Spishak, E., Dietl, W., Ernst, M.D.: A type system for regular expressions. In: Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs. pp. 20–26. FTfJP '12 (2012). https://doi.org/10.1145/2318202.2318207

[28] Sung, S., Cheon, H., Han, Y.S.: How to settle the ReDoS problem: Back to the classical automata theory. In: Implementation and Application of Automata: 26th International Conference, CIAA 2022. pp. 34–49. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-031-07469-1_3

[29] Wadden, J., Dang, V., Brunelle, N., II, T.T., Guo, D., Sadredini, E., Wang, K., Bo, C., Robins, G., Stan, M., Skadron, K.: ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In: International Symposium on Workload Characterization. pp. 1–12. IISWC '16 (Sept 2016). https://doi.org/10.1109/IISWC.2016.7581271

[30] Yang, Y.H., Prasanna, V.: High-performance and compact architecture for regular expression matching on fpga. IEEE Transactions on Computers **61**(7), 1013–1025 (2012). https://doi.org/10.1109/TC.2011.129

[31] Zhang, L., Deep, S., Patel, J.M., Sankaralingam, K.: Regular expression indexing for log analysis. Proc. ACM Manag. Data **3**(6) (Dec 2025). https://doi.org/10.1145/3769820