Normal Forms for Perl-Compatible Regular Expressions

Oleg Efimov[1] Tommy Tracy II[2] Whiting Zheng[2]
Kevin Skadron[2] **Kevin Angstadt**[1]

[1]St. Lawrence University, Canton, NY, USA
[2]University of Virginia, Charlottesville, VA, USA

30th International Conference on Implementation and Application of Automata (CIAA 2026)

Tool Track Presentation

Lead Author Announcement: Oleg Efimov recently graduated with his B.S. from St. Lawrence University and is **actively seeking a Graduate Research Assistantship (RA) / Ph.D. position**!

Bridging Formal Languages and Systems Engineering

Donald Knuth on Theory and Practice

"If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice."

— Donald Knuth

Theory vs. Practice

Donald Knuth on Theory and Practice

"If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice."

— Donald Knuth

Automata Theory vs. Systems

- **The Automata Theory View**: Regular expressions (regex) are ideal formal abstractions over finite alphabets with known decision properties.
- **The Systems View**: Industrial software relies heavily on **Perl-Compatible Regular Expressions (PCRE)**, which extend regex syntax with backreferences, lookarounds, and atomic groups.

Motivation 1: Real-World Regex Maintenance Concerns

Ubiquity in Enterprise Infrastructure

- **Splunk**: Mission-critical for enterprise log parsing, field extraction, and SIEM filtering.
- **Sigma Rules**: Standardized threat detection signatures across security analytics platforms.
- Over **30% of public Python modules** rely directly on regex pattern matching.

The Engineering & Readability Maintenance Trap

- Regexes are notoriously difficult to read, debug, and maintain (often treated as "write-only code").
- Ad-hoc developer construction leads to redundant capture groups, over-escaping, and overlapping quantifiers.
- As systems evolve, maintaining regexes becomes an expensive security and correctness challenge.

Motivation 2: Automated Regex Synthesis and Search Space

Automated Regex Generation & Repair

- Automated synthesis tools (e.g., *AutomataSynth* (Angstadt et al., 2020), DSL translation, repair frameworks) attempt to generate or fix regexes automatically from specifications or examples.
- **The Search Space Bottleneck**: PCRE features (and standard regex!) create a vast, redundant search space of syntactically distinct patterns matching the exact same language.

The Need for Canonical Search Spaces

- **Without Constraints**: Synthesis tools explore redundant candidate expressions, leading to combinatorial search explosion and unreadable output.

The Unifying Threat: Unintentional Performance Degradation

Key Insight: Both Human & Automated Traps Cause Degradation

Whether created by human maintenance errors or unconstrained synthesis tools, small changes to PCRE patterns can result in **Unintentional Performance Degradation**.

The Unifying Threat: Unintentional Performance Degradation

Key Insight: Both Human & Automated Traps Cause Degradation

Whether created by human maintenance errors or unconstrained synthesis tools, small changes to PCRE patterns can result in **Unintentional Performance Degradation**.

What Causes Unintentional Performance Degradation in Production?

- Innocuous syntactic choices (e.g., nested quantifiers `(.)*`, redundant capturing groups, or overlapping alternations) trigger exponential state exploration in backtracking CPU engines.
- In the worst case, performance degradation manifests as catastrophic backtracking — creating Regular Expression Denial of Service (**REDoS**) vulnerabilities.

The Solution: A Hierarchy of PCRE Normal Forms

Database Systems Analogy

- Relational database systems can organize the same data in various schemas
- Choices in schema have *significant* performance and data integrity implications
- Relational database theory introduced Normal Forms (1NF–5NF; Codd 1970, Kent 1983) to eliminate data redundancy and anomalies

The Solution: A Hierarchy of PCRE Normal Forms

Database Systems Analogy

- Relational database systems can organize the same data in various schemas
- Choices in schema have *significant* performance and data integrity implications
- Relational database theory introduced Normal Forms (1NF–5NF; Codd 1970, Kent 1983) to eliminate data redundancy and anomalies

Introducing: Four PCRE Normal Forms

PCRE Normal Forms provide a formal structural framework to help practitioners eliminate Unintentional Performance Degradation **by construction** while simultaneously enhancing code readability.

- Lower forms prioritize readability (and simplicity)
- Higher forms prioritize performance
- Hierarchical: **4NF** $\subseteq$ **3NF** $\subseteq$ **2NF** $\subseteq$ **1NF**

The Solution: A Hierarchy of PCRE Normal Forms

Database Systems Analogy

- Relational database systems can organize the same data in various schemas
- Choices in schema have *significant* performance and data integrity implications
- Relational database theory introduced Normal Forms (1NF–5NF; Codd 1970, Kent 1983) to eliminate data redundancy and anomalies

Introducing: Four PCRE Normal Forms

PCRE Normal Forms provide a formal structural framework to help practitioners eliminate Unintentional Performance Degradation **by construction** while simultaneously enhancing code readability.

- Lower forms prioritize readability (and simplicity)
- Higher forms prioritize performance
- Hierarchical: **4NF** $\subseteq$ **3NF** $\subseteq$ **2NF** $\subseteq$ **1NF**

Note: Normal forms exist to simplify theoretical regular expressions (e.g., strong star normal form [Gruber and Gulan, 2010]), but tend to aid formal methods or improve theoretical bounds.

Running Worked Example: Initial Pattern

Real-World URL Pattern (RegExLib ID 2508)

Simplified URL-matching expression written by a practitioner:

Original Pattern

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

Running Worked Example: Initial Pattern

Real-World URL Pattern (RegExLib ID 2508)

Simplified URL-matching expression written by a practitioner:

Original Pattern

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

Identified Structural & Performance Flaws

1. **Redundant Parentheses**: Unnecessary nested capture groups `((?:(\w+-)*\w+)\.)+`.
2. **Over-Escaping**: Escape inside single-character set `[\.]?`.
3. **Kleene-Class Ambiguity**: Group of optional characters `(\/?\w?-?=?_?\??&?)+`.
4. **Capture Allocation Overhead**: Allocates heap/stack frames for groups that are never backreferenced.

A Note on Notation

- From this point forward, *regex* and *PCRE* are synonymous
- Sub-expressions are capitalized (e.g., A, B)
- Exact character matches are lowercase (e.g., a, b, c)
- Generic Quantifiers
  - $\circ$ represents *unbounded* quantifiers (e.g., $*$, $+$, $\{n,\}$)
  - $\bullet$ represents *arbitrary* quantifiers (e.g., ?, $\{n\}$, $\{n,m\}$, $+$, $*$)
- "It's in the paper"
  - Our paper in the tools track proceedings gives more formal definitions of terms and notation
  - Proofs of major properties may also be found in the paper

First Normal Form (1NF): Structural Simplification

Objective:

Remove redundant syntax & enforce canonical categories

1NF Features:

- **Non-Ambiguity**: Collapse nested quantifiers on single characters $(a^{\circ})^{\bullet} \to a^{*}$.
- **Repetition Collapse**: Replace sequences of identical elements with explicit ranges.
- **Disjointness of Alternation**: Remove language overlap from alternations (`\w|\d` $\to$ `\w`).
- **Kleene-Class Conversion**: Collapse $(a?b?c?)^{+} \to [a-c]^{*}$.
- **Character Class Rules**: Lexicographic ordering, single-element elimination (`[a]` $\to$ `a`).
- **No Unnecessary Groups**: Eliminate non-functional parentheses.

First Normal Form (1NF): Structural Simplification

Objective:

Remove redundant syntax & enforce canonical categories

1NF Features:

- **Non-Ambiguity**: Collapse nested quantifiers on single characters $(a^{\circ})^{\bullet} \rightarrow a^{*}$.
- **Repetition Collapse**: Replace sequences of identical elements with explicit ranges.
- **Disjointness of Alternation**: Remove language overlap from alternations (`\w|\d` $\rightarrow$ `\w`).
- **Kleene-Class Conversion**: Collapse $(a?b?c?)^{+} \rightarrow [a-c]^{*}$.
- **Character Class Rules**: Lexicographic ordering, single-element elimination (`[a]` $\rightarrow$ `a`).
- **No Unnecessary Groups**: Eliminate non-functional parentheses.

Worked Example → First Normal Form (1NF)

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

Highlighted Text shows changes relative to Original pattern (collapsed nested groups, Kleene-class conversion, and single-element class elimination).

Second Normal Form (2NF): Quantifier Subsumption

Objective:

Remove ambiguity from composition of sub-patterns of proper subsets (i.e., one pattern *subsumes* another), remove all *unnecessary* PCRE features, and minimize escaping (e.g., `[\&\^]` $\rightarrow$ `[&^]`)

2NF Feature:

A pattern is **Quantifier-Subsumption-Free (QS-free)** if adjacent sub-expressions A and B with $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ do not exhibit redundant choice points:

- **Adjacency Subsumption**: $A^{\bullet}B^{\circ} \rightarrow A^{n}B^{\circ}$ (e.g., `[ab]+[a-d]*` $\rightarrow$ `[ab][a-d]*`).
- **Nested Subsumption**: $(AB^{\circ})^{\bullet} \rightarrow (AB^{\circ})^{n}$.

Second Normal Form (2NF): Quantifier Subsumption

Objective:

Remove ambiguity from composition of sub-patterns of proper subsets (i.e., one pattern *subsumes* another), remove all *unnecessary* PCRE features, and minimize escaping (e.g., `[\&\^]` $\rightarrow$ `[&^]`)

2NF Feature:

A pattern is **Quantifier-Subsumption-Free (QS-free)** if adjacent sub-expressions A and B with $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ do not exhibit redundant choice points:

- **Adjacency Subsumption**: $A^{\bullet}B^{\circ} \rightarrow A^{n}B^{\circ}$ (e.g., `[ab]+[a-d]*` $\rightarrow$ `[ab][a-d]*`).
- **Nested Subsumption**: $(AB^{\circ})^{\bullet} \rightarrow (AB^{\circ})^{n}$.

Worked Example $\rightarrow$ Second Normal Form (2NF)

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

Highlighted Text shows minimal escaping changes relative to 1NF (`[\/\w=\?&-]*` $\rightarrow$ `[/\w=?&-]*`).

Third Normal Form (3NF): Capture Minimality

Objective:

- Reduce memory allocations for sub-matches
- **The Systems Problem**: Standard parentheses `(...)` force regex engines to allocate memory for tracking sub-match offsets

3NF Feature:

- **Capture-Minimal Rule**: Convert all grouping constructs to non-capturing syntax `(?:...)` unless the group is explicitly required for backreferencing or match output.

Third Normal Form (3NF): Capture Minimality

Objective:

- Reduce memory allocations for sub-matches
- **The Systems Problem**: Standard parentheses `(...)` force regex engines to allocate memory for tracking sub-match offsets

3NF Feature:

- **Capture-Minimal Rule**: Convert all grouping constructs to non-capturing syntax `(?:...)` unless the group is explicitly required for backreferencing or match output.

Worked Example → Third Normal Form (3NF)

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

Highlighted Text shows non-capturing groups `(?:...)` added relative to 2NF to eliminate stack allocation overhead.

Fourth Normal Form (4NF): Atomic Optimality

Objective:

Harden against catastrophic backtracking (e.g., REDoS)

4NF Features:

- **Atomic Optimality**: Commit the engine to greedy choices, permanently pruning dead backtracking branches.
- **Possessive Canonicalization**: Prefer possessive quantifiers (`*+`, `++`) over atomic groups (`?>...`) when equivalent.
- **Possessive-Safety Condition**: Greedy commitment preserves original language $\mathcal{L}(P)$.

Fourth Normal Form (4NF): Atomic Optimality

Objective:

Harden against catastrophic backtracking (e.g., REDoS)

4NF Features:

- **Atomic Optimality**: Commit the engine to greedy choices, permanently pruning dead backtracking branches.
- **Possessive Canonicalization**: Prefer possessive quantifiers (*+, ++) over atomic groups (?>...) when equivalent.
- **Possessive-Safety Condition**: Greedy commitment preserves original language $\mathcal{L}(P)$.

Worked Example → Fourth Normal Form (4NF)

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

Highlighted Text shows possessive quantifier *+ replacing * relative to 3NF to harden against catastrophic backtracking.

Summary Review: Worked Example Transformation

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

Summary Review: Worked Example Transformation

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

1NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[\/\w=\?&-]*\.?` *(Group collapse, Kleene-class)*

Summary Review: Worked Example Transformation

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

1NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[\/\w=\?&-]*\.?` *(Group collapse, Kleene-class)*

2NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Minimal escaping)*

Summary Review: Worked Example Transformation

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

1NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[\/\w=\?&-]*\.?` *(Group collapse, Kleene-class)*

2NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Minimal escaping)*

3NF `(?:https?://)?(?:(?:\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Capture minimality (?:...))*

Summary Review: Worked Example Transformation

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

1NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[\/\w=\?&-]*\.?` *(Group collapse, Kleene-class)*

2NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Minimal escaping)*

3NF `(?:https?://)?(?:(?:\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Capture minimality (?:...))*

4NF `(?:https?://)?(?:(?:\w+-)*+\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Possessive hardening *+)*

Summary Review: Worked Example Transformation

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

1NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[\/\w=\?&-]*\.?` *(Group collapse, Kleene-class)*

2NF `(https?://)?((\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Minimal escaping)*

3NF `(?:https?://)?(?:(?:\w+-)*\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Capture minimality (?:...))*

4NF `(?:https?://)?(?:(?:\w+-)*+\w+\.)+(?:com|org)[/\w=?&-]*\.?` *(Possessive hardening *+)*

Overall Impact

1NF and **2NF** reduce pattern size while **3NF** and **4NF** relax this constraint to improve engine performance.

Evaluating PCRE Normal Forms

PCRE Performance has Real-World Impact

- More than 30% of public Python modules use at least one regex (Davis et al., 2018).
- Unoptimized regex patterns can introduce 0.6–1.2 seconds of matching delay per execution (Liu et al., 2026).

Guiding Question

To what extent does applying PCRE normal forms to real-world patterns improve maintainability and reduce unintentional performance degradation?

Note: A human subjects study measuring the readability and maintainability of normal forms falls outside of the scope of work presented here.

Experimental Methodology

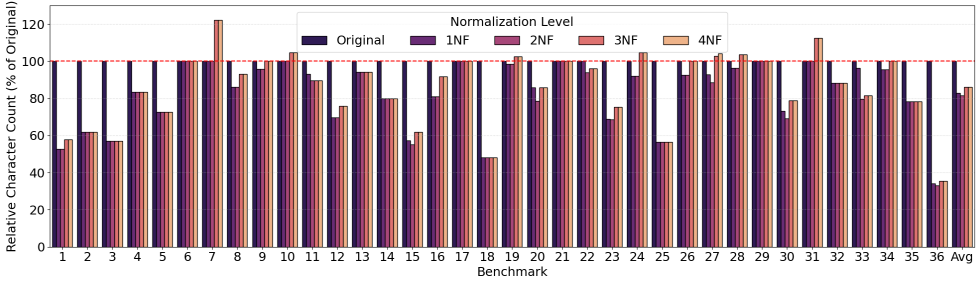
Benchmark Selection

- Randomly sampled **36 regex patterns** from RegExLib repository.
- Applied 1NF–4NF transformations manually to construct **180 total variants**.
- Evaluated across two popular Python libraries: standard `re` and third-party `regex` (v2026.2.28).

Test Input Corpus (5.4 GB)

- Generated up to 150 positive example strings per pattern using `rstr`.
- Generated **30 mutated negative variants** per positive string (character insertion/deletion/substitution).
- **Why Mutated Negatives?** Forces PCRE engine deep into matching paths before rejection, thoroughly exercising backtracking behavior.

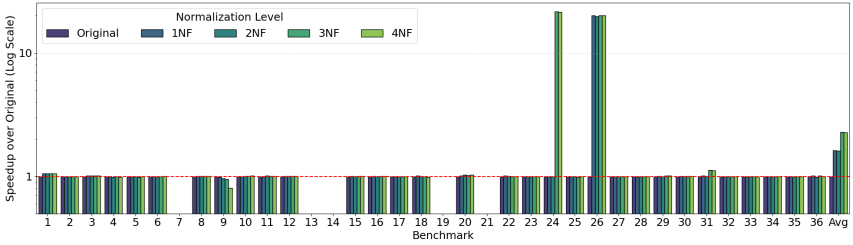
Empirical Evaluation: Expression Length and Readability



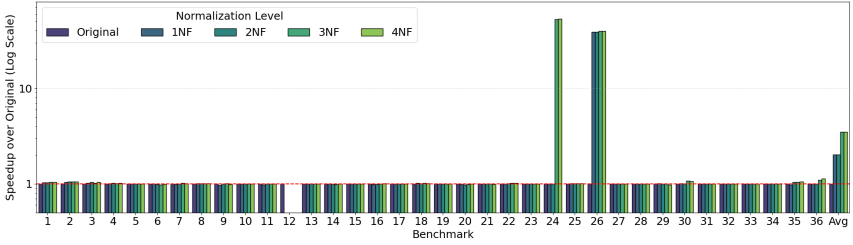
Readability Proxy Metrics

- Prior empirical research (Chapman et al., 2017) establishes an inverse relationship between pattern length and developer comprehension.
- **1NF & 2NF**: Average character reduction of **18.4%**.
- **3NF & 4NF**: Maintains an average character reduction of **14%** despite introducing `(?:...)` non-capturing syntax and possessive operators.

Empirical Evaluation: Runtime Execution Speedup Plots



(a) Speedup using Python standard `re` library (Log scale)



(b) Speedup using third-party `regex` library (Log scale)

Empirical Evaluation: Performance and REDoS Hardening Insights

Average Execution Speedups

- **Python `re` Library**: Average speedup of **2.3x** across benchmark suite.
- **Python `regex` Library**: Average speedup of **3.5x** across benchmark suite.

Empirical Evaluation: Performance and REDoS Hardening Insights

Average Execution Speedups

- **Python `re` Library**: Average speedup of **2.3x** across benchmark suite.
- **Python `regex` Library**: Average speedup of **3.5x** across benchmark suite.

REDoS Hardening & Timeout Mitigation

- Benchmarks 24 & 26 originally timed out (exceeding the **25-minute execution cutoff**; Davis et al., 2018).
- 3NF and 4NF normalized versions completed execution in **under 2 minutes** (>100x speedup).
- Converting unnecessary capture groups to `(?:...)` in 3NF eliminated massive stack allocation overhead during deep backtracking.

Empirical Evaluation: Performance and REDoS Hardening Insights

Average Execution Speedups

- **Python `re` Library**: Average speedup of **2.3x** across benchmark suite.
- **Python `regex` Library**: Average speedup of **3.5x** across benchmark suite.

REDoS Hardening & Timeout Mitigation

- Benchmarks 24 & 26 originally timed out (exceeding the **25-minute execution cutoff**; Davis et al., 2018).
- 3NF and 4NF normalized versions completed execution in **under 2 minutes** (>100x speedup).
- Converting unnecessary capture groups to `(?:...)` in 3NF eliminated massive stack allocation overhead during deep backtracking.

Library Quirks

Benchmark 12 timed out in `regex` following 1NF conversion of `[0-9]` to `\d` due to Unicode shorthand handling in `regex`, while remaining performant in `re`.

Future Opportunities

Automated Tooling

- **Static Linters**: Integrate 1NF–4NF checks into CI/CD pipelines to flag un-normalized regexes before deployment.
- **PCRE Rewriter**: Automatically rewrite PCREs to apply normal forms (likely theoretical limitations).
- **Synthesis Engines**: Constrain pattern search space in tools to guarantee efficient, readable outputs by construction.

Further Evaluation

- More comprehensive evaluation of RegExLib patterns for performance degradation.
- Human subjects study measuring the readability and maintainability of normal forms.

Conclusions

Summary of Contributions

- Defined a formal hierarchy of 4 Normal Forms for PCRE balancing structural clarity (1NF–2NF) with runtime operational efficiency (3NF–4NF).
- Empirical evaluation on 36 RegExLib benchmarks demonstrated a **14% average length reduction** and **2.3x–3.5x average execution speedups**.

Thank You! Questions?

Contact: Kevin Angstadt (kangstadt@stlawu.edu)

Lead Author **Oleg Efimov** (oleg2003efimov@gmail.com)
is seeking a **Graduate RA / Ph.D. Position**!

Bonus / Backup Slides

Backup: Experimental Infrastructure and System Specs

Hardware & Operating System Platform

- **CPU**: 8-core Intel Xeon E5-1620 v4 CPU @ 3.50GHz.
- **Memory**: 64 GB RAM.
- **Operating System**: Ubuntu 22.04.5 LTS (Linux Kernel 5.15.0-161).

Software Runtimes & Execution Controls

- **Python Environment**: Python 3.14.3.
- **Regex Engines**: Standard library `re` vs. third-party `regex` library (v2026.2.28).
- **Corpus Generation**: 5.4 GB total input corpus (150 positive base strings via `rstr` + 30 mutated negative variants per string).
- **Execution Safeguard**: Hard **25-minute execution cutoff** per trial run to handle catastrophic backtracking stalls.

Backup: 1NF Lemmas (Disjointness and Kleene-Class Fusion)

Lemma 1 (Disjointness of Alternation)

Statement: Let $R = R_1 \mid \cdots \mid R_n$ be a PCRE alternation without backreferences or recursion. There exists an equivalent expression $R' = R'_1 \mid \cdots \mid R'_n$ such that $\mathcal{L}(R) = \mathcal{L}(R')$ and sub-languages are pairwise disjoint: $\forall i \neq j, \mathcal{L}(R'_i) \cap \mathcal{L}(R'_j) = \emptyset$.

Proof Idea: Construct partition $R'_1 = R_1$ and $R'_k = R_k \setminus \left(\bigcup_{j=1}^{k-1} R'_j\right)$. Closed under regular union/difference; preserves first-match policy while eliminating redundant backtracking paths.

Lemma 2 (Kleene-Class Conversion)

Statement: 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 Idea: Generalization of Kahrs & Runciman (2022) fusion. Optionality of every a_i guarantees $\varepsilon \in \mathcal{L}(R)$, collapsing inner choices into a single Kleene-$*$ character class.

Backup: 2NF Lemmas (Quantifier Subsumption)

Lemma 3 (Search Tree Branching Factor Reduction)

Statement: If a PCRE P is QS-free, the number of successful paths in the backtracking search tree for string w is $\leq$ that of its QS-afflicted counterpart.

Proof Idea: Adjacency subsumption ($A^\bullet B^\circ$) creates choice points for $s \in \mathcal{L}(A) \cap \mathcal{L}(B)$; enforcing minimum repetitions $A^n B^\circ$ eliminates ambiguity. Nested subsumption ($(AB^\circ)^\bullet$) collapses redundant outer iterations.

Lemma 4 (Quantifier Subsumption Language Preservation)

Statement: Let P be a PCRE with quantifier subsumption. The transformation to QS-free P' preserves language: $\mathcal{L}(P) = \mathcal{L}(P')$.

Proof Idea: *Adjacency*: Repetitions beyond minimum n matched by B° ($A^n A^{k-n} B^\circ \equiv A^n B^\circ$). *Nested*: $(B^\circ)^\bullet \equiv (B^\circ)^n$; extra outer iterations subsumed by inner closure B°.

Backup: 4NF Lemma (Possessive Safety and Language Preservation)

Lemma 5 (4NF Possessive Safety)

Statement: The transformation $\mathcal{T}_{\text{4NF}}$ preserves language $\mathcal{L}(P) = \mathcal{L}(P')$ provided that the atomized sub-expression is *possessive-safe*: greedy commitment of the prefix does not prevent reaching a valid match.

Proof Idea:

- **Soundness ($\mathcal{L}(P') \subseteq \mathcal{L}(P)$)**: Atomic constructs only restrict backtracking behavior; they never generate new strings.
- **Completeness ($\mathcal{L}(P) \subseteq \mathcal{L}(P')$)**: Possessive safety ensures greedy commitment does not preclude reaching valid matches.
- **Pruning Mechanism**: Engines evaluate paths greedy-first; atomization locks the path the engine prioritized, pruning unneeded failing branches.