Tool Track Papers of the 30th International Conference on Implementation and Application of Automata (CIAA 2026)

Taylor J. Smith (Ed.)

August 2026
External Technical Report
ISSN-0836-0227-
2026-651

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared August 1, 2026

Abstract

The 30th International Conference on Implementation and Application of Automata (CIAA 2026) was held in Kingston, Canada on August 5–8, 2026, and was organized by the School of Computing at Queen's University.

CIAA is an annual international conference concerning research on all aspects of implementation and application of automata and related structures, including theoretical aspects. Topics of interest include but are not limited to: complexity of automata operations, compilers, computer-aided verification, concurrency, data structure design for automata, data and image compression, design and architecture of automata software, digital libraries, DNA/molecular/membrane computing, document engineering, editors and environments, experimental studies and practical experiences, industrial applications, natural language processing, networking, new algorithms for manipulating automata, object-oriented modeling, pattern-matching, quantum computing, speech and speaker recognition, structured and semi-structured documents, symbolic manipulation environments for automata, teaching, text processing, techniques for graphical display of automata, VLSI, and viruses and related phenomena.

CIAA originated in the mid-1990s as the International Workshop on Implementing Automata (WIA). This workshop was held in London, Canada (1996 and 1997), Rouen, France (1998), and Potsdam, Germany (1999).

In 2000, WIA became CIAA, and past editions of the conference were held in London, Canada (2000), Pretoria, South Africa (2001), Tours, France (2002), Santa Barbara, USA (2003), Kingston, Canada (2004), Nice, France (2005), Taipei, Taiwan (2006), Prague, Czech Republic (2007), San Francisco, USA (2008), Sydney, Australia (2009), Winnipeg, Canada (2010), Blois, France (2011), Porto, Portugal (2012), Halifax, Canada (2013), Gießen, Germany (2014), Umeå, Sweden (2015), Seoul, South Korea (2016), Marne-la-Vallée, France (2017), Charlottetown, Canada (2018), Košice, Slovakia (2019), Loughborough, UK (2020), Bremen, Germany (2021), Rouen, France (2022), Famagusta, Cyprus (2023), Akita, Japan (2024), and Palermo, Italy (2025).

The planned 2020 edition of CIAA was cancelled due to the pandemic. The in-person arrangements for the 2021 edition of CIAA were also cancelled due to the pandemic, though the conference went ahead as a virtual meeting.

This technical report contains the three papers selected for presentation in a special tool track session at CIAA 2026, and complements the proceedings volume published in the Springer *Lecture Notes in Computer Science* series containing both the invited contributions and the 14 accepted papers presented at the conference. We thank each of the members of the program committee for their excellent work in producing the program, and we thank the additional reviewers for their helpful contributions.

Contents

Automated Verification of Student Automata Submissions Using AFCT 1
Jesse Burdick-Pless, Lucas Famous, Chang you Yu, Teddy FitzPatrick, Alexander Day

Normal Forms for Perl-Compatible Regular Expressions 17
Oleg Efimov, Tommy Tracy II, Whiting Zheng, Kevin Skadron, Kevin Angstadt

Compiling Rewrite Rules to Finite-State Transducers with the Worsening Trick 32
Mans Hulden, Michael Ginn

Author Index 49

Automated Verification of Student Automata Submissions Using AFCT

Jesse Burdick-Pless[1,2] Lucas Famous[1] Chang you Yu[1]
Teddy FitzPatrick[1] Alexander Day[1]

Abstract

The Automated Feedback for Computing Theory (AFCT) system (`https://www.cs.rit.edu/~afct/`) provides automated verification of automata submissions. Students develop and submit automata in response to instructor-developed problems. Students get immediate feedback on the correctness of their submission with a witness string (e.g. "010" SHOULD be accepted) as feedback for incorrect submissions. Immediate feedback and the ability to re-submit assignments multiple times support efficient learning and student satisfaction. Automated assignment verification also reduces instructor grading workload. AFCT has successfully been tested in classroom situations, most recently in the spring 2026 semester. Recent update efforts address support for work with generalized nondeterministic finite automata (GNFA), Turing Machine equivalence, as well as optical character recognition (OCR) for hand-drawn automata.

1 Introduction

In a computing (or computer science) theory course, students construct various automata with states and transitions, or grammars with rules, to recognize or generate specified languages. Computational models are essential building blocks in computing theory, but they can be challenging for students to fully understand. Grading instances of these models can be time-consuming for instructors, which can lead to delayed feedback for students. Much of computing theory builds directly on previous concepts. Thus, it is important that learners know in the moment whether or not a concept is understood before moving on to more complex ones.

JFLAP is a package of graphical tools which can be used as an aid in learning the basic concepts of formal languages and automata theory [9]. First developed in 1990 at Rensselaer Polytechnic Institute by students working under the direction of Susan Rodger, JFLAP moved to Duke University in 1994 when Professor Rodger accepted a position there. JFLAP has been continually updated by Dr. Roger's students through the last JFLAP release in 2018.

Similar to hand-drawn automata, traditional grading of JFLAP model constructions typically requires manual checking. This can be labor-intensive for instructors, and time delays can adversely affect student learning. To address this issue, AFCT was developed.

[1]Rochester Institute of Technology, Rochester, NY 14623, United States of America
[2]Pennsylvania State University, Hazleton, PA 18202, United States of America

The AFCT project builds on JFLAP by updating it and going beyond to check the equivalence of student submissions with an instructor's answer and give immediate feedback. For incorrect submissions, AFCT provides a "witness" string (a string on which the submission fails) as succinct feedback to the student [4]. AFCT is constructed of a client/editor for student use, a server for instructor use and assignment submission, and an evaluator for submission validation.

Immediate feedback with the ability to reformulate and resubmit over and over allows students to learn from their errors in real time, and gain confidence that they have fully grasped the material. Some students will resubmit even after getting the problem correct to experiment with other configurations, further cementing their understanding. This is in contrast to waiting days to a week (or more) for graded homework to be returned, by which time the instructor has inevitably moved on to other topics. Figure 1 shows an image of the AFCT client v1.6.7.

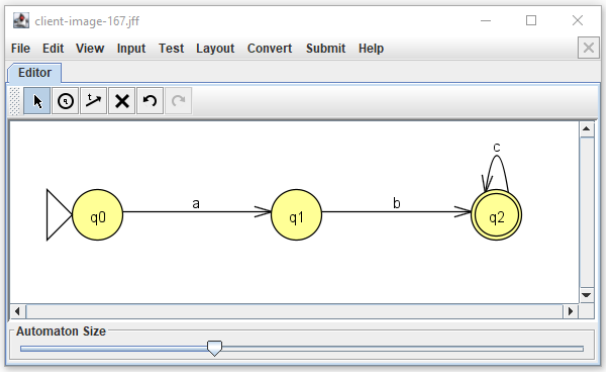


Figure 1: AFCT client v1.6.7

For deterministic and nondeterministic finite automata (DFAs and NFAs) and regular expressions, equivalence can be checked algorithmically in polynomial-time for DFAs. While the problem is PSPACE-complete for the other two models, exponential algorithms have proved sufficient for homework-level assignments. For models with higher complexity, such as context-free grammars (CFGs), equivalence checking is undecidable. To compensate, in accordance with Axelsson, Heljanko, and Lange [3] utilizing SAT solvers, the AFCT tool checks a substantial number of inputs in the alphabet and compares the results. If any input does not match, the tool determines that the answer is incorrect and can provide the current string as minimal feedback to the student. Validating all inputs up to and including length 15 was found to be sufficient for standard homework assignments.

The only related tool that we are aware of is the Automata Tutor (AT) [5, 1], which currently appears to be in an intermediate state of development. An advantage of AFCT is that it can be hosted by the course instructor, who has full control over the software configuration and deployment. The AFCT setup ensures that no third party has access to any student's private data since all submissions stay within the same university. This avoids any potential Family Educational Rights and Privacy Act (FERPA) violations.

Currently, AFCT is being updated to expand its functionality, including the addition of support for generalized nondeterministic finite automatons (GNFAs) and Turing machine

submissions. To improve student learning and be inclusive of students who prefer handwriting, work is underway to add a data detection model to convert handwritten automata to JFLAP file format.

We would plan to demonstrate AFCT installation, problem setup from the instructor's perspective for various automata types, and usage of the system from the students' perspective including witness string feedback.

2 Turing Machine Equivalence

2.1 Turing Machine Introduction and Notation

A *Turing machine* (TM) is a model with an infinite tape and a head that can read and write symbols and move left or right. Formally it is defined as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q is the set of states, Σ is the input alphabet not including $\sqcup$ (the blank tape character), Γ is the tape alphabet, δ is the transition function, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (where L means the tape head moves left, and R means the tape head moves right) , q_0 is the start state, q_{accept} is the set of accepting states, and q_{reject} is the set of rejecting states.

To simplify later explanations, we will be expanding this transition function to include a "stay" option, which results in $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$, where S means the tape head does not move. This change does not affect the computational power of the TM, as this behavior can be simulated in a standard TM through the use of extra intermediate states that move the tape head right then back left again which puts the tape head back to the same place it started.

The notation $\delta(q_i, a_k) = (q_j, b_k, D)$ where $D \in \{L, R, S\}$ means the TM starts in state q_i, reads a_k from the tape (at the current position of the tape head), moves to state q_j, writes b_k to the tape, and moves the tape head in direction D (left, right, or stay).

A Turing Machine *configuration* provides a "snapshot" of a Turing Machine at any point during computation. This snapshot consists of the current state, the contents of the tape, and the position of the tape head.

A TM may either act as a decider that gives a Boolean output based on whether the machine accepts or rejects, or as a transducer that produces an output String via the contents of the tape.

A *nondeterministic* Turing machine (NDTM) can have a choice of possible moves. In other words, when an NDTM is in a given state and reads a symbol, it may have a choice of which transition to take, and thus which state to move to next. To accommodate this behavior, the transition function is modified to be $\delta : Q \times \Gamma \to Pow(Q \times \Gamma \times \{L, R\})$, where Pow denotes the power set. For an NDTM, as long as there exists some valid sequence of states that ends in an accepting state, the input is accepted.

2.2 Equivalence Checking

As simply determining an arbitrary TM's output for a given input is an undecidable problem, it comes as no surprise that Turing machine equivalence is likewise undecidable as well. Turing machines are not guaranteed to halt in general, and there is no computable way to check if a given Turing machine will actually halt on a particular input. In an academic

setting however, students are typically given assignments where the Turing machines involved are decidable and will always halt. As a result, using brute force to compare student submissions to instructor solutions is viable.

To limit run time, we employ a Monte Carlo algorithm that probabilistically determines the correctness of a solution by testing a series of inputs and skipping iterations that are determined not to halt. Our implementation allows for equivalence to be judged based on whether inputs are correctly accepted/rejected (as with other automata types), or by comparing the tape once computation has finished.

Algorithm 1 TMEQ Algorithm

Require: Σ
  $Generate\_inputs(\Sigma)$
  while $Elapsed\_time < Overall\_max\_time$ **do**
    while $Input\_time < Input\_max\_time \cdot f(|input|)$ & $|Configs| > 0$ **do**
      $configs \leftarrow step\_Configs()$
      if Accept **then**
        return Accept
      end if
    end while
    if $Time\_expired$ **then**
      return Continue
    end if
    return Reject
  end while

Our algorithm 1 validates sequential input strings until it reaches a maximum allocated time. An input may not halt and may run until it reaches a maximum allotted time limit. Detected infinite loops are counted as rejections. During initial testing, we have had success with values $Overall\_max\_time = 5s$ and $Input\_max\_time = 0.01s$. The maximum input time was enough for moderately complex TMs to halt on most inputs, and the overall max time lets the algorithm try all inputs up to a sufficient size while not taking an unreasonable amount of time. The input time is scaled based on the input length, so that TMs can still run in a reasonable maximum time frame for both large and small inputs.

If both TMs return matching answers or if either fails to halt, then no information is gained. However, if the TMs return conflicting answers, the submission is guaranteed to be incorrect. Perfect accuracy cannot be achieved in this situation. For example, if a submission would only fail for very large inputs (i.e. larger than what can be tested in the allotted time frame), the answer would not be determined to be incorrect. Instead, submissions can be better thought of as definitely incorrect or probably correct. Submissions marked as incorrect are indisputably incorrect. However, other submissions can only be thought of as correct with high probability. Given the undecidability of TM equivalence, this is an expected and acceptable limitation.

2.2.1 Early stopping

Our test suites have shown that the early stopping condition of this algorithm almost instantly rejects TMs that have been manually altered to be incorrect. On matching TMs,

it can go through hundreds to thousands of example inputs, depending on $|Q|, |\delta|, |\Sigma|$, and $f(n)$.

If the submission is time-bound by $f(n)$, then incorrect submission witness strings will be of minimum size. This algorithm also checks inputs independently, so it can be efficiently run in parallel using a shared input queue. This was tested using two examples to recognize the language $L = \{a^m b^m\}$ - one on a 6 state TM directly replacing pairs of a and b with $\sqcup$, and one on a 5 state TM using a marking character. Averaging 3 attempts each, the singular program went through 293,614 halting strings, with 4 that failed to halt. Using 4 threads, the program went through 943,370 halting strings with 10 that failed to halt.

This represents a significant improvement, as the addition of multithreading more than tripled the number of strings the program was able to check in the same timeframe. For a two character alphabet, previously the program was only able to check strings with lengths up to and including length 18 as well as some length 19 strings, whereas with the addition of multithreading, the program was able to check all strings with length 1 to 19, and nearly ninety percent of strings of length 20.

Additionally, the software outputs the number of inputs that did and did not halt on either TM. This can act as an indication to students of incorrect submissions that accidentally include an infinite loop. Loops that revisit previous configurations can be detected by storing each encountered configuration in a set and checking for repeats. Infinite reflexive loops can also be efficiently identified by detecting transitions that leave the machine in an equivalent configuration, such as when the machine remains in the same state, preserves the tape contents, and either does not move the tape head or moves only within blank boundary regions.Implementing this did not significantly slow down TM equivalence checking. It accurately detected and ended testing on TMs with recursive loops.

2.2.2 Input generation

The input alphabet can be inferred from the union of the tape alphabets of both TMs, excluding the blank symbol, as shown in Equation 1.

$$\Sigma \subseteq (\Gamma_{TM1} \cup \Gamma_{TM2}) - \{\sqcup\} \tag{1}$$

TMs can create a finite number of their own symbols that are unique to the input alphabet, which are generally used in textbook problems to mark tape locations. These cannot automatically be inferred, so in the AFCT client comparison, users are shown the input alphabet list to correct, and we plan to implement the ability for instructors to set a specific alphabet for a given problem in the AFCT dashboard. Figure 2 demonstrates this ambiguity with a TM in AFCT, where if $\Sigma = \{x, y\}$, it accepts the language $L = \{x^m y^m \mid m >= 1\}$, and if $\Sigma = \{x, y, z\}$, it accepts the language $L = \{xz^* x^{m-1} z^* y^m z^* \mid m >= 1\}$. Once Σ is detected, all possible permutations of characters are generated to a given length that is extended when depleted.

2.2.3 Multitape compatibility

In the case of multitape TMs, the user is prompted to select one tape number as the input and one as the output. The algorithm starts with the input on the input tape and checks equivalence on the output tape after acceptance for multitape transducers. This only works for two TMs that have the same number of tapes. As TMs of differing tape sizes are

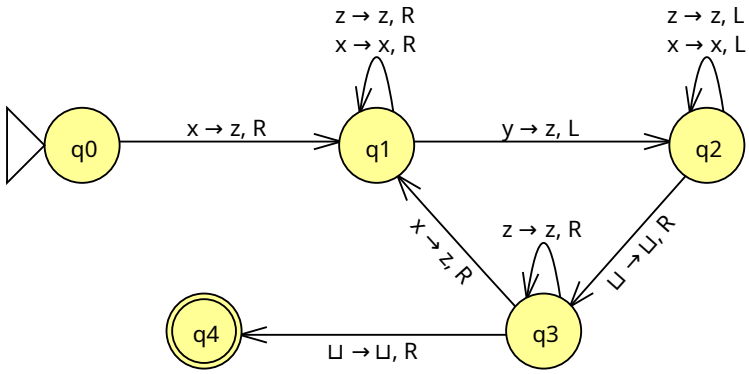


Figure 2: Σ detection problem

computationally equivalent, if users want to compare TMs with differing tape sizes, adding $|k_{TM_1} - k_{TM_2}|$ blank tapes between the input and output tape of the smaller TM will convert it to match the output scheme.

2.2.4 Nondeterministic compatibility

Boolean outputs for nondeterministic TM deciders can be compared by stepping through configurations as normal and returning acceptance for the first accepting configuration or returning rejection after reaching an empty configuration list. When comparing tape outputs for nondeterministic TM transducers, the TM configuration list must always be exhausted, and all accepting configuration tape values are added to a set to be compared for intersection.

3 Generalized nondeterministic finite automata (GNFA)

AFCT provides students with an editor to construct various types of state machines (e.g., NFAs, DFAs, TMs). As students progress in their studies of automata theory, they arrive at the important theme of equivalence in computational power between different state machines and formal expressions. The next iteration of AFCT will solidify this theme by introducing support for generalized nondeterministic finite automata (GNFA) construction, editing, and input validation.

The first class of languages that students usually encounter is the class of regular languages, which is why JFLAP includes support for DFAs, NFAs, and regular expressions.

When showing the equivalence of DFAs, NFAs, and regular expressions, GNFAs are typically used as an intermediary to assist in converting DFAs to regular expressions. This is accomplished by first converting a given DFA into a GNFA, then removing states from the GNFA until only two remain, at which point the single remaining transition contains the desired regular expression.

The addition of a capable GNFA editor to AFCT will provide a powerful pedagogical tool, allowing students to complete the conversion between finite-state machines and regular expressions through GNFA reduction.

Formally, a GNFA is defined as a 5-tuple (Q, Σ, δ, q_{start}, q_{accept}), where Q is a finite set of states, Σ is a finite input alphabet, δ is a transition function, defined as $\delta : (Q -$

$q_{accept}) \times (Q - q_{start}) \rightarrow \mathcal{R}$, where $\mathcal{R}$ is the set of all regular expressions over Σ, q_{start} is the start state, which has no incoming edges, and q_{accept} is the accept state, which has no outgoing edges. As well, GNFAs have some special conditions. Firstly, the starting state has outgoing transitions to every other state. But has no transitions into it from other states. Second, there is only one final state, which cannot be the start state. This final state has transitions into it from every other state, but has no outgoing transitions. Finally, each state (other than the start state and final state) has a single transition to every other state (including itself).

3.1 Supported Operators

For regular expressions and GNFA transitions, the following operators are supported in AFCT:

( )	Parenthesis, used for explicitly defining order of operations
*	Kleene star
+	Union
!	Represents the empty string

Note that concatenation is implicit, and thus does not have a defined symbol.

3.2 Implementation

To simulate DFAs and NFAs, JFLAP simulates each configuration step by iterating transitions and checking if the transition's string matches the remaining input prefix of the same length. This cannot be used for or GNFAs because their transitions are regular expressions, which do not exclusively accept fixed-length prefixes (e.g. navigating input "$aaab$" on a^*). We modified the DFA/NFA design to simulate GNFAs by attempting to match every prefix of the remaining input string. We chose this approach instead of converting the GNFA to a regular expression for testing because it allowed us to show a traceback through the GNFA for accepting inputs.

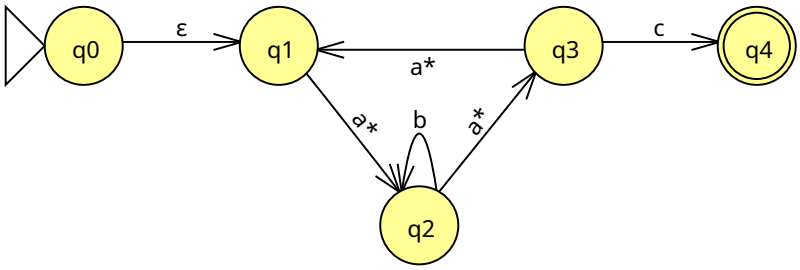


Figure 3: GNFA possible loop

One of the problems we encountered in simulating GNFAs was detecting infinite loops on certain inputs, as shown in Figure 3. Our solution was to apply epsilon closures, treating x^* as $\epsilon|x^+$, to find the set of states reachable from the current state using a nonempty

transition. This guarantees that each time a transition is navigated, the remaining input string is truncated, avoiding cyclic epsilon transitions that loop indefinitely.

4 Automata OCR

Students who use handwriting may achieve better learning outcomes than those who use only computer tools. In tasks that require a high level of processing, handwriting may be advantageous over computers and may allow for better recall of information [2]. In math tutoring systems, it has been shown that handwriting can be more efficient and can lead to better results when the computer input method is less than ideal [6].

For a subject as abstract as computer science theory, students may benefit from the option to verify the correctness of hand-drawn solutions. Supporting handwritten submissions would increase accessibility for those who have limited computer access or for those that simply prefer handwriting. For instructors, the existence of an OCR model would allow exam answers and paper homework to be scanned into JFLAP to expedite the grading process.

To expand our immediate feedback mechanism to handwriting contexts, we plan to introduce a general automaton OCR model. We have developed a data-generation-focused variant of JFLAP that allows us to generate synthetic automata OCR examples and have fine-tuned a specialized OCR Vision-Language Model (VLM) using the synthetic dataset to obtain the final model prototype.

4.1 Model choice

DeepSeek-OCR 2 [12] was chosen as the base model due to its novel DeepEncoder architecture. After tokenizing the input image using a segment anything model (SAM) and convolutional pooling, the DeepEncoder architecture employs a small 500M parameter language model as an encoder. The vision tokens are augmented with new "causal flow query" tokens that are appended to the end of the sequence. As their name suggests, the causal flow tokens only perform attention on themselves and previous tokens. The addition of "causal flow" tokens preserves global attention on all the vision tokens and introduces causal attention for vision token reordering.

Figure 4 gives a visual representation of what the attention mask would look like with causal flow tokens. With the introduction of a learnable query to the vision tokens, a two-stage causal reasoning pipeline is established. Vision tokens are first re-ordered by the causal flow tokens, then the 3B MoE (mixture-of-experts) decoder applies its own causal attention to the re-ordered tokens. This double causal approach proves to be a promising solution in converting 2D problems into a form more suitable for the unidirectional attention patterns used by current LLMs [10, 8, 11].

4.2 Synthetic data generation

To obtain the data necessary for fine-tuning, we expanded the functionalities of JFLAP to include automatic finite state machine generation capabilities. The JFLAP backend was augmented to autonomously generate new automata in a structured internal representation. To get a visual representation, we leveraged the existing ability of JFLAP to render state

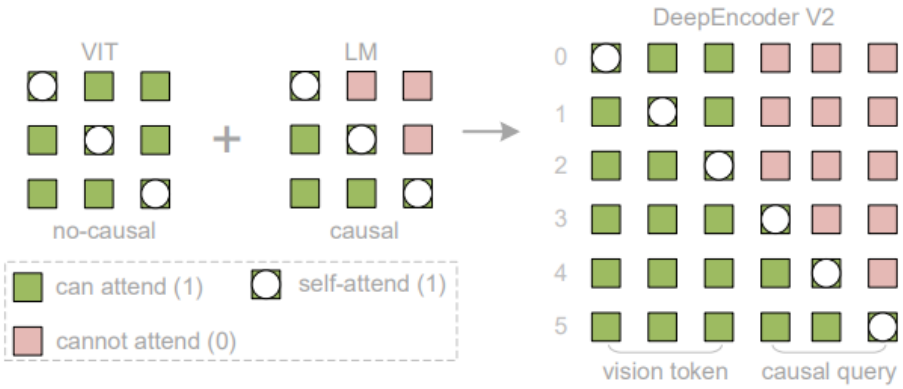


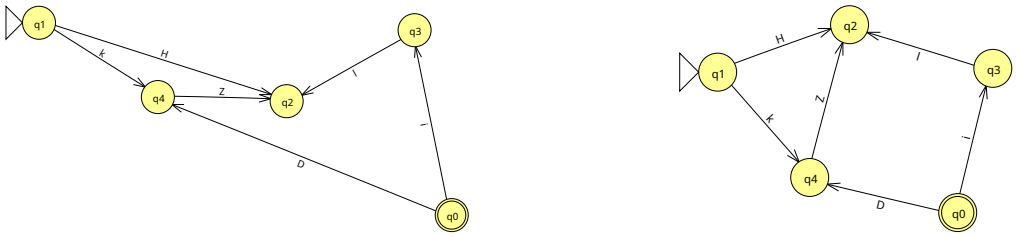
Figure 4: Attention Mask for DeepEncoder V2 [12]

diagrams based on the aforementioned internal representation. For the initial implementation of the automaton OCR model, we chose to focus on non-deterministic finite automatons (NFAs). We define an NFA as: $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function, q_0 is the initial state, and F is the final state. We focused on NFAs due to their conceptual simplicity. Specifically, the non-deterministic nature of NFAs allows for minimal validation of the automatically generated automata.

The data generation process for creating NFAs can be thought of as an initialization, layout, and saving loop. In the initialization phase, states from $S_{min} = 3$ to $S_{max} = 11$ are added. An initial state and up to three final states are chosen at random. Transitions are then added, with each state having an equal chance to create a transition between itself and any of the other states. We found that a transition rate of $T_{rate} = 0.33$ produced authentic looking examples. It is important to note that at this stage, states are simply placed right next to each other without regard to the overall visual appearance of the automaton. After state and transition creation, heuristics are applied to increase dataset quality. Orphaned states (states in which there are no transitions to or from the state) are removed. Image manipulations such as panning and rotation are applied to ensure translational and rotational invariance. Additional logic is added to ensure the automaton is fully in frame before screen capture happens.

After the heuristics are applied, the layout process begins. From a visual standpoint, a state diagram can be thought of as a graph $G = (V, E)$, where V is a set of vertices corresponding to the states Q, and E is a set of two-element subsets $(u, v) \subseteq V$ representing the transitions. For the purposes of performing the layout operation, we consider an edge (u, v) to be in the set of edges if $(u, v) \in \delta$ or $(v, u) \in \delta$. After obtaining an undirected graph representation, a variety of graph layout algorithms can be applied. We randomly apply one of the following in the subset of built-in JFLAP algorithms: *circle*, *two-circle*, *random*, *random* then *graph embedder* (*GEM*), *tree* then *GEM*. Using multiple of these layouts covers the various ways students could construct automata for diverse problems. The circle layout creates circles of connected components while trying to minimize edges. The two circle algorithm creates an inner circle of high degree and an outer circle of low degree. The tree layout sorts vertices into layers, then places vertices in each layer to minimize edge

crossings. GEM is a randomized adaptive algorithm using heuristics to quickly generate a visually appealing graph [7].



(a) Before (b) After

Figure 5: Before and after applying the GEM layout algorithm

Once the layout process concludes, the visual representation of the automaton is captured, and the associated XML file is saved to disk. The cycle will then repeat n times, where n is the number of examples desired by the user.

4.3 Training and Evaluation

To fine-tune DeepSeek OCR2 we worked with the synthetic data generated from JFLAP. Fine-tuning was performed on one NVIDIA A100 GPU. Initial results are promising. Using only 10,000 synthetic examples, the model was able to recognize states, their relative locations, and whether their status is that of an initial or final state. The model does relatively well identifying transitions, making only minor mistakes depending on the overall complexity of the automaton. A potential future improvement would be creating a custom loss function that ignores the order in which the transitions are labeled and instead compares if a transition outputted by the model exists somewhere in the ground truth example. Results so far are promising, and with additional data cleaning and training improvements, we can likely get the overall model accuracy to be above 95%.

5 Results

The usage and benefits of AFCT were studied at the Rochester Institute of Technology in Fall 2019 and Spring 2021. Each term, students in one section of the Introduction to Computer Science Theory course used AFCT for problems regarding DFAs, NFAs, RegExs, CFGs, and PDAs. Comparison data was collected from another section of the same class that did not use AFCT, but had the same experienced instructor and the same assignments. The students using AFCT performed better than the comparison group. Students who used AFCT reported that it helped them not only to solve homework problems, but also to better understand the underlying theory. The authors noted that it is not possible to random-walk to a solution, and students mentioned in their survey comments that they appreciated getting the gentle nudge of a witness string telling them where to look for a problem with their current submission [4].

More recently in the spring 2026 semester, Jesse Burdick-Pless taught a computer science theory class using AFCT as an important tool for student homework and exam submissions.

AFCT data was gathered over six assignments and two exams, for a total of 46 individual problems 6.

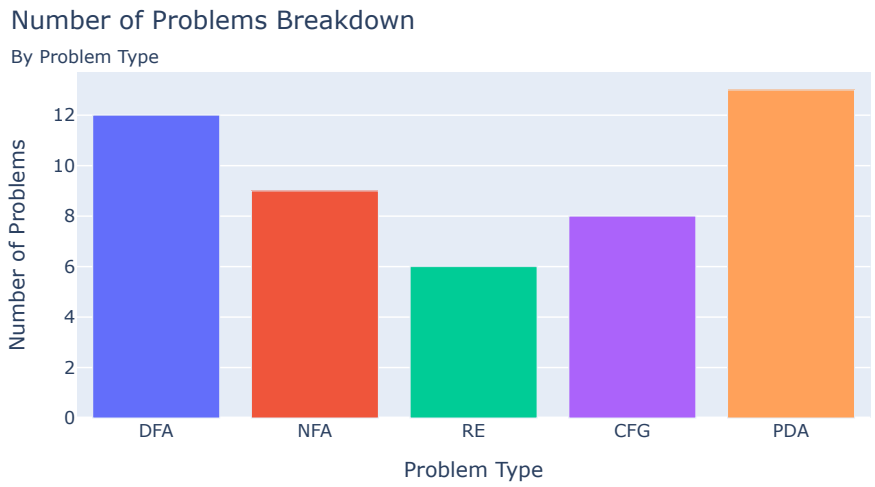


Figure 6: Number of Problems Breakdown (By Problem Type)

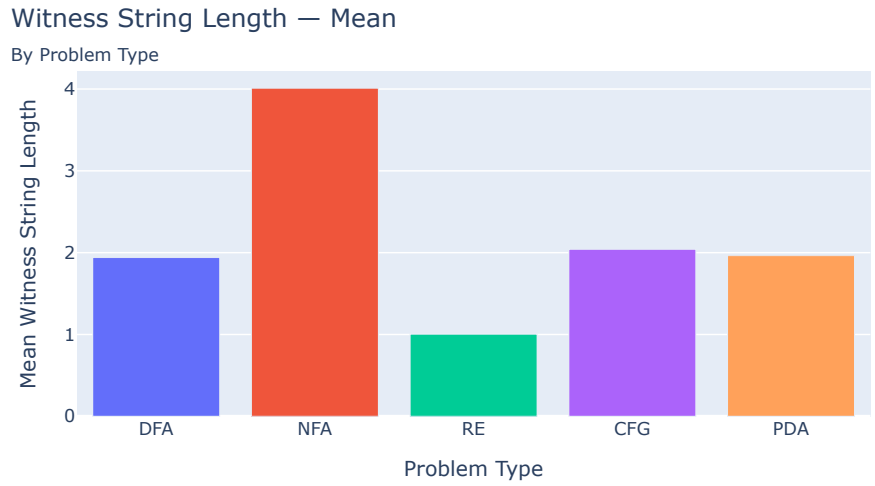


Figure 7: Witness String Length – Mean (By Problem Type)

In Figure 7, the mean witness string is shown to be relatively short. Witness string messages were of the form: "010" SHOULD be accepted. In this case the witness string "010" is of length three. Witness string length averaged from two to four depending on problem type, but with some outliers in the teens (Figure 8). Regular expressions had the shortest mean witness string length, with the empty string being a frequent point of error. NFAs generally had the longest witness strings, though REs had some long outliers.

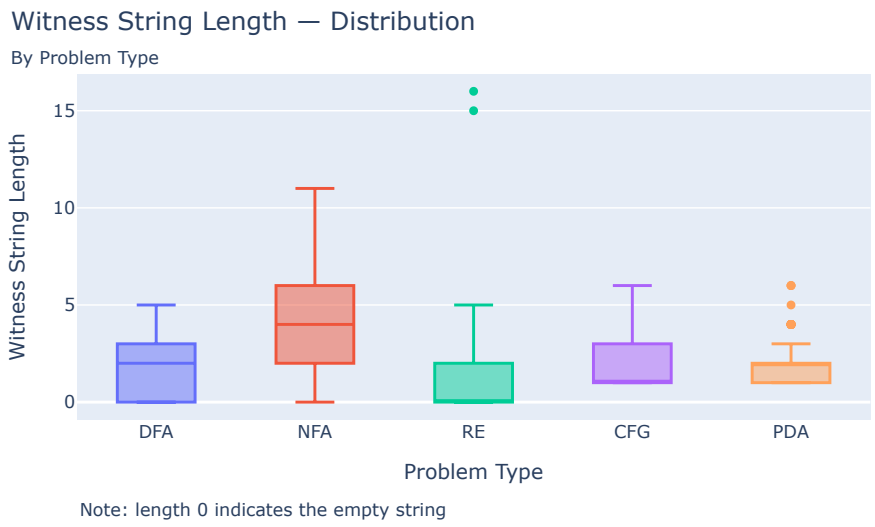


Figure 8: Witness String Length – Distribution (By Problem Type)

Figure 9: First Attempt Success Rate – Mean (By Assignment)

With the exception of homework 5, less than half of students submitted correct submissions on their first attempt 9. On average, across all assignments, 41.47% of students submitted correct submissions on their first try. Despite this, in almost all cases, students submitted correct solutions to the problems by the deadline, indicating determination in

Assignments Included	Running Mean % Correct on First Attempt
Homework 2	37.50%
Homeworks 2 & 3	35.61%
Homeworks 2 - 4	36.07%
Homeworks 2 - 5	40.62%
Homeworks 2 - 5, Midterm	39.69%
Homeworks 2 - 5, Midterm, Homework 6	38.74%
Homeworks 2 - 5, Midterm, Homeworks 6 & 9	39.58%
Homeworks 2 - 5, Midterm, Homeworks 6 & 9, Final Exam	41.47%

Figure 10: Running First Attempt Success Rate By Assignments Included

learning by adjusting their automata until they achieved a correct solution. There was a weak trend toward more students getting submissions correct on the first attempt 10. However the failure of the majority of the students to reach that level underscores the educational importance of the "learning by doing" afforded by allowing multiple AFCT attempts as students gain mastery in their own individual time.

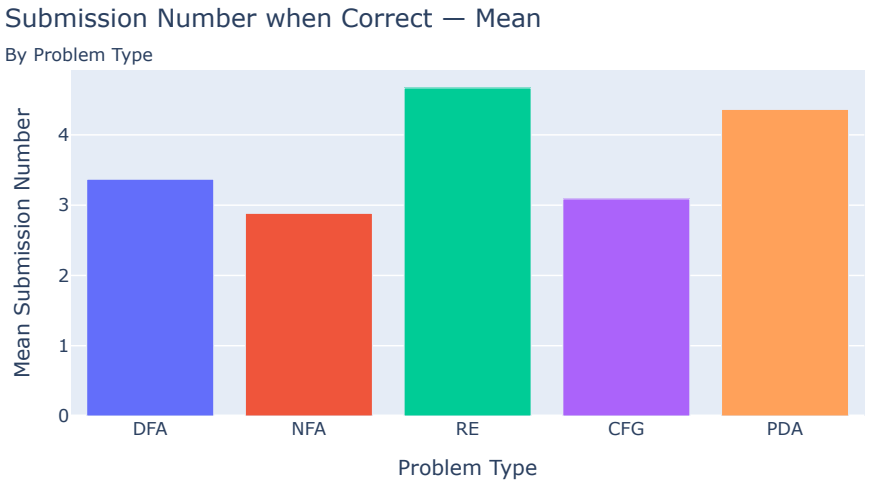


Figure 11: Submission Number when Correct – Mean (By Problem Type)

On average it took approximately three to four and a half submissions, depending on the type of problem, for students to achieve the correct answer 11. Significantly though, perhaps a quarter of the class needed something closer to twenty submissions to get to the correct answer and a few students had submissions into the forties 12. This suggests that while many students are able to master the material with only a small number of attempts, other

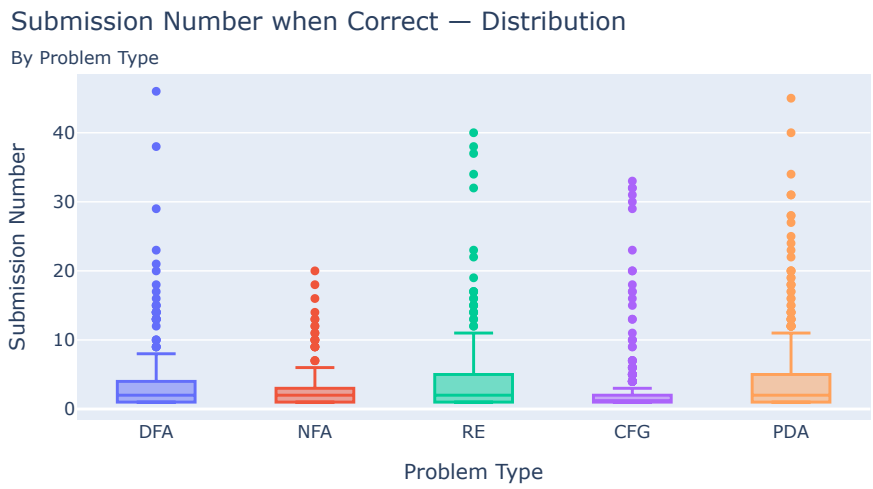


Figure 12: Submission Number when Correct – Distribution (By Problem Type)

students benefit from a much larger number of attempts. Given traditional assignments, these students could have been left behind. With the benefit of unlimited attempts, such students were able to master the material and keep up with their classwork where they might have struggled unsuccessfully in a more traditional single-submission homework setting.

Unsolicited written feedback from a spring 2026 student illuminates the student learning experience using AFCT: "*AFCT allowed as many attempts as we wished which actually allowed me to experiment with the problems and figure out what mistakes I was doing to eventually figure out the right answer. If the tests and assignments had been one attempt only, I'm sure they would have been near impossible! I think I speak for the whole class too when I say this but thank you for allowing us the flexibility to explore with the problems, which if anything deepened my understanding of what we were learning.*"

6 Conclusion

Previously, AFCT demonstrated superior learning when students have the ability to revise their automata assignments in response to minimal feedback. In our most recent in-class testing of AFCT, we have shown equivalent learning outcomes for students who grasp concepts quickly with those that need more time and multiple attempts to fully grasp the material.

We are in the process of improving AFCT and extending its pedagogical scope by enabling equivalence checking of TMs and construction of GNFAs. Training the OCR model has shown early success in creating JFLAP files from hand-drawn pictures of NFAs.

6.1 Future work

Currently, TM equivalence has only been developed for the AFCT client. However, it will also need to be implemented server-side for student submission verification. GNFAs have received support in AFCT for their construction, but not yet for equivalence checking. We hope to implement this by converting GNFAs to their equivalent regular expressions and checking equivalence using the existing regular expression validation code.

The automata OCR model currently shows success in detecting structure in images of JFLAP images to create a JFLAP file. To detect handwritten automata, these images are put through a filter to convert the digital model to resemble imperfect handwritten drawings. The current implementation only detects DFAs and NFAs, but could in the future be amended to work with PDAs or TMs based on special characters in the detected transition text, (e.g. a transition of "$0 \to x, R$" could be detected and converted to a TM. Likewise a transition of "$a, \epsilon \to a$" could be detected and converted to a PDA).

Acknowledgements. This undergraduate research is funded by NSF grants 2439323 and 2439326 (LF, CYY, TF, and AD are current undergraduate students; JBP graduated in 2025). We would like to thank Professors Ivona Bezáková and Edith Hemaspaandra for their guidance throughout this project. We would also like to acknowledge Professor Jeffrey Chiampi and students, Edwin Thomas Kimsal, Andrew Michael Sutton, and Adam Manowski from Pennsylvania State University, Hazleton for developing the AFCT web-based dashboard.

References

[1] Automata tutor: ATv3. Website. `https://automata-tutor.live-lab.fi.muni.cz/`.

[2] Estíbaliz Aragón-Mendizábal, Cándida Delgado-Casas, José I. Navarro-Guzmán, Inmaculada Menacho-Jiménez, and Manuel F. Romero-Oliva. A comparative study of handwriting and computer typing in note-taking by university students. *Comunicar*, 24(48):101–107, 2016. `https://doi.org/10.3916/C48-2016-10`.

[3] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental SAT solver. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 410–422, Berlin, Heidelberg, 2008. Springer. `https://doi.org/10.1007/978-3-540-70583-3_34`.

[4] Ivona Bezáková, Kimberly Fluet, Edith Hemaspaandra, Hannah Miller, and David E. Narváez. Effective succinct feedback for intro CS theory: A JFLAP extension. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education, Volume 1*, SIGCSE 2022, pages 976–982, New York, NY, USA, 2022. Association for Computing Machinery. `https://doi.org/10.1145/3478431.3499416`.

[5] Loris D'Antoni, Martin Helfrich, Jan Křetínský, Emanuel Ramneantu, and Maximilian Weininger. Automata tutor: Original website. Website. `https://automata-tutor.model.in.tum.de/`.

[6] Felipe de Morais and Patrícia A. Jaques. Does handwriting impact learning on math tutoring systems? *Informatics in Education*, 21(1):55–90, 2022. https://doi.org/10.15388/infedu.2022.03.

[7] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In Roberto Tamassia and Ioannis G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 388–403, Berlin, Heidelberg, 1995. Springer. https://doi.org/10.1007/3-540-58950-3_393.

[8] Gemma Team. Gemma: Open models based on gemini research and technology, 2024. https://arxiv.org/abs/2403.08295.

[9] Susan H. Rodger and Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Sudbury, MA, 2006. https://www.jflap.org/jflapbook/.

[10] Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019. https://arxiv.org/abs/1911.02150.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[12] Haoran Wei, Yaofeng Sun, and Yukun Li. DeepSeek-OCR 2: Visual causal flow, 2026. https://arxiv.org/abs/2601.20552.

Normal Forms for Perl-Compatible Regular Expressions

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

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.

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 [27, 23]. 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 [24, 14]. 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

[1]St. Lawrence University, Canton, NY 13617, United States of America
[2]University of Virginia, Charlottesville, VA 22903, United States of America

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 [30, 2, 29, 7]. In the worst case, a carefully crafted input can stall a PCRE engine, creating

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

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.

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

```
1  (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

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

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*`*).*

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

```
1  (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]*)+` → `[ab][a-d]*` *and* `([ab][a-d]*){2,5}` → `([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, $\mathcal{T}_{2\text{NF}}(R_{1\text{NF}})$, 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

```
1  (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

```
1  (?: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

```
1  (?: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

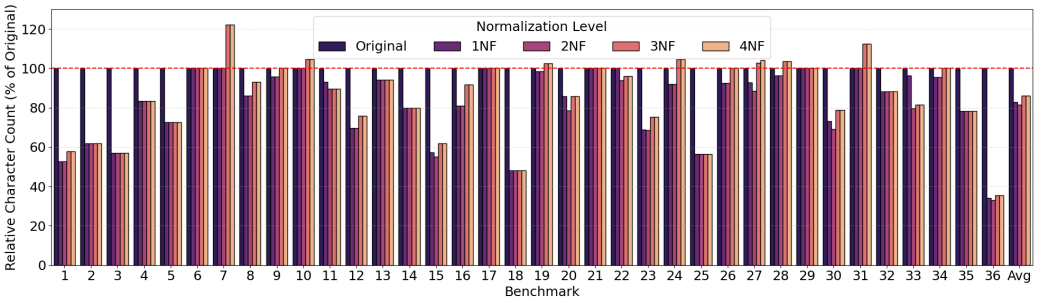


Figure 1: Relative character counts of PCRE across Normal Forms. A lower value represents a reduction in the length of the regex. In most cases, the character count decreases with average normalized expressions 86% the length of the original.

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.

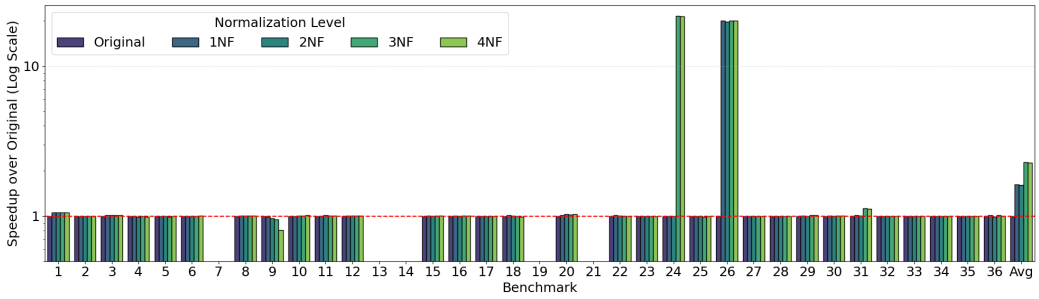
5 Evaluation

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?

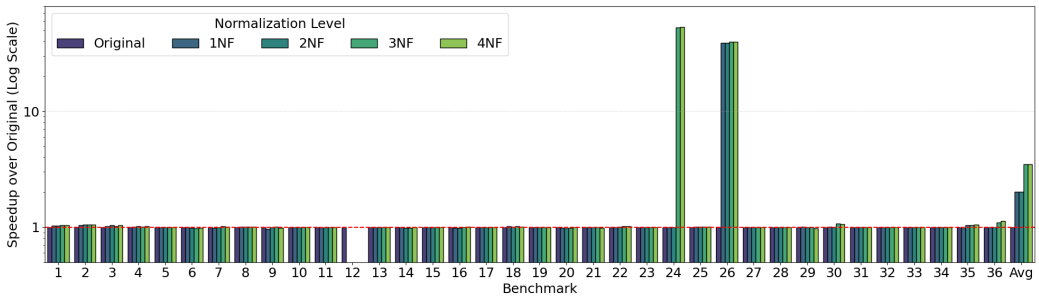
5.1 Normalization and Readability

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

[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.



(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.

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 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] Matthew Barnett. regex: Alternative regular expression module, to replace re. `https://pypi.org/project/regex/`. Accessed March 26, 2026.

[2] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture*

for Networking and Communications Systems, ANCS '07, pages 145–154, New York, NY, USA, 2007. Association for Computing Machinery. https://doi.org/10.1145/1323548.1323573.

[3] Anne Brüggemann-Klein. Regular expressions into finite automata. In Imre Simon, editor, *LATIN '92*, volume 583 of *Lecture Notes in Computer Science*, pages 87–98, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0023820.

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

[5] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. Exploring regular expression comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pages 405–416. IEEE Press, 2017. https://doi.org/10.1109/ASE.2017.8115653.

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

[7] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 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*, ESEC/FSE 2018, pages 246–256, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3236024.3236027.

[8] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 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)*, pages 343–359, Baltimore, MD, aug 2018. USENIX Association. https://www.usenix.org/conference/usenixsecurity18/presentation/davis.

[9] Google. Re2. https://github.com/google/re2. Accessed March 26, 2026.

[10] Hermann Gruber and Stefan Gulan. Simplifying regular expressions: a quantitative perspective. In *Proceedings of the 4th International Conference on Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 285–296, Berlin, Heidelberg, 2010. Springer-Verlag. https://doi.org/10.1007/978-3-642-13089-2_24.

[11] Philip Hazel. PCRE - Perl compatible regular expressions, 2026. https://www.pcre.org. Accessed March 17, 2026.

[12] Intel. Hyperscan, 2023. https://github.com/intel/hyperscan. Accessed March 26, 2026.

[13] Stefan Kahrs and Colin Runciman. Simplifying regular expressions further. *Journal of Symbolic Computation*, 109:124–143, 2022. https://doi.org/10.1016/j.jsc.2021.08.003.

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

[15] Su-Hyeon Kim, Hyeonseung Im, and Sang-Ki Ko. Efficient enumeration of regular expressions for faster regular expression synthesis. In Sebastian Maneth, editor, *Implementation and Application of Automata*, volume 12803 of *Lecture Notes in Computer Science*, pages 65–76, Cham, 2021. Springer International Publishing. https://doi.org/10.1007/978-3-030-79121-6_6.

[16] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing Chi Cheung. Redoshunter: A combined static and dynamic approach for regular expression dos detection. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3847–3864. USENIX Association, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yeting.

[17] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. RegexScalpel: Regular expression denial of service (ReDoS) defense by Localize-and-Fix. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4183–4200, Boston, MA, aug 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/li-yeting.

[18] Yichen Liu, Berk Çakar, Aman Agrawal, Minseok Seo, James C. Davis, and Dongyoon Lee. Regular expression denial of service induced by backreferences. *arXiv preprint arXiv:2602.21459*, 2026. https://doi.org/10.48550/arXiv.2602.21459.

[19] Achmad Maududie, Windi Eka Yulia Retnani, and Muhamat Abdul Rohim. An approach of web scraping on news website based on regular expression. In *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)*, pages 203–207, 2018. https://doi.org/10.1109/EIConCIT.2018.8878550.

[20] Brendan McCollam. rstr = random strings in python. https://pypi.org/project/rstr/. Accessed March 26, 2026.

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

[22] Taisei Nogami and Tachio Terauchi. On the Expressive Power of Regular Expressions with Backreferences. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, volume 272 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 71:1–71:15, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.MFCS.2023.71.

[23] Olaperi Yeside Okuboyejo, Sigrid Ewert, and Ian Sanders. Goofs in the class: Students' errors and misconceptions when learning regular expressions. In George Wells, Monelo Nxozi, and Bobby Tait, editors, *ICT Education*, Communications in Computer and Information Science, pages 57–71, Cham, 2021. Springer International Publishing. https://doi.org/10.1007/978-3-030-92858-2_4.

[24] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 7 edition, 2020. https://www.db-book.com/.

[25] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2 edition, 2006.

[26] Steven Smith. RegExLib.com: Regular expression library, 2026. http://regexlib.com. Accessed March 26, 2026.

[27] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM. https://doi.org/10.1145/2318202.2318207.

[28] Sicheol Sung, Hyunjoon Cheon, and Yo-Sub Han. How to settle the ReDoS problem: Back to the classical automata theory. In *Implementation and Application of Automata: 26th International Conference, CIAA 2022*, volume 13266 of *Lecture Notes in Computer Science*, pages 34–49, Cham, 2022. Springer. https://doi.org/10.1007/978-3-031-07469-1_3.

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

[30] Yi-Hua Yang and Viktor Prasanna. 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] Ling Zhang, Shaleen Deep, Jignesh M. Patel, and Karthikeyan Sankaralingam. Regular expression indexing for log analysis. *Proceedings of the ACM on Management of Data*, 3(6):355:1–355:29, dec 2025. https://doi.org/10.1145/3769820.

Compiling Rewrite Rules to Finite-State Transducers with the Worsening Trick

Mans Hulden[1] Michael Ginn[2]

Abstract

Finite-state transducers (FSTs) are essential for modeling string rewriting in computational linguistics and natural language processing (NLP), particularly for phonological and morphological rewrite rules. Compiling general rewrite rules of the form $A \rightarrow B / L \_ R$, where A, B, L, and R are arbitrary regular languages, is complex due to overlapping matches and context constraints. Traditional methods, such as those by Kaplan and Kay or Karttunen, rely on intricate transducer compositions with auxiliary markers. This paper presents a compact compilation scheme based on the "worsening trick" [6, 5]: generate all legal rewrite candidates, then filter candidates that are worse than another candidate for the same input. Implemented as the built-in rewrite compiler in PyFoma [12], the construction supports multiple contexts, arbitrary transductions, markup, directed rewriting, weights, and parallel rewriting. The resulting formulas are short and uniform, and where semantics coincide, they reproduce the same rule transducers as earlier approaches while remaining easier to extend. The implementation has been validated against *foma* [10] on both a substantial collection of rewrite grammars and an automated regression suite covering the major rewrite modalities, with the resulting transducers matching exactly apart from state numbering.

1 Introduction

String rewriting is fundamental to computational linguistics, particularly in phonology and morphology, where rewrite rules transform strings based on contextual constraints [3, 13]. A general rewrite rule, written as $A \rightarrow B / L \_ R$, replaces strings from a regular language A with B in the context $L \_ R$, where L and R are also regular languages, while copying unmatched input substrings to the output. Compiling such rules into finite-state transducers (FSTs) enables efficient processing but is challenging due to overlapping matches and multiple contexts, especially when A, B, L, and R are arbitrary regular languages [14]. The difficulty lies in the fact that material targeted for rewriting can simultaneously serve as context for another site of rule application [16].

We present a compilation method based on the "worsening trick" [6, 5]. We first generate all candidates that mark possible rewrite sites, then restrict those candidates to the legal contexts, and finally remove every candidate that can be made strictly worse by deleting or shifting some rewrite marking. The same three-stage pattern handles obligatory rewriting,

[1]New College of Florida, Sarasota, FL 34243, United States of America
[2]University of Colorado, Boulder, CO 80309, United States of America

leftmost/longest/shortest preferences, arbitrary transductions, markup, parallel rules, and weights.[1]

Our contribution is a formulation in which the compilation itself can be stated with substantially shorter and more uniform formulas than in marker-heavy classical presentations, while still matching the established semantics. The paper is organized as follows: Section 5 states the rule semantics assumed throughout, Section 6 reviews worsening, Section 7 gives the construction, and Section 8 concludes. A readable reference implementation in PyFoma, closely following the constructions in the paper, is given in the appendix (A).

2 Related Work

Foundational work by Kaplan and Kay [14] introduced a method for compiling rewrite rules by inserting auxiliary markers, constraining their distribution, performing the relevant replacements, and finally removing the markers again. Subsequent approaches by Karttunen [16, 17], Kempe and Karttunen [18], and Mohri and Sproat [19] refined this general marker-based strategy, incorporating directed rewriting, parallel rewriting, and weights. The Pynini treatment described by Gorman and Sproat [7] follows this same general line, presenting rewrite compilation as a composition of several auxiliary transducers with marker symbols and boundary-handling machinery.

A different attempt to make rewrite compilation conceptually clearer is *foma*'s multitape treatment [9], where an overgenerating multitape automaton is first constructed and then filtered so that only licensed rewrites remain before projection to an ordinary two-tape transducer. This idea also underlies other practical systems such as Kleene [1]. Efforts by Vaillette [21] and Hulden [11] use predicate logic to simplify correctness arguments, but the resulting rewrite constructions still require substantial auxiliary machinery.

The worsening trick itself has already been used for regular sets and transductions [6, 5]. Gerdemann [4] sketches a declarative rewrite implementation based on worsening using star-marked candidates and a flattening step. In contrast, our contribution is a complete, practical compilation scheme for the classical rule format $A \to B / L \_ R$ that operates entirely within ordinary regular languages and relations. The construction factors cleanly into three transparent steps: (i) generate all candidate rewrites by bracketing possible transduction sites in T_{base}, (ii) filter out candidates containing rewrite sites in illicit contexts using the dotted-position restriction technique of Yli-Jyrä and Koskenniemi, and (iii) retain only maximal candidates under a worsening relation that encodes obligatory application and directionality preferences. Where the semantics coincide, the resulting transducers are identical (modulo state renumbering) to those produced by the established *foma* compiler, yet the formulas are substantially shorter and more uniform than both classical marker-based constructions and multitape approaches, and significantly easier to extend to weights, parallel rules, markup, epsilon rules, and output-side context checking.

[1] A JavaScript port of PyFoma is available for interactive experimentation in the browser at `https://fomafst.github.io/pyfomajs/`. It implements the rewrite compiler described here and can visualize the resulting machines. PyFoma is available at `https://github.com/mhulden/pyfoma` and `https://pypi.org/project/pyfoma/`.

3 Implementation and Validation

Our construction is implemented in PyFoma, and we validated it against the C-based *foma* rule compiler both on a collection of 31 grammars comprising 427 rules[2] and on an automated regression suite covering the major rewrite modalities shared by the two systems, including ordinary contextual rewriting, directed rewriting, insertion, markup, and parallel rewriting. At the time of writing, the regression suite contains 2,217 individual cases. In all tested cases the resulting transducers were structurally identical, differing only in state numbering. This is consistent with the fact that both the worsening construction and *foma*'s multitape-based compiler build the same basic pattern of identity transitions outside rewrite centers and canonical cross-product transductions inside rewrite spans, so under standard Hopcroft-style minimization [8] over input:output label pairs they yield the same result in the shared cases tested here.

4 Notation

We work over an alphabet Σ of ordinary symbols and use auxiliary symbols $\{\#, <, >\}$, disjoint from Σ, to mark word boundaries and candidate rewrite sites. Let $\Gamma = \Sigma \cup \{\#, <, >\}$. Intermediate languages are subsets of Γ^*, and intermediate transducers are relations in $\Gamma^* \times \Gamma^*$. When convenient, we identify a language $L \subseteq \Gamma^*$ with its identity relation $\{(x, x) \mid x \in L\}$. The final compiled rule transducers, however, relate strings over Σ only.

Composition is denoted by $\circ$, with the convention

$$T_1 \circ T_2 = \{(x, z) \mid \exists y\, ((x, y) \in T_1 \wedge (y, z) \in T_2)\}.$$

For regular languages A, B, we express the cross product as

$$A{:}B = \{(x, y) \mid x \in A, y \in B\},$$

that is, the regular relation pairing every string in A with every string in B. The complement of a language $L \subseteq \Gamma^*$ is $\neg L = \Gamma^* - L$. For a relation T, $\mathrm{range}(T) = \{y \mid \exists x\, (x, y) \in T\}$ and $\mathrm{dom}(T) = \{x \mid \exists y\, (x, y) \in T\}$.

PyFoma examples use the regular-expression syntax shown in Table 1. In the transducer diagrams (Figure 2 and later), a `.` on a transition denotes the identity transition on any symbol not explicitly included in that transducer's alphabet, and multiple identical transitions are merged into one with comma-separated labels.

When Σ occurs inside a transducer expression, we use it as shorthand for the identity relation on symbols from Σ.

5 Rewrite Rules

Rewrite rules originated in phonological theory, notably in Chomsky and Halle's *The Sound Pattern of English* [3], where rules like $b \to p / \_\ \#$ devoice a b word-finally. Johnson [13] showed that such rules, when contexts are regular languages, can be modeled as FSTs, if

[2] A subset of these grammars is publicly available on the PyFoma JavaScript port site.

Operator	**PyFoma**	**Regular expression**
union	|	$\cup$
intersection	&	$\cap$
cross-product	:	:
Kleene star	*	*
Kleene plus	+	+
wildcard	.	Σ
epsilon	''	ϵ
subtraction	-	$-$
composition	@	$\circ$
character class	[a - z]	N/A
negated character class	[^a-z]	N/A
domain	$^input(T)	dom(T)
range	$^output(T)	range(T)
rewrite rule	$^rewrite(A:B / L _ R)	$A \rightarrow B / L \_ R$

Table 1: PyFoma and standard regular expression syntax for various finite-state operators.

they are assumed not to reapply to their own outputs, unlike context-sensitive grammar rules.

A general rewrite rule is written as:

$$A \rightarrow B / L_1 \_ R_1, \ldots, L_n \_ R_n,$$

where A, B, L_i, and R_i are regular languages, and # denotes a word edge in the contexts. For example, the rule $b \rightarrow p / \_ \#, \#\_$ would turn a b to a p either word-initially or word-finally.

We assume the standard simultaneous interpretation of rewrite rules. A rule application chooses a possibly empty set of non-overlapping substrings of the input, each belonging to A and satisfying at least one listed context $L_i \_ R_i$ on the input side, replaces each chosen substring once by its image under $A{:}B$, and copies all remaining symbols unchanged. Thus the rule defines a relation between whole input strings and whole output strings; it is not interpreted as an iterative procedure that repeatedly reapplies the rule to its own output. Later, in Section 7.11, we discuss the alternative variant where contexts are checked on the output side. If several distinct sets of legal occurrences exist, the rule is nondeterministic unless further preferences such as leftmost or longest are imposed.

For example, the rule $a^+ \rightarrow x / a \_ a$ replaces sequences of one or more a with x when flanked by a on both sides. An input string aaa rewrites to axa. For $aaaa$, the legal simultaneous applications yield $\{axxa, axa\}$: either the two middle a's are rewritten separately, or the middle aa is rewritten as one span. Outputs such as $axaa$ or $aaxa$ omit a licensed rewrite site and are therefore excluded under obligatory simultaneous semantics. If we additionally require longest-match behavior, then only axa is produced from $aaaa$.

6 Worsening

6.1 The worsening trick

[6, 5] use a strategy of imposing a preference relation on strings in a regular language to filter out candidate strings from the same set deemed to be 'suboptimal'. To illustrate the general technique, consider a language L_m containing strings with optional morpheme boundaries (a +-symbol), e.g., {de+construct+ion+s, deconstruct+ion+s, deconstructions, ... }. To select those strings in L_m with the maximal number of + symbols, define a worsening transducer W that removes at least one + (but may remove more):

$$W = (\Sigma^* \ (+ : \epsilon) \ \Sigma^*)^+,$$

where ϵ is the empty string. The language of strings with maximal + symbols is:

$$\max_{+}(L_m) = L_m \cap \neg\text{range}(L_m \circ W),$$

where range(T) denotes the output language of transducer T. This filters out strings from L_m that can be derived by removing +-symbols, retaining only those with the most +-symbols, e.g., de+construct+ion+s.

For transducers, suppose a transducer T generates candidate transductions, and we want to filter its output. Define a worsening transducer W that encodes a preference relation among competing outputs, and compute:

$$T \circ \neg\text{range}(T \circ W),$$

or, symmetrically

$$\neg\text{range}(\text{dom}(T) \circ W) \circ T,$$

if we want to filter out strings from the input language of a transducer T. The key idea is that the worsener does not describe the bad candidates directly; instead, it maps better candidates to worse competing candidates, which can then be removed algebraically in one step. Here the language $\neg\text{range}(\cdots)$ is understood as its identity relation when composed with a transducer.

This retains only outputs that cannot be worsened, forming the core of our compilation strategy. Figure 1 illustrates the idea.

7 Compiling Rewrite Rules—Basic Cases

We compile a rewrite rule $A \rightarrow B / L \_ R$ into an FST as follows using the notation introduced above. Here, T denotes the center transduction $A{:}B$, a cross-product of two regular languages, though we will later make T more general (see 7.6).

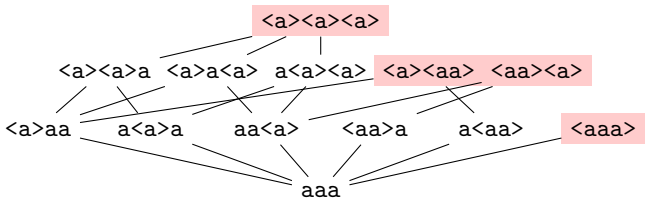


Figure 1: Hasse-style diagram of the ordering induced by the obligatory worsener for the rewrite rule $a^+ \to x$ on input *aaa*. Nodes represent candidate strings; edges show selected one-step worsenings obtained by removing a single bracket pair, rather than every comparable pair in the order. The maximal elements are `<a><a><a>`, `<a><aa>`, `<aa><a>`, and `<aaa>`; they survive filtering and produce the outputs *xxx*, *xx*, *xx*, and *x*, respectively.

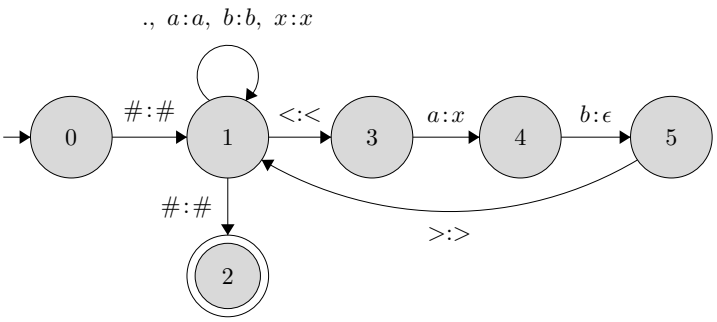


Figure 2: T_{base} for the rule $ab \to x$. This transducer is independent of the contexts L and R; the same machine is used for any context and the restriction to valid sites is enforced later by composition with L_{context}. The symbol **.** denotes PyFoma's catch-all transition label (identity on any symbol not explicitly listed in the current alphabet).

7.1 Base Transducer

Generate all possible rewrite candidates with:

$$T_{base} = \#\,(\Sigma \cup < T >)^*\,\#, \tag{1}$$

where Σ represents non-rewritten symbols, and $< T >$ marks a rewrite site where strings from A are replaced by strings from B.

A typical path through T_{base} corresponds to a candidate such as $\#\,a < a : x\; b : \epsilon > a\,\#$, representing a rewrite of ab to x, with other symbols in an identity relation (see Figure 2). We explicitly insert # symbols at the beginning and end of every string so that word-edge contexts (such as _ # or # _) can be treated uniformly as ordinary left- and right-contexts in the same restriction mechanism—no special handling for string boundaries is required.

7.2 Context Restriction

We then restrict bracket pairs (rewrite sites) to the context $L \_ R$. For a single specified context $L \_ R$ contexts, we can filter out illicit centers by composing $\neg(\neg(\Gamma^* \, L) \, < \, \Gamma^* \cup \Gamma^* \, > \, \neg(R \, \Gamma^*)) \circ T_{base}$. However, a generalization of this formula grows exponentially with the number of context pairs $L_i \_ R_i$, which is why we resort to a method by Yli-Jyrä and Koskenniemi [22]. Define the center language

$$C = < \, (\Gamma - >)^* \, >,$$

which denotes a single bracketed rewrite site. The context restriction ensures that every occurrence of C appears only in $L \_ R$.

Introduce a fresh auxiliary symbol $\cdot \notin \Gamma$, and define

$$K = \widetilde{L} \cdot \Gamma^* \cdot \widetilde{R},$$

where $\widetilde{L}$ and $\widetilde{R}$ are obtained from L and R by freely inserting symbols from $\{<, >\}$ between ordinary symbols. Intuitively, the two copies of $\cdot$ delimit a single site in the string. Thus K is the language of strings in which the marked position is licensed by the left and right contexts.

We can now express the bad strings, namely those containing a dotted rewrite site that is not properly surrounded, as

$$\Gamma^* \cdot C \cdot \Gamma^* \; - \; \Gamma^* K \Gamma^*.$$

Deleting $\cdot$ then projects away the temporary marker, and negation yields the admissible candidates:

$$L_{context} = \neg h_{\{\cdot\}}(\Gamma^* \cdot C \cdot \Gamma^* - \Gamma^* K \Gamma^*) \tag{2}$$

Here, $h_{\{\cdot\}}$ is the homomorphism deleting $\cdot$. With this, every rewrite site must satisfy the context test, and the two copies of $\cdot$ ensure that the same site is quantified in the two subexpressions [11].

Compose this with T_{base} to get a rule transducer that rewrites relevant input strings nondeterministically (i.e. optionally), but only in the correct context:

$$T_{\text{optional\_rewrite}} = L_{context} \circ T_{base} \tag{3}$$

7.3 Obligatory Rewriting

The above produces a transducer that models optional rewriting in the correct contexts, since we have not yet introduced a mechanism that forces a legal rewrite to apply. To enforce obligatory rewriting, define a worsening transducer that removes at least one bracket pair, setting up the partial order:

$$W_{\text{obl}} = \Gamma^* \; \underbrace{(<\!:\epsilon \; \Sigma^* \; >\!:\epsilon)}_{\text{Remove bracket pair}} \; \underbrace{(<\!:\epsilon \; \Sigma^* \; >\!:\epsilon \cup \Gamma)^*}_{\text{Remove more or repeat}} \tag{4}$$

This maps, e.g., $\# \, a \, < \, a \, b \, > \, a \, \#$ to $\# \, a \, a \, b \, a \, \#$, undoing a rewrite site (Figure 3 shows the transducer). We now filter suboptimal candidates as in Section 6, producing our final rule transducer:

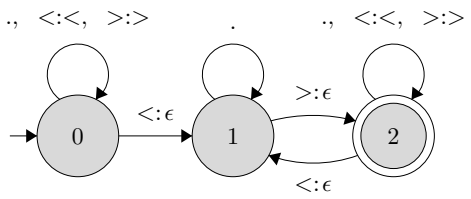


Figure 3: The worsening transducer W_{obl} which nondeterministically removes at least one bracket pair. The symbol **.** denotes PyFoma's catch-all transition label: in an FST, it repeats any symbol not explicitly listed in the transducer's current alphabet.

$$T_{rule} = \neg\text{range}(\text{dom}(T_{\text{optional\_rewrite}}) \circ W_{\text{obl}}) \circ T_{\text{optional\_rewrite}} \tag{5}$$

This retains candidates from which no additional legal non-overlapping rewrite site can be obtained by adding bracket pairs. As a final step, we remove the auxiliary symbols by applying the homomorphism $\{<, >, \#\} \mapsto \epsilon$.

The logic of the construction is simple. T_{base} generates all candidate bracketings and replacements. $L_{context}$ removes those with a bracketed site in an illicit context, leaving the legal optional rewrites. W_{obl} then makes one legal candidate worse than another if it is obtained by removing one or more bracket pairs, i.e., by undoing one or more legal rewrites. Hence $\neg\text{range}(\text{dom}(T_{\text{optional\_rewrite}}) \circ W_{\text{obl}}) \circ T_{\text{optional\_rewrite}}$ keeps exactly those legal candidates from which no legal rewrite has been omitted. Figure 4 illustrates this for the rule $a \to b/c\_d$.

A useful property of the obligatory construction is that the resulting rule transducer is total on Σ^*. For every input string, T_{base} admits the candidate in which no rewrite is applied, so all symbols are copied identically and no rewrite brackets are inserted. This candidate trivially survives context filtering, since $L_{context}$ excludes only strings containing a bracketed site in an illicit context. Moreover, on the candidates that survive this step, W_{obl} is acyclic, since it worsens a candidate only by removing one or more bracket pairs. For any fixed input, only finitely many bracketings are possible, so at least one surviving candidate is not worsened by any other, and the final filtering step therefore cannot eliminate them all.

The above gives the basic construction for a rule $A \to B/L\ \_\ R$. We now briefly consider extensions to this basic case.

7.4 Multiple Contexts

For multiple alternative rewrite contexts

$$L_1\ \_\ R_1, \ldots, L_n\ \_\ R_n,$$

the same dotted-position construction extends directly by union. Define

$$K = \bigcup_{i=1}^{n} \widetilde{L_i} \cdot \Gamma^* \cdot \widetilde{R_i},$$

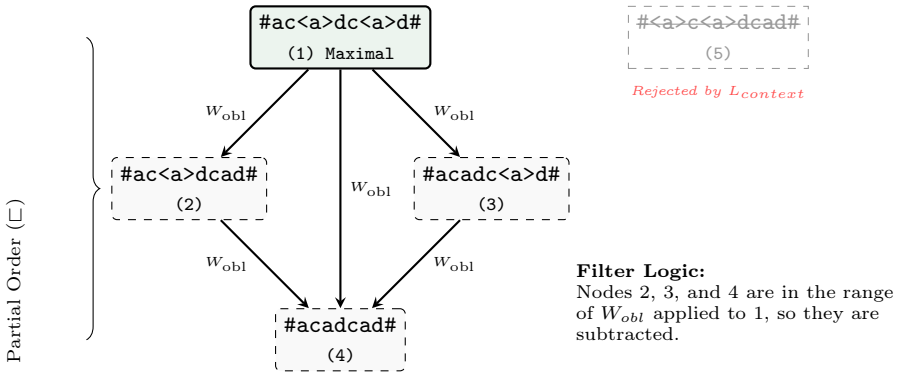


Figure 4: Some candidates generated by T_{base} for $a \to b/c\_d$ on input `#acadcad#`. Candidate 5 contains a rewritten site outside the context c_d and is removed by $L_{context}$. Candidate 1 has competitors 2, 3, and 4 obtained by deleting one or both bracket pairs, so only 1 survives obligatory filtering. Obligatory worsening compares bracketed candidate strings on the input side; only those representations are shown here.

where each $\widetilde{L_i}$ and $\widetilde{R_i}$ is obtained from L_i and R_i by freely inserting symbols from $\{<, >\}$ between ordinary symbols. Thus K is the language of strings in which the $\cdot$-marked span is surrounded by at least one allowed context pair. In other words, disjunction of contexts is handled entirely by union at the level of the dotted context language.

Using the same center language

$$C =< (\Gamma - >)^* >,$$

we obtain

$$L_{context} = \neg h_{\{\cdot\}}(\Gamma^* \cdot C \cdot \Gamma^* - \Gamma^* K \Gamma^*) \tag{6}$$

This ensures that every rewrite site belongs to at least one valid context pair [22]. The worsening transducer W_{obl} is unchanged.

7.5 Directed Rewriting

To control ambiguity (e.g., $a^+b^+c^+ \to x$ on input *aabbcc*, where the four matching spans *abbc*, *abbcc*, *aabbc*, and *aabbcc* produce $\{axc, ax, xc, x\}$), define further worsening transducers:

- **Leftmost**: W_{left}, shifting opening brackets rightward, e.g., $< aabbcc > \to a < abbcc >$.
- **Longest**: W_{long}, shortening spans, e.g., $< aabbcc > \to < aabbc > c$.
- **Shortest**: W_{short}, lengthening spans, e.g., $< aabbc > c \to < aabbcc >$.

See Section B for the exact constructions of the worsening transducers above.

When several criteria are active, we combine their worsening relations by union and construct the corresponding filter exactly as in the basic case. Thus a candidate is excluded whenever some competing candidate for the same input is better under *any* one of the

active criteria. For example, to require rewrites to be obligatory, leftmost, and longest, we construct:

$$T_{rule} = \neg\text{range}(\text{dom}(T_{\text{optional\_rewrite}}) \circ (W_{\text{obl}} \cup W_{\text{left}} \cup W_{\text{long}})) \circ T_{\text{optional\_rewrite}} \quad (7)$$

So omitted obligatory rewrites, non-leftmost choices, and non-longest choices are each sufficient for exclusion. For example, applying $a^+ \rightarrow x / a \_ a$ to *aaaaa* with all three criteria active yields *axa* as the only possible output. In PyFoma, this is expressed as:

```
$^rewrite(a+:x  / a _ a, leftmost = True, longest = True)
```

7.6 Arbitrary Transductions

The input T can be any FST, not necessarily a cross-product of regular languages $A:B$. In PyFoma, the basic interface is `$^rewrite(T / L1 _ R1, ..., Ln _ Rn)`.

A word-final devoicing rule for stops (b,d,g) is:

```
$^rewrite(b:p|g:k|d:t / _ #)
```

This allows arbitrary transductions without introducing additional rule types.

7.7 Markup Transducers

Markup transducers, common in tools such as *xfst* and *foma* [2], insert material on either or both sides of some target string. Such transducers are commonly used for HTML-style markup [2]. A simple example is wrapping vowels with tags while leaving the vowel itself unchanged:

```
$^rewrite('':'<VOWEL>' [aeiou] '':'</VOWEL>')
```

Other toolkits often have a special rule type to perform this kind of insertion. Because our rules are not restricted to cross-products, we need no special treatment for such transductions.

7.8 Parallel Rewriting

Parallel rewriting, as in Kempe and Karttunen [18], applies multiple rules simultaneously. Consider the pair of rules

$$ab \rightarrow x / \_ba, \qquad ba \rightarrow y / ab\_$$

on input `abba` (i.e., `#abba#`). One candidate generated by an indexed extension of T_{base} marks the two non-overlapping rewrite sites with indexed brackets:

$$\#<_1 \texttt{ab} >_1<_2 \texttt{ba} >_2\#$$

Context restriction (via the dotted-position method) and obligatory worsening are then enforced separately for each indexed bracket pair ($<_i$, $>_i$). The same typed-bracket mechanism extends directly to any number of parallel rules and to the different rewrite strategies discussed above (obligatory, leftmost, longest, etc.); in the implementation, the indices and auxiliary brackets are projected away in the final step exactly as in the single-rule case, yielding, in the present example:

```
xy
```

7.9 Weights

PyFoma supports weights in the tropical semiring [20], and weighted compilation fits naturally into the same construction because worseners never compare weights. For example, word-final devoicing with cost 1.0:

```
$^rewrite((b:p | g:k | d:t)<1.0> / _ #)
```

Allowing non-rewriting with a different cost:

```
$^rewrite(((b:p | g:k | d:t)<1.0> | [bdg]<0.0>) / _ #)
```

No special treatment is required: weights enter only through the underlying transduction T, while the later stages merely filter candidates, so the weights of surviving candidates are exactly those assigned by T.

7.10 Epsilons in Left-Hand Sides

For many applications, rewrite rules whose left-hand side contains ϵ require a more restricted interpretation. A rule such as $\epsilon \to x$ or $b^* \to x$ would otherwise license zero or arbitrarily many insertions at the same position. Often, the desired behavior is to allow at most one insertion site at each boundary between adjacent positions, including word edges, so that input `aa` yields the single output `xaxax` rather than arbitrarily many insertions at the same point.

PyFoma adopts by default the bounded interpretation just described. This is enforced by an initial filter excluding candidate strings that contain consecutive empty bracket pairs <><><>.

The transducer T_{base} is restricted on its input side to exclude consecutive bracket pairs using:

$$L_{\text{dotted}} = \Gamma^* - (\Gamma^* \ <><>\ \Gamma^*) \tag{8}$$

Thus, we modify the earlier T_{base}; $T_{base} = L_{\text{dotted}} \circ \#\,(\Sigma \cup <\ T\ >)^*\,\#$.[3]

7.11 Spreading and Harmony Rules

Karttunen [16] noted that one can achieve different behavior of rewrite rules depending on whether the contextual constraints $L_i \ \_ \ R_i$ are assumed to hold on the input side or on the output side. The traditional assumption, and the default adopted here, is that they hold on the input side. However, by swapping the order of composition in (3), we can constrain the contexts to hold on the output side:

$$T_{\text{optional\_rewrite}} = T_{base} \circ L_{context} \tag{9}$$

A minimal example of the different behavior is seen in a rule such as

$$ab \to x / \ ab \ \_ \ a$$

[3] Such rules were called **dotted** rules in the xfst-toolkit [2] and we inherit the name.

and the input *abababa*, where constraining the context on the input yields one output, *abxxa*, whereas constraining the context on the output side produces two different outputs, $\{ababxa, abxaba\}$[4] (see Table 2).

	Input-side context	**Output-side context**
Input	a b a b a b a	a b a b a b a
Output(s)	a b x x a	a b a b x a
		a b x a b a

Table 2: The rule $ab \rightarrow x / ab\_a$ on input `abababa`, with contexts checked on the input side vs. output side.

Although the input-side restriction often gives the expected behavior for ordinary rewrite rules, output-side restriction has an important use in computational phonology, related to vowel-harmony rules. This arises when a vowel acquires its quality from a preceding vowel in the word. In Finnish, certain suffix vowels are underspecified and acquire their surface quality from the preceding vowel [15]. For example, the word **kylä** 'village' has front vowels, **{y,ä}** in it, and any suffixes added to it should also receive front vowels. Commonly, these underspecified suffixes are denoted with capital letters (A = either a or ä, O = either o or ö). A typical rule to handle the front-vowel case would rewrite A as ä if the previous vowel is ä or ö or y, and O as ö in the same circumstance. With a rule like

```
1  $^rewrite(A:ä | O:ö / (ä|ö|y) $nonvowel+ _ )
```

this can be achieved, but only if the contexts hold on the output side, which allows the vowel quality to 'spread' rightward (see Table 3).

	Input-side context	**Output-side context**
Input	k y l ä + s s A k O	k y l ä + s s A k O
Output(s)	k y l ä + s s ä k O	k y l ä + s s ä k ö

Table 3: A harmony rule whose context must be checked on the output side to allow the feature to spread rightward.

8 Conclusion

We have presented a rewrite compilation pattern built from three ingredients: candidate generation, context restriction, and preference filtering by worsening. The same pattern covers obligatory rewriting, multiple contexts, directed rewriting, arbitrary transductions, markup, parallel rules, and weights.

[4] [16] actually makes a 4-way distinction based on whether the left and right contexts hold on the input or output side.

More broadly, rewrite compilation is only one application of worsening. Although the technique arose in finite-state phonology, it is better understood as a general finite-state design pattern for turning procedural-looking selection tasks into algebraic ones: generate a regular set of candidates, define a worsening relation, and filter away the non-optimal candidates. Besides its established use in finite-state Optimality Theory [6, 5], this perspective points to a wider role for worsening in extracting preferred subsets of regular languages and relations—for example shortest, longest, or otherwise minimal representatives—and in encoding complex preference orders declaratively within the regular calculus itself.

References

[1] Kenneth R. Beesley. Kleene, a free and open-source language for finite-state programming. In Iñaki Alegria and Mans Hulden, editors, *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 50–54, Donostia–San Sebastián, Spain, July 2012. Association for Computational Linguistics. https://aclanthology.org/W12-6209/.

[2] Kenneth R. Beesley and Lauri Karttunen. *Finite-State Morphology*. CSLI Publications, Stanford, CA, 2003.

[3] Noam Chomsky and Morris Halle. *The Sound Pattern of English*. Harper and Row, New York, 1968.

[4] Dale Gerdemann. Mix and match replacement rules. In Núria Bel, Erhard Hinrichs, Petya Osenova, and Kiril Simov, editors, *Proceedings of the Workshop on Adaptation of Language Resources and Technology to New Domains*, pages 39–47, Borovets, Bulgaria, September 2009. Association for Computational Linguistics. https://aclanthology.org/W09-4106/.

[5] Dale Gerdemann and Mans Hulden. Practical finite-state Optimality Theory. In Iñaki Alegria and Mans Hulden, editors, *Proceedings of the 10th International Workshop on Finite State Methods and Natural Language Processing*, pages 10–19, Donostia–San Sebastián, Spain, July 2012. Association for Computational Linguistics. https://aclanthology.org/W12-6202/.

[6] Dale Gerdemann and Gertjan van Noord. Approximation and exactness in finite-state Optimality Theory. In Jason Eisner, Lauri Karttunen, and Alain Thèriault, editors, *Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology*, pages 34–45, Luxembourg, August 2000. International Committee on Computational Linguistics. https://aclanthology.org/W00-1804/.

[7] Kyle Gorman and Richard Sproat. *Finite-State Text Processing*, volume 50 of *Synthesis Lectures on Human Language Technologies*. Morgan and Claypool Publishers, 2021. https://doi.org/10.2200/S01086ED1V01Y202104HLT050.

[8] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971. https://doi.org/10.1016/B978-0-12-417750-5.50022-1.

[9] Mans Hulden. *Finite-State Machine Construction Methods and Algorithms for Phonology and Morphology*. PhD thesis, University of Arizona, Tucson, AZ, 2009. https://repository.arizona.edu/handle/10150/196112.

[10] Mans Hulden. Foma: A finite-state compiler and library. In *Proceedings of the Demonstrations Session at EACL 2009*, pages 29–32, Athens, Greece, April 2009. Association for Computational Linguistics. https://aclanthology.org/E09-2008/.

[11] Mans Hulden. Regular expressions and predicate logic in finite-state language processing. In Jakub Piskorski, Bruce W. Watson, and Anssi Yli-Jyrä, editors, *Finite-State Methods and Natural Language Processing: Post-Proceedings of the 7th International Workshop, FSMNLP 2008*, volume 191 of *Frontiers in Artificial Intelligence and Applications*, pages 82–89. IOS Press, Amsterdam, 2009.

[12] Mans Hulden, Michael Ginn, Miikka Silfverberg, and Michael Hammond. PyFoma: A Python finite-state compiler module. In Yixin Cao, Yang Feng, and Deyi Xiong, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 258–265, Bangkok, Thailand, August 2024. Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.acl-demos.24.

[13] C. Douglas Johnson. *Formal Aspects of Phonological Description*. Mouton, The Hague, 1972.

[14] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994. https://aclanthology.org/J94-3001/.

[15] Fred Karlsson. *Finnish: An Essential Grammar*. Routledge, London, 3 edition, 2013. https://doi.org/10.4324/9780203520741.

[16] Lauri Karttunen. The replace operator. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Cambridge, MA, June 1995. Association for Computational Linguistics. https://doi.org/10.3115/981658.981661.

[17] Lauri Karttunen. Directed replacement. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 108–115, Santa Cruz, CA, June 1996. Association for Computational Linguistics. https://aclanthology.org/P96-1015/.

[18] André Kempe and Lauri Karttunen. Parallel replacement in finite-state calculus. In *COLING 1996, Volume 2: The 16th International Conference on Computational Linguistics*, pages 622–627, Copenhagen, Denmark, 1996. International Committee on Computational Linguistics. https://aclanthology.org/C96-2105/.

[19] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238, Santa Cruz, CA, June 1996. Association for Computational Linguistics. https://aclanthology.org/P96-1031/.

[20] Jean-Éric Pin. Tropical semirings. In Jeremy Gunawardena, editor, *Idempotency*, volume 11 of *Publications of the Newton Institute*, pages 50–69. Cambridge University Press, Cambridge, 1998. `https://hal.science/hal-00113779`.

[21] Nathan Vaillette. *Logical Specification of Finite-State Transductions for Natural Language Processing*. PhD thesis, Ohio State University, Columbus, OH, 2004.

[22] Anssi Yli-Jyrä and Kimmo Koskenniemi. Compiling contextual restrictions on strings into finite-state automata. In Loek Cleophas and Bruce W. Watson, editors, *The Eindhoven FASTAR Days: Proceedings*, Eindhoven, The Netherlands, 2004. Technische Universiteit Eindhoven.

A PyFoma Implementation

The following PyFoma code illustrates the rewrite rule compilation, using < and > for brackets and # for word boundaries. It relies on the built-in `$^restrict()` for context restriction. The code is a simplified adaptation from PyFoma's actual `$^rewrite()` functionality, which handles multiple contexts and modalities. As in *foma* and *xfst*, the symbol `.` has two related interpretations: in regular expressions it denotes any symbol, while in compiled FSTs it functions as a catch-all transition for symbols not explicitly present in the machine's current alphabet.

Listing 1: PyFoma implementation of a rewrite rule compilation $A \rightarrow B \ / \ L \ \_ \ R$

```
from pyfoma import re
t = {}
t['A'] = re("a")        # A, B, L, R
t['B'] = re("b")        # can be
t['L'] = re("c")        # arbitrary regular languages
t['R'] = re("d")        # use # for edges in contexts

t['T']          = re("$A:$B<1.0>", t) # Cross-product with weight
t['sigma']      = re("[^<>#]")
t['base']       = re("# (< $T > | $sigma)* #", t)
t['leftc']      = re("$^ignore($L, [<>])", t)
t['rightc']     = re("$^ignore($R, [<>])", t)
t['context']    = re("$^restrict(< [^>]* > / $leftc _ $rightc)", t)
t['optrewr']    = re("$context @ $base", t)
t['remrewr']    = re(" <:'' [^>]* >:'' ")
t['oblworsen'] = re(".* $remrewr (. | $remrewr)*", t)
t['badrewr']    = re("$^output($^input($optrewr) @ $oblworsen)", t)
t['oblrewr']    = re("~$badrewr @ $optrewr", t)
t['rule']       = t['oblrewr'].map_labels({'<':'', '>':'', '#':''})\
                  .epsilon_remove().determinize().minimize_as_dfa()
```

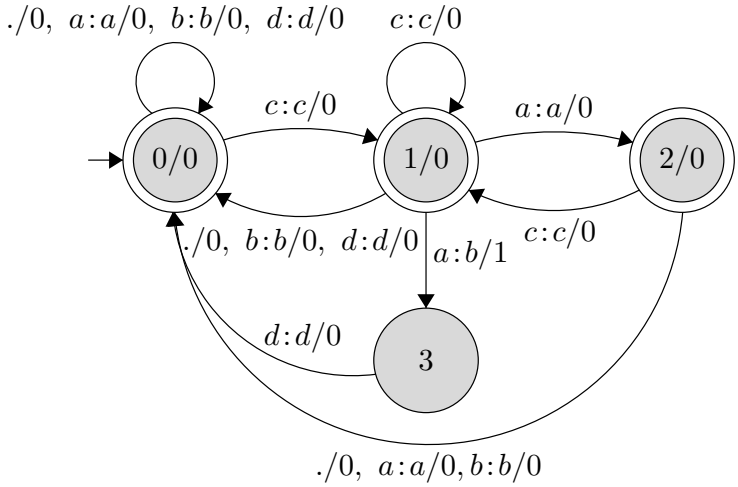


Figure 5: Resulting transducer compiled from Listing 1.

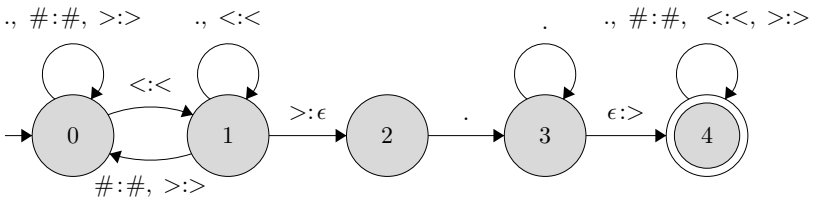


Figure 6: The worsener W_{short}. The symbol **.** denotes PyFoma's catch-all transition label: in an FST, it repeats any symbol not explicitly listed in the transducer's current alphabet.

B Directed Rewriting Worseners

For compactness, we use postfix ? for optional material, so that $X?$ abbreviates $(X \cup \epsilon)$.

$$W_{\text{short}} = \Gamma^* \; \underbrace{< \; \Sigma^* \; >:\epsilon}_{\substack{\text{keep span,} \\ \text{delete old } >}} \; \underbrace{\Sigma^+ \; \epsilon :>}_{\substack{\text{extend span} \\ \text{rightward}}} \; \Gamma^*$$

$$W_{\text{long}} = \Gamma^* \; \underbrace{< \; \Sigma^+ \; \epsilon : (> \; <?) \; \Sigma}_{\substack{\text{close earlier,} \\ \text{optionally reopen}}} \; \underbrace{\big((< \cup >) : \epsilon \cup \epsilon : (< \cup >) \cup \Sigma\big)^*}_{\substack{\text{copy remainder and} \\ \text{clean up old brackets}}} \; \Gamma^* \tag{10}$$

$$W_{\text{left}} = \Gamma^* \; \underbrace{<: \epsilon \; \Sigma^+ \; \epsilon :< \; \Sigma^*}_{\substack{\text{delete old } <, \text{ then} \\ \text{re-insert farther right}}} \left(\underbrace{\epsilon :> \; \Sigma^+ \; >: \epsilon}_{\substack{\text{new match ends} \\ \text{before old match}}} \cup \underbrace{>: \epsilon \; \Sigma^* \; \epsilon :>}_{\substack{\text{new match ends at} \\ \text{or after old match}}} \right) \; \Gamma^*$$

W_{short} moves a closing bracket rightward, making a rewrite span less short. W_{long} moves a closing bracket leftward, optionally reopening immediately so that later bracket structure can still be preserved. Finally, W_{left} moves the opening bracket rightward. Its initial move is the mirror image of W_{short} (Figure 6), but the closing bracket cannot simply be left untouched: a later-starting competing span may end earlier, at the same point, or later. The two alternatives therefore cover the possible placements of the new closing bracket relative to the old one. In each case, the worsener relates a preferred candidate to a competing candidate that is worse with respect to the corresponding criterion.
Examples of single applications to bracketed candidate strings:

- W_{short}: $\langle \texttt{aabbc} \rangle \texttt{c} \mapsto \langle \texttt{aabbcc} \rangle$ (moves > right)
- W_{long}: $\langle \texttt{aabbcc} \rangle \mapsto \langle \texttt{aabbc} \rangle \texttt{c}$ (moves > left)
- W_{left}: $\langle \texttt{aabbcc} \rangle \mapsto \texttt{a} \langle \texttt{abbcc} \rangle$ (moves < right)

Author Index

Angstadt, Kevin, 17

Burdick-Pless, Jesse, 1

Day, Alexander, 1

Efimov, Oleg, 17

Famous, Lucas, 1
FitzPatrick, Teddy, 1

Ginn, Michael, 32

Hulden, Mans, 32

Skadron, Kevin, 17

Tracy, Tommy II, 17

Yu, Chang yu, 1

Zheng, Whiting, 17