Linear Matching of JavaScript Regular Expressions

AURÈLE BARRIÈRE and CLÉMENT PIT-CLAUDEL, EPFL, Switzerland

Modern regex languages have strayed far from well-understood traditional regular expressions: they include features that fundamentally transform the matching problem. In exchange for these features, modern regex engines at times suffer from exponential complexity blowups, a frequent source of denial-of-service vulnerabilities in JavaScript applications. Worse, regex semantics differ across languages, and the impact of these divergences on algorithmic design and worst-case matching complexity has seldom been investigated.

This paper provides a novel perspective on JavaScript's regex semantics by identifying a larger-than-previously-understood subset of the language that can be matched with linear time guarantees. In the process, we discover several cases where state-of-the-art algorithms were either wrong (semantically incorrect), inefficient (suffering from superlinear complexity) or excessively restrictive (assuming certain features could not be matched linearly). We introduce novel algorithms to restore correctness and linear complexity. We further advance the state-of-the-art in linear regex matching by presenting the first nonbacktracking algorithms for matching lookarounds in linear time: one supporting captureless lookbehinds in any regex language, and another leveraging a JavaScript property to support unrestricted lookaheads and lookbehinds. Finally, we describe new time and space complexity tradeoffs for regex engines. All of our algorithms are practical: we validated them in a prototype implementation, and some have also been merged in the V8 JavaScript implementation used in Chrome and Node.js.

CCS Concepts: • **Theory of computation → Regular languages**; **Design and analysis of algorithms**.

Additional Key Words and Phrases: Regex, Automata, JavaScript

1 INTRODUCTION

The expressive power and computational complexity of traditional regular expressions (composed only of characters, concatenations, alternations, and Kleene stars) are well understood. From an expressive-power standpoint, they are known to be exactly equivalent to finite automata. From a computational-complexity standpoint (deciding whether a regular expression r matches a string s), well-known trade-offs exist between pre-processing and matching time. For instance, one can create a deterministic finite automaton (DFA) for r in worst-case time complexity $O\left(2^{|r|}\right)$ and then perform matching in $O(|s|)$, where $|r|$ and $|s|$ represent the respective lengths of r and s. Alternatively, to avoid the expensive cost of determinization, one can instead create a nondeterministic finite automaton (NFA) for r [Thompson 1968] in $O(|r|)$ and then perform matching with worst-case time complexity $O(|r| \times |s|)$ [Cox 2007; Pike 1987].

Modern regular expressions, which we call *regexes* to differentiate them from traditional ones, have strayed far from their ancestors: they are significantly more complex and expressive, but much less well studied. Their nontraditional features have allowed them to become one of the most pervasive embedded domain-specific languages in programming: one study found them in more than 30% of npm and PyPI packages [Davis et al. 2018]; another in 42% of Python developments [Chapman and Stolee 2016]; and 8 of the top 10 Tiobe languages support them natively [TIOBE 2023]. One crucial extension is the introduction of capture groups, requiring regex engines to return not only whether there is a match, but also which part of the input string matches each sub-expression inside parentheses. For instance, when matching `/(a*)b/` on `"caabd"`, modern regex engines report a match on sub-string `"aab"`, and a sub-match on `"aa"` for the capture group `/(a*)/`. Capture groups fundamentally transform the problem from one of language recognition to one of pattern matching: traditional regular expression matching checks whether an input string belongs to a formal language, whereas modern regex matching computes the positions of the sub-strings

Authors' address: Aurèle Barrière, aurele.barriere@epfl.ch; Clément Pit-Claudel, clement.pit-claudel@epfl.ch, EPFL, Lausanne, Switzerland.

matched by the regex and each of its capture groups. As such, regexes represent not a language, but a way to search and segment a piece of text. This problem is often ambiguous, so modern regex languages define priority rules to deterministically disambiguate results when a regex can match a string in multiple ways. For instance, when matching `/(a|a*)/` on `"aa"`, most modern regex engines give priority to the left branch of the alternation and return a match only on the first letter (this differs from the longest-match semantics often found in lexers).

Other modern features include backreferences, lookarounds (lookaheads and lookbehinds, both positive and negative), anchors, character classes, and repetition. Backreferences force a later part of a regex to match the same substring as an earlier group: for instance, `/(a*)b\1/` matches `"aabaa"` but not `"aaba"`, as `\1` matches exactly the sub-string captured by the first group `/(a*)/`. Lookarounds condition a match without consuming the corresponding characters: for instance, the regex `/(?<=£)1/` matches the character `"1"` in `"£1.2"` and does not match anything in `"v1.2"` (the `"£"` that is checked for by the lookbehind is not included in the final match). Anchors `^`, `$`, and `\b` match at the beginning and end of the string or at a word boundary, character classes match ranges of characters (e.g., `[a-e]` matches any character between a and e), and counted repetition repeat the same regex multiple times (`(a|b){4,8}` matches any combinations of 4 to 8 `"a"` and `"b"`). To illustrate the convenience of these features together, the regex `/(?<=PLDI)[0-9]{2,4}/` matches the year of a reference to a PLDI paper (e.g., `"2024"` in `"PLDI2024"`).

These nontraditional features and their interplay are only partially understood. It is well-known, for example, that backreferences make the matching problem NP-hard [Aho 1990; Dominus 2000]. At the other end of the spectrum, linear-time algorithms are in common use for features such as capture groups, anchors and character classes [Gallant 2014; Google 2022]. In-between is uncertainty: to the best of our knowledge, no such matching complexity result has been stated for, e.g., lookarounds with capture groups.

Accordingly, modern regex engines fall into two categories. *Backtracking* engines support all features (including backreferences) but suffer from algorithmic blowups — even on simple patterns composed exclusively of traditional features for which linear-time algorithms are known. This disastrous worst-case performance has serious security implications, ranging from partial service degradation to complete outages of large websites [Cloudflare 2019; Stack Exchange 2016]. A recent study [Staicu and Pradel 2018] estimated that 12% of JavaScript-based web servers are vulnerable to regex-based denial-of-service attacks, or "ReDoS".

These security concerns and complexity issues have led to revived interest in engines that match regexes with worst-case linear time guarantees, in exchange for a reduced feature set (no backreferences nor lookarounds [RE2 2017]) [Gallant 2023; Toub 2022]. Interest in these linear engines is causing a paradigm shift: an increasing number of platforms and languages are now either secure-by-default with linear engines (Rust [Gallant 2014] and Go, and any language linking to the popular RE2 library), or at least offer the possibility to switch to a linear engine for a subset of regexes (.NET [Moseley et al. 2023]). Our work demonstrates that this same paradigm shift is applicable to JavaScript, and that the expressivity tradeoff is less dire than previously believed.

Unfortunately, even in the new, safer world, the word *linear* still hides a lot of variability: some engines aim for linearity in just the input string length (e.g. .NET, so that performing matching in $O\left(|r|^2 \times |s|\right)$ is acceptable), whereas others attempt to guarantee linear execution in both $|r|$ and $|s|$ (e.g. Rust) and are therefore suitable for user-provided regexes or regexes derived from user input (for instance, Google Sheets evaluates user-provided regexes with RE2). Our work achieves both regex- and string-linear performance.

To make matters worse, different regex languages make different semantic design choices when it comes to these nontraditional features [Davis et al. 2019] and regexes written for a language are not always portable to another one: quantifiers (`*`, `+`, `?`) and capture groups have different semantics in JavaScript and Perl, most valid lookarounds in .NET and JavaScript are not valid in Python, Perl or Java (which prevent the use of quantifiers inside lookarounds as in `/(?<=ba*)/`), etc. The impact of these semantic design choices on matching complexity is not well understood.

This problem has several consequences. From a programming language design point of view, the situation is dire. If we ever want to design expressive but secure regex languages, we need a better understanding of the worst-case complexity of each nontraditional feature, *under each semantic design choice*. The situation is no better from a programming language implementation standpoint: even for features and semantics thought to be well understood in practice, our research shows that multiple deployed implementations make invalid algorithmic assumptions. In particular, we found that some features commonly assumed not to be matchable in linear time could in fact be supported by linear engines; that algorithms assumed to be linear were in fact not; and that deployed algorithms assumed to be applicable to modern regex languages led to semantically incorrect results. The following sections provide concrete examples of all three cases.

Our work focuses on the JavaScript regex language; its semantics is specified in the ECMAScript standard by means of a pseudocode backtracking algorithm [ECMA-262 2024]. Javascript supports many nontraditional regex features, including capture groups, backreferences and lookarounds, yet its semantics also make some atypical choices (§3.2) that separate it from other languages and impact matching complexity (§4.5). All JavaScript regexes engines that we could find, except one, use a backtracking implementation strategy [Chromium 2009; DukTape 2013; Hermes 2022; MuJS 2014; QuickJS 2020; WebKit 2018]. The exception is V8 (the JavaScript implementation used in Google Chrome and Node.js), which has two engines: a full-featured backtracking engine called Irregexp [Chromium 2009], and a linear engine with support for almost all of JavaScript regex constructs except backreferences and lookarounds, which we call "V8Linear" [V8 2021], available through a command-line flag.[1] To the best of our knowledge, V8Linear is the only industrial-strength linear-time implementation of a significant subset of JavaScript regexes.

Our work answers the following research questions: Which part of the JavaScript regex language can be matched with linear worst-case time complexity? Can we get linearity in *both* the size of the regex and the size of the string $O(|r| \times |s|)$? To what extent the semantic choices done in JavaScript have an impact on what features can be matched linearly?

Our contributions can be summarized as follows:

- We provide a **novel understanding of JavaScript regex semantic properties**, by identifying a large subset of JavaScript regexes that can be matched in linear time $O(|r| \times |s|)$. We show that JavaScript's semantics for nullable quantifiers (`*`, `+`, `?`) is incompatible with traditional linear-matching techniques, and we present a way to adapt these techniques to achieve linearity. We further show that several JavaScript features (nullable plus and capture groups inside quantifiers) were not implemented linearly in the size of the regex in V8Linear and we present new algorithms to match most of these features linearly.
- We introduce the **first nonbacktracking algorithms for matching lookarounds**, showing that traditional linear engines are too conservative in their feature set. We present an algorithm to match lookbehinds not containing capture groups in linear time that is independent of JavaScript's semantics and could be applied to other regex languages. We then show how to leverage a JavaScript-specific semantic property to match unrestricted lookarounds with capture groups in linear time, albeit with additional memory complexity. To the best of our knowledge, this is the first time that linear algorithms for lookarounds with capture groups have been presented and implemented.

[1]Mozilla recently abandoned the development of the regex engine used in Firefox and replaced it with Irregexp, owing to the complexity of JavaScript regex engines [Ireland 2020].

- We provide **practical implementations for all our algorithms**, showing that they are generally applicable and that their complexity can be validated through experiments. We implemented several of our algorithms in V8Linear,[2] with some having been merged and released in V8 already. We also present OCaml prototype implementations for all of the algorithms presented in this paper, together supporting a very large fragment of JavaScript regexes. This prototype provides an ideal playground for implementation experimentation and we use it to exhibit a novel trade-off between space complexity and time complexity for capture groups.

JS Regex Feature	**V8Linear (state of the art)**	**Our new complexity**	
Nullable quantifiers	incorrect	$\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)$	§ 4.1
Capture Groups in Quantifiers	$\mathcal{O}(\lvert r\rvert^2 \times \lvert s\rvert)$	$\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)$	§ 4.2
Nonnullable Plus	$\mathcal{O}(2^{\lvert r\rvert} \times \lvert s\rvert)$	$\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)$	§ 4.5
Nullable Greedy Plus	$\mathcal{O}(2^{\lvert r\rvert} \times \lvert s\rvert)$	$\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)$	§ 4.5
Captureless Lookbehinds	unsupported	$\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)$	§ 4.4
Unrestricted Lookarounds	unsupported	$\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)^\dagger$	§ 4.3

†: with an additional $\mathcal{O}(\lvert r\rvert \times \lvert s\rvert)$ space complexity

This table summarizes most of our results. Taken together, they show that **a large subset of JavaScript regexes can be matched in linear time**, for both the string and the regex size.

2 LINEAR REGEX ENGINES BACKGROUND

To help position our contributions, this section presents a brief overview of commonly used regex-matching algorithms. Traditional regexes can be represented as nondeterministic finite automata (NFAs), with a mix of ε-transitions and transitions labeled with characters to read from the string. A regex matches a string if there exists a path whose labeled transitions spell out the string in the corresponding NFA. Figure 1 summarizes the traditional Thompson construction [Thompson 1968] for traditional regexes (other constructions exist [Gluškov 1961]). For a traditional regex of size $|r|$ (meaning that its textual representation uses $|r|$ characters), computing the Thompson NFA has $\mathcal{O}(|r|)$ time complexity and produces an NFA with $\mathcal{O}(|r|)$ states and transitions. In the rest of this paper, we refer to a regex and its Thompson NFA interchangeably, so that the *states* of a regex are those of its NFA.

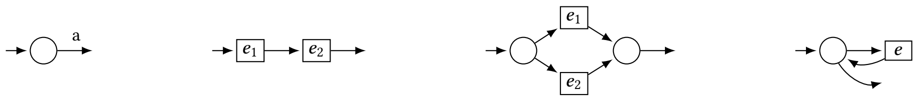
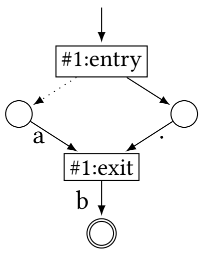


Fig. 1. Recursive Thompson NFA constructions for a, e_1e_2, $e_1|e_2$ and e^*.

This traditional construction can be extended to handle capture groups. First, nodes with multiple outgoing edges are augmented with a notion of edge priority, indicating which path should be considered first. Second, effectful nodes are added to track the string positions at which each capture group is entered and exited. An example of such a *tagged* NFA [Laurikari 2000] is shown in Figure 2 (there and in the rest of this work dotted arrows indicate high priority edges). A path through the `#1:entry` node records at which string position it entered the first capture group.

[2]Nonnullable plus https://chromium-review.googlesource.com/c/v8/v8/+/4778506, Nullable quantifiers https://chromium-review.googlesource.com/c/v8/v8/+/4755530, Lookbehinds https://chromium-review.googlesource.com/c/v8/v8/+/5093860.



```
0: SetReg #1:entry
1: Fork 2 4
2: Consume 'a'
3: Jump 5
4: ConsumeAny
5: SetReg #1:exit
6: Consume 'b'
7: Accept
```

Fig. 2. Tagged NFA and its corresponding bytecode for `/(a|.)b/`. The bytecode representation is explained later in Section 3.3.

Backtracking engines match regular expressions by enumerating all paths of the corresponding NFA in order of priority and returning the first accepting path. This technique can naively be adapted to support all regex features (backreferences, lookarounds…), but it has worst-case exponential time complexity in the size of the string, and the worst case can happen even on regexes using only traditional regular expression features. In the absence of backreferences (for which the matching problem is NP-hard for the size of both the input string and the regex [Aho 1990; Dominus 2000]), and by excluding lookarounds and some other features, linear engines achieve linear-time complexity using the following insight:

Uniform-futures property: Consider two path prefixes of an automaton p_1 and p_2. If they both reach the same regex state while having read the same prefix of the input string, then any extension of p_1 into a complete path is also a valid extension of p_2. In the presence of capture groups, if p_1 has higher priority than p_2, then any extension of p_1 also has higher priority than p_2. In other words, the future of a path prefix only depends on its current regex state and its current string position.

To illustrate this property, consider the regex `/(a+)*b/` executed on the string `"aaa"`. Let p_1 be the partial path matching the first two `"a"` with a single iteration of the star, and p_2 the partial path matching the first two `"a"` with *two* iterations of the star. Despite their diverging pasts, these paths have the same future: each can lead to a match if and only if the second path can also find one, and hence they do not need to be explored separately. A typical backtracking engine will not perform such path merging: it will instead enumerate all possible decompositions of the input string into non-empty sub-strings. Only at the end of each path will it notice that there is no `"b"` character to complete the match, leading to complexity exponential in the number of `"a"`. All linear engines use the uniform-futures property to merge convergent paths. With backreferences, however, this uniform-futures property does not hold. The future of a path may depend on its prefix, because matching a backreference depends on how capture groups were previously matched. As a result, linear engines do not support backreferences. Until this paper, linear engines also excluded lookarounds. We show in Section 4.3 that lookarounds can in fact be matched linearly.

Bit-state backtracking (or memoized backtracking). The most straightforward linear-matching algorithm, bit-state backtracking, simply augments backtracking with a memoization table that records each pair of string position and regex state considered along the search. This table is sufficient to leverage the uniform-futures property: if a path reaches a previously visited pair, it is simply discarded. This approach preserves the usual benefits of backtracking engines (regexes that require little backtracking have excellent performance), but it comes at a significant additional memory cost $O\left(|r| \times |s|\right)$. Consequently, state-of-the-art engines use it only for small regex and strings [Google 2023].

Thompson Simulation and PikeVM. More memory-efficient is *NFA simulation* (Thompson's algorithm [Thompson 1968]), which is used by most linear engines in the common case. It explores the graph breadth-first, ensuring that paths that have consumed the same input prefix are considered at the same time, and leveraging the uniform-futures property to convergent paths. NFA simulation extends to tagged NFAs (Figure 2) to support capture groups [Cox 2009; Pike 1987], in which case the graph and its exploration algorithm are typically translated into a unified byte code, and executed by a so-called *Pike VM* (Section 3.3). This paper describes extensions to the NFA simulation and *PikeVM* algorithms.

Lazy DFA. Finally, for regexes without capture groups, many engines perform *lazy DFA simulation*, a variant of NFA simulation that performs better on average [Google 2023] while maintaining worst-case linear complexity. Lazy DFA performs similarly to NFA simulation but keeps visited states as *sets* instead of lists (order is irrelevant when no groups are present). The result can be viewed as interleaving traditional determinization and matching: it materializes the states that would have been visited by the traditional ahead-of-time DFA lazily, as it matches its input string (ahead-of-time construction has $O\left(2^{|r|}\right)$ time complexity, whereas lazy DFA performs at most $|s|$ transitions, each of which have complexity $|r|$ to compute neighbor states).

3 TECHNICAL BACKGROUND

We now move to our main focus: adapting NFA simulation to match JavaScript regular expressions. This section first presents the JavaScript regex language and its semantics specificities, then describes NFA simulation in detail, laying the groundwork for the presentation of our contributions in Section 4.

3.1 JavaScript Regex Syntax

Regular Expressions:		
$e ::=$	c	Character
$\mid$	`.`	Any Character
$\mid$	ϵ	Empty
$\mid$	$e_1\ e_2$	Concatenation
$\mid$	$e_1 \mid e_2$	Union
$\mid$	$e\ q$	Quantifier
$\mid$	(e)	Capture Group
$\mid$	$(?{:}\,e)$	Noncapturing Group
$\mid$	$(lk\ e)$	Lookaround
Quantifiers:		
$q ::=$	`*`	Greedy Star
$\mid$	`*?`	Lazy Star
$\mid$	`+`	Greedy Plus
Lookarounds:		
$lk ::=$	?=	Positive Lookahead
$\mid$	?!	Negative Lookahead
$\mid$	?<=	Positive Lookbehind
$\mid$	?<!	Negative Lookbehind

Fig. 3. Regex Syntax

This work considers the subset of JavaScript regexes depicted on Figure 3. Greedy operators are also sometimes referred to as *eager*, and lazy operators as *nongreedy*. All our algorithms also support additional features with linear-time guarantees, like anchors (^,$), character classes (`[a-z]`, `\d`, `\w...`), or even arbitrary predicates that only consult the surrounding characters. Counted repetitions are also supported by repeating the sub-expression (just like in state-of-the art linear engines). This makes the regex and the matching time complexity grow with the counters used in counted repetitions.

Notations. To make regexes more readable, we use the notation $(\!|\,\texttt{r}\,|\!)$ for *noncapturing groups*, instead of the usual `(?:r)` JavaScript syntax. Such noncapturing groups are just annotations to make parsing unambiguous, but don't introduce any capture groups to extract. Similarly, we use the character ϵ to denote the empty regex, instead of not writing anything at all (*e.g.* we write `/a|`ϵ`/` instead of `/a|/`). We sometimes annotate each regex capture group as follows: `/a(b)(c)/` gets annotated to `/a(b)`$_{\#1}$`(c)`$_{\#2}$`/`. In JavaScript, each capture group must have a distinct identifier, even for named capture groups. Similarly, we annotate each lookaround with an unique identifier. Identifiers are integers given in a preorder AST traversal. This ensures that if lookaround $\texttt{l}_{\leqslant i}$ contains another lookaround $\texttt{l}_{\leqslant j}$, then $i < j$. For instance, the regex `/(?=a(?<=a))a/` gets annotated to `/(?=`$_{\leqslant 1}$`a(?<=`$_{\leqslant 2}$`a))a/`. We write $\ell(r)$ the total number of lookarounds in a regex r. For instance, $\ell($`/(?=a(?<=a))a/`$) = 2$.

3.2 JavaScript Regex Semantics Peculiarities

While most modern regex languages share a number of nontraditional features, they also present a number of differences [Davis et al. 2019]. These differences are scarcely documented, yet they can have a substantial impact on the semantics or time complexity of regex matching. In this section, we focus on some subtle and notable properties of the JavaScript regex language, that may separate it from other modern regex languages.

Priority The JavaScript backtracking semantics [ECMA-262 2024] explores paths in a given priority order. In an alternation, the left branch has priority over the right one. In a greedy quantifier, the priority is to iterate as much as possible, while a lazy quantifier tries to iterate as few times as possible. JavaScript regexes are unanchored, meaning that matches can start anywhere in the input string. The match that starts the earliest in the input string has the most priority. In practice, engines add the prefix `.*?` to the regex they are executing to find that match.

Capturing Lookarounds[ECMA-262 2024, 22.2.2.4]. Lookarounds in JavaScript are more than assertions. Lookahead and lookbehinds can define capture groups that the engine should return. For instance, `/(?=(c)`$_{\#1}$`)/` on `"c"` returns that capture group #1 is set to `"c"`. However, as in most modern regex languages with them, negative lookarounds cannot define any capture groups.

Unbounded Lookarounds[ECMA-262 2024, 22.2.1:Assertion] Some languages (*e.g.* Perl, Python, Java) only allow fixed-width lookarounds. Meaning that regular expression patterns inside lookaheads and lookbehinds should not contain unbounded quantifiers like star and plus. In JavaScript, there is no such restriction and `/(?=a*)/` is a valid regex. Fixed-width lookarounds are not much harder to implement than anchors, but unbounded ones are more expressive and complex.

Capture Reset[ECMA-262 2024, 22.2.2.3.1, step 4.] When entering a quantifier, the value of the capture groups defined inside that quantifier are reset to `undefined`. For instance, matching `/((a)`$_{\#2}$`|(b)`$_{\#3}$`)`$_{\#1}$`*/` on string `"ab"` will return a match where group #2 is set to undefined. On the first star iteration, #2 is set to the range `0-1`, matching the first character of the string. When executing the second star iteration, #2 is reset to undefined, and group #3 is set to the range `1-2`, matching character `"b"`. This property is specific to the JavaScript regex language. We show that, while difficult to implement in linear time (see Section 4.2), this property helps implement other features in linear time (see Sections 4.5.3 and 4.3).

Nullable Quantifiers[ECMA-262 2024, 22.2.2.3.1, step 2.b] Quantifiers can have both mandatory and optional iterations. For instance, the plus has one mandatory iterations, followed by any number of optional ones. The star has no mandatory iterations, but can have any number of of optional ones. In JavaScript, optional repetitions of a quantifier cannot match the empty string. This prevents the backtracking implementation from executing an infinite loop when matching a nullable star for instance. This semantics for nullable quantifiers is specific to JavaScript. Other regex languages typically have different semantics for nullable quantifiers that is discussed in Section 4.1.

Usual NFA Simulation instructions:

`Consume` c	Consumes character c.
`ConsumeAny`	Consumes any character.
`Jump` l	Jumps to label l.
`Fork` l_1 l_2	Creates a new thread. l_1 has higher priority.
`Accept`	A match is found.
`SetReg` *reg*	Writes current string position to register *reg*.

New instructions introduced in this work:

`BeginLoop`	§ 4.1
`EndLoop`	§ 4.1
`SetQuant` q	§ 4.2
`WriteOracle` l	§ 4.3
`CheckOracle` l	§ 4.3
`NegCheckOracle` l	§ 4.3
`WriteLB` b	§ 4.4
`CheckLB` b	§ 4.4
`NegCheckLB` b	§ 4.4
`SetNullPlus` q	§ 4.5
`CheckNull` q	§ 4.5

Fig. 4. NFA Simulation Bytecode Instructions

3.3 NFA Simulation Engines

Algorithm 1: Simulation

```
for i=0 to str.length do
    while active≠[] do
        t = active.top()
        if processed[t.pc] then
            active.pop()
            continue
        end
        processed[t.pc] = true
        match bytecode[t.pc] with
            case Consume c ⇒
                if c = str[i] then
                    t.pc = t.pc+1
                    next.push(t)
                end
                active.pop()
            case Jump l1 ⇒
                t.pc = l1
            case Fork l1 l2 ⇒
                t.pc = l2
                t' = t.copy()
                t'.pc = l1
                active.push(t')
            case Accept ⇒
                bestmatch = t
                active = []
            case SetReg r ⇒
                t.regs[r] = i
                t.pc = t.pc+1
        end
    end
    active = next.reverse()
    next = []
    processed.fill(false)
end
return bestmatch
```

The NFA simulation algorithm is a well-established way to avoid the exponential cost of determinization, used in V8Linear, RE2, Rust and Go. It is common to represent the tagged NFA with a bytecode. This bytecode corresponds to an array of bytecode instructions depicted on Figure 4, each associated with a label.
A simulation engine (as shown on Algorithm 1) maintains a list of threads ordered by priority, `active` (a LIFO list). Each thread contains a program counter `pc` and a set of registers `regs`. Threads represent incomplete paths of the NFA synchronized at the same string position `i`. Initially, `active` contains a single thread with `pc` 0.
The algorithm goes through the string one character at a time (the **for** loop). To compute the `next` list of threads for the next string position, it follows transitions of the NFA, starting with the highest priority thread, using `active.top()`. When a thread reaches a `Consume` instruction that corresponds to the next character, it is pushed into `next`. New threads may be created with the `Fork` instruction. If a thread reaches an `Accept` instruction, it is stored as being the best possible match found so far. Lower priority threads are discarded, but the algorithm keeps running with higher-priority threads in `next`.
The algorithm also maintains a `processed` array indicating which bytecode instruction has already been executed at this step. Any thread that reaches a bytecode instruction already in that array (when `processed[t.pc]` is true) is discarded with `active.pop()` (using the uniform-futures property of section 2). Thanks to this array, each bytecode instruction can be executed at most once for each string position.

Example. Consider the regex `/(a|.)b/` (whose bytecode is shown in Figure 2), with string `"ab"`. Initially, `active` contains a single thread with `pc` 0. After a first iteration for the first character, the `active` list contains a thread at `pc` 3, and another at `pc` 5. During the second iteration, the second thread gets killed, as the first one executes the instruction at `pc` 5 first. After reading the last character, a single thread reaches the `Accept` instruction. Using its register values, the capture group #1 is known to contain the sub-string delimited by indices 0 and 1, indicating the first letter `"a"`. Note that on the string `"ac"`, the simulation engine would only check once if the thread in state 6 can accept the second character, and immediately conclude that there is no match. A backtracking implementation would check it twice: one for each path that consumed the first character.

Complexity. For a regex r, the size of the generated bytecode and its number of epsilon transitions grow linearly with the size of the regex $O(|r|)$. Then, at each string position, the simulation executes each bytecode instruction at most once, and follows each transition at most once. It follows that, for a regex r and a string s, this executes in the worst case $O(|r| \times |s|)$ bytecode instructions. All of these instructions, except `Fork`, are trivially implemented in $O(1)$ time complexity. The case of `Fork` and the space complexity are discussed later in Section 4.6.

4 MATCHING JAVASCRIPT REGEXES WITH LINEAR-TIME GUARANTEES

In this section, we present six different algorithms to match different features of the JavaScript regex language with linear time guarantees. We first show in Section 4.1 that the Nullable Quantifier semantic property of JavaScript is incompatible with a traditional NFA simulation engine, but present an extension that solves the issue while retaining linear complexity. We then present in Section 4.2 an algorithm to implement the Capture Reset semantic property in linear time, while this previously introduced regex-quadratic time complexity in other implementations. Next, we introduce previously unsupported JavaScript features in a linear engine. Section 4.3 presents an algorithm for unrestricted lookarounds, while Section 4.4 presents a streaming algorithm for lookbehinds without capture groups. We then present algorithms for linear matching of any nonnullable or greedy JavaScript plus (Section 4.5). Finally, we exhibit a novel space and time complexity tradeoff when implementing capture group registers with NFA simulation (Section 4.6). Our algorithms are composable (Section 4.7). Taken together, we show that a vast majority of JavaScript regexes without backreferences can be matched in $\mathcal{O}(|r| \times |s|)$ worst-case time complexity.

4.1 Matching JavaScript Nullable Quantifiers in a Linear Engine

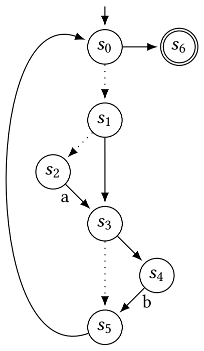


Fig. 5. Usual NFA for `/⦇⦇a|ϵ⦈⦇ϵ|b⦈⦈*/`

Among all regex languages, JavaScript has a unique semantics when it comes to matching nullable quantifiers (`*` or `+` for instance). Surprisingly, the techniques and the uniform-futures property presented in section 2 do not obey these semantics. As a result, the V8Linear engine (using NFA simulation) was incorrect and would sometimes return a different result than specified. However, we present a way to adapt these algorithms without changing their asymptotic complexity.

The NFA simulation algorithm cannot visit the same regex state twice without consuming any character in the string (see the `processed` array of Algorithm 1). In JavaScript however, it is possible to visit the same regex state twice without consuming, as long as this state does not mark the beginning of a quantifier. Consider for instance matching the regex `/⦇⦇a|ϵ⦈⦇ϵ|b⦈⦈*/` on the string `"ab"` (its NFA is represented on Figure 5). The top priority result of an usual NFA simulation is to match only `"a"` in a single star iteration. Doing a second iteration of the star is invalid, since it would visit the same regex state (s_3) without having consumed anything from the string yet.

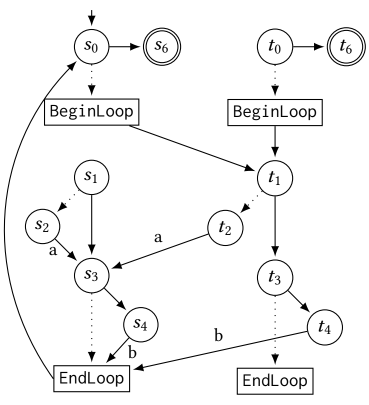


Fig. 6. Fixed NFA for `⦇⦇a|ϵ⦈⦇ϵ|b⦈⦈*`

In JavaScript, the correct result is to match the entire string `"ab"` in two iterations of the star. The first iteration matches `"a"` in the first alternation and the empty string in the second alternation. The second iteration matches ϵ then `"b"`. Both of these iterations of the star have matched a nonempty part of the string. In JavaScript, the future of a path prefix depends not only on the current string position and the current regex state, but also on what has been consumed in the current quantifier. Going from s_3 to s_3 without consuming characters in Figure 5 is only possible when the current execution has consumed `"a"` in s_2 before reaching s_3.

To adapt the NFA simulation algorithm to JavaScript semantics, we observe that, given a regex state and a string position, there are at most two possible behaviors depending on whether the current path is allowed to exit a quantifier without consuming a character. To materialize these two behaviors, we duplicate the states of the regex, as shown on Figure 6. The sub-automaton on the left of the Figure contains states s_i that are allowed to exit any quantifier without consuming a character, because they already consumed a character in all of their parent quantifiers. States t_i of the right sub-automaton however are states that should not be allowed to exit their current innermost quantifier without consuming. A thread that just began a quantifier iteration should not be able to exit this iteration just yet. Consequently, at the beginning of each quantifier, we insert a new `BeginLoop` bytecode instruction, which switches the execution to the automaton on the right. When consuming a character, transitions always point back to the automaton on the left. Even if a thread was inside several nested quantifiers, all of them just consumed a character and are then allowed to conclude their current iteration. At the end of each quantifier, we insert a new `EndLoop` instruction. In the left automaton, this behaves like a `Jump`, but on the right automaton this is a blocking state.

```
case Consume c ⇒
    if c=str[i] then
        t.pc = t.pc+1
        t.left = true
        next.push(t)
    end
    active.pop()
case BeginLoop ⇒
    t.left = false
    t.pc = t.pc+1
case EndLoop ⇒
    if t.left then
    | t.pc = t.pc+1
    else
    | active.pop()
    end
```

Fig. 7. Executing `BeginLoop`/`EndLoop`

Correctness. Regex quantifiers are well-parenthesized, meaning that it's never possible for a thread to be allowed to exit its innermost quantifier but not an outermost one. As a result, no matter the number of nested quantifiers, two copies of the original automaton are enough to capture all the behaviors of the JavaScript quantifier semantics.

We reported this semantic mismatch, then implemented and merged our solution in V8Linear. In practice in an NFA simulation implementation, one does not need to actually duplicate the bytecode. Instead, threads are augmented with a boolean indicating in which automaton they currently belong. `BeginLoop`, `EndLoop` and `Consume` instructions each modify or read this boolean. An extension of Algorithm 1 is shown on Figure 7, where this boolean is called `left`. When the regex inside a quantifier is not nullable (cannot match the empty string), there is no need to insert `BeginLoop` and `EndLoop` instructions. A simple analysis to determine if a regex is non nullable is shown later in Section 4.5. We illustrated our solution on NFA simulation, but the same insight can be used for a bit-state backtracker or a Lazy DFA matcher. This also generalizes to counted repetition, where V8Linear used to return incorrect results. For instance, in `/⦇⦇a|ϵ⦈⦇ϵ|b⦈⦈{0,7}/`, the optional repetitions are not allowed to match the empty string. It suffices to wrap the bytecode of each optional repetition with `BeginLoop` and `EndLoop` instructions.

4.2 Linear Matching of the Capture Reset Property

The Capture Reset property introduced in section 3.2 is unique to the JavaScript regex language. In this section we show how the solution used in V8Linear has quadratic complexity in the size of the regex, and we present a new linear algorithm for the Capture Reset property.

The previous quadratic algorithm. The intuitive solution used in V8Linear defines a new bytecode instruction, `ClearReg` *reg*, which clears the value of a capture register, setting the corresponding group to `undefined`. Such instructions are inserted at the beginning of each quantifier, for each capture group defined inside that quantifier. For instance, Figure 8 shows the bytecode instructions generated for `/⦇⦇(a)|b⦈*|c⦈*/`. Here we omit the `BeginLoop` and `EndLoop` instructions of section 4.1 to explain both properties independently. When a capture group is defined inside several quantifiers, one needs to clear its capture registers at the beginning of each of these quantifiers. For instance, when matching `/⦇⦇(a)|b⦈*|c⦈*/` on `"ac"`, the first `ClearReg` instruction is needed to clear the capture group as we enter the outer star a second time. When matching the same regex on `"ab"`, the second `ClearReg` instruction is needed to clear the value of the group as we enter the inner star a second time. As a result, each capture group might need as many bytecode instructions as the number of quantifiers above in the regex AST. Consider the following family of regexes:

$r_0 =$ `/./` and $r_{n+1} =$ `/(`r_n`)*/`. While its size $|r_n|$ grows linearly in $O(n)$, its bytecode size grows quadratically in n (with $O(n^2)$ complexity).

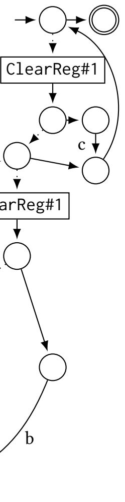


Fig. 8. `/⦇⦇(a)|b⦈*|c⦈*/` NFA with `ClearReg` instructions.

Our linear algorithm. We now present our solution to implement the Capture Reset property linearly in the size of the regex with an NFA simulation. In essence, it consists in not clearing the capture group values during the execution of the bytecode for all threads, but instead after a match is found, only for the winning thread. Consider the regex `/⦇⦇(a)|b⦈*|c⦈*/` again. If a match is found with some value for the capture group, there are only two reasons for which we would like to clear its value: either the inner star or the outer star were entered again after the capture group was set. If we know when was the last time each quantifier and each group were entered we can filter accordingly, keeping the capture value only if it was defined later than the times both quantifiers were entered. To define this notion of time, we extend the NFA simulation engine with a global *clock*, an integer value starting at 0 and increasing each time the simulation executes any bytecode instruction. We show on Figure 10 how to extend Algorithm 1, where `clk` is that clock. We also extend the threads memory with some new registers containing clock values. One new register for each group (in `gclocks`), and one for each quantifier (in `qclocks`). Whenever the NFA simulation executes a `SetReg` instruction corresponding to the entry of a capture group, it records both the current string position and the current clock value. We also add a new `SetQuant` q instruction, inserted at the beginning of each quantifier body, that records the clock value for quantifier q. As a result, the size of the bytecode corresponding to a quantifier is now constant and does not depend on how many capture groups are defined inside. This requires more registers for each thread, but as both the number of groups and the number of quantifiers are bounded by $|r|$, this memory still grows linearly with the size of the regex.

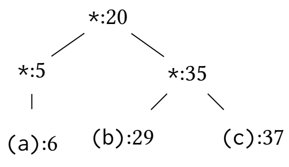


Fig. 9. Simplified AST for `/⦇(a)*|⦇(b)|(c)⦈*⦈*/` with clock values of the winning thread for "abc".

Filtering the capture groups after a match is found can be done with time complexity $O(|r|)$, with an AST traversal of r. For instance, consider the regex `/⦇(a)*|⦇(b)|(c)⦈*⦈*/` and the string `"abc"`. Figure 9 represents a simplified AST of the regex, where we wrote the final clock values of each group and quantifier in the winning thread. We start with the root of the AST, and compare its value with both its children. Because the inner left star has a smaller clock value than the outer star, all capture groups inside are cleared. We then recursively consider the inner right star and compare its last clock value to its children. One of the groups, `(b)`, has a smaller clock value, meaning that it was not defined in the last iteration of its parent star and is cleared. Finally, we keep the value of the last group `(c)`.

Correctness. Our new NFA construction does not change the paths explored by the threads, it only replaces some `ClearReg` instructions with a single `SetQuant` instructions. The correctness of the new filtering algorithm relies on the following informal invariant: at any moment during the NFA simulation, for any thread t, any group g inside a quantifier q, the clock value of g in t is strictly greater than the clock value of q if and only if the capture value of g

was defined in the last iteration of q in t. This invariant holds even when g is inside several nested quantifiers.

```
case SetReg r ⇒
  t.regs[r] = i
  t.gclocks[r] = clk
  t.pc = t.pc+1
case SetQuant ⇒
  t.qclocks[r] = clk
  t.pc = t.pc+1
```

Fig. 10. Updating clocks

The SetQuant instructions can be omitted for quantifiers with no capture group inside, ensuring that our new bytecode generation is always smaller than the previous solution with ClearReg instructions. While we present this solution on an NFA simulation engine, the same idea could be applied to a backtracking or a bit-state backtracking implementation, as both may also spend a quadratic amount of time clearing capture registers if they follow the implementation described in the JavaScript semantics. Note however that in a backtracking implementation supporting backreferences, groups that may be backreferenced need to be cleared dynamically, so that the backreference matches the correct value (in JavaScript, a backreference to an undefined group matches the empty string).

4.3 Unrestricted JavaScript Lookarounds in Linear Time

We now present an algorithm to match all JavaScript lookarounds (both lookaheads and lookbehinds, both positive and negative) in linear time. JavaScript lookarounds have two roles: filtering matches and defining capture groups. Our algorithm handles these roles separately in different phases. First, lookarounds act as assertions filtering matches. For this aspect, we show that we can precompute an *oracle* boolean truth table, indicating each string position at which each lookaround holds. We show that constructing this oracle can be done in $O(|r| \times |s|)$ complexity. We then match the main expression, simply consulting the oracle when a path reaches a lookaround. Finally, we reconstruct missing capture groups defined inside lookarounds in a third phase. Thanks to the Capture Reset property of JavaScript, we show that we can also do that in $O(|r| \times |s|)$ complexity. An example of executing all phases of our algorithm is provided in the supplemental material. This algorithm comes with an additional space complexity of $O(\ell(r) \times |s|)$ for the oracle where $\ell(r) \leq |r|$ is the number of lookarounds in r. We will show in Section 4.4 how to avoid this space complexity in the particular case of captureless lookbehinds.

4.3.1 First Phase: Building the Oracle. We define the *intrinsic size* of a regex r, noted $||r||$, the size of its textual represenation without descending recursively inside lookarounds. We get the following **intrinsic equality**, expressing that the full size of a regex is equal to the sum of its intrinsic size and the intrinsic sizes of all its lookarounds:$\forall r, |r| = ||r|| + \sum_{i=1}^{\ell(r)} ||r_{\leqslant i}||$.

To build each row of the oracle, we present a modification of the NFA simulation algorithm that allows to find in $O(||r_{\leqslant i}|| \times |s|)$ all the places in s where $r_{\leqslant i}$ matches. In essence, we match $\mathtt{r}_{\leqslant i}$ *in reverse* and modify the Accept instruction.

Observation 1: Replacing the final Accept instruction of an NFA by an instruction that writes the current string position and does not discard lower priority threads allows the NFA simulation algorithm to find all positions at which a match can *end*. We consequently define the instruction WriteOracle i which writes to the oracle.

Observation 2: Define the reverse of $\mathtt{r}_{\leqslant i}$, $rev(\mathtt{r}_{\leqslant i})$ by recursively inverting the two subexpressions of each concatenation in $\mathtt{r}_{\leqslant i}$. Matching /.*?$rev(\mathtt{r}_{\leqslant i})$/ on the reverse of a string s can find all positions where a match of $\mathtt{r}_{\leqslant i}$ in s could begin. Consequently, we extend the NFA simulation algorithm by allowing it to read the input string backward, starting with the last character of the string, then moving toward its beginning. We use the standard forward direction for lookbehinds, and the backward direction for lookaheads that we also reverse.

Using these observations, we can compute each string positions at which each lookaround hold (*i.e.* the positions at which a match can begin). We construct the oracle one row at a time, starting with lookarounds with the highest indices (the deepest in the AST). For nested lookarounds, the row of the inner lookarounds gets computed before its parent. The parent can then replace its inner lookaround $\mathtt{r}_{\leqslant i}$ by a single `CheckOracle i` instruction, which checks the oracle table at the current string position. Doing so, we build the entire row i of the oracle table with a single match of $\mathtt{r}_{\leqslant i}$ on the string.[3] The complexity of building the entire table is then $O\left(\sum_{i=1}^{\ell(r)} ||r_{\leqslant i}|| \times |s|\right)$, bounded by $O\left(|r| \times |s|\right)$ using the intrinsic equality.

4.3.2 Second Phase: Matching the main expression. Once we know the positions at which each lookaround holds, we can simply run the main expression in a forward direction. Whenever a thread encounters a lookaround, it simply accesses the oracle with a `CheckOracle` instruction. When the NFA simulation executes this instruction, it also records in each thread the string position at which each lookaround was last used. This means defining new thread registers, one for each of $O\left(|r|\right)$ many lookarounds, just as the ones used for capture groups (no change to the asymptotic complexity). This phase has time complexity $O\left(||r|| \times |s|\right)$. When a match is found, this phase returns both the register values for all capture groups defined inside the main expression, and the last string position each lookaround was used. We reconstruct the values of capture groups defined inside positive lookarounds in the third phase.

4.3.3 Third Phase: Reconstructing Missing Capture Groups. Finally, we run an NFA simulation for each outermost lookaround that was used to produce the main match. This time, this simulation is run forward for lookaheads and backward for lookbehinds (that we reverse), to comply with the capture semantics inside lookarounds. We start the simulation exactly at the input string position where the lookaround was used. If lookarounds are inside quantifiers, they may have been used several times in a match. However, the Capture Reset property ensures that only its last visit can define the capture groups inside. As we run the simulation for an outer lookaround, it may require going through an inner lookaround. In that case, we check the oracle with a single `CheckOracle` instruction, mark the inner lookaround and execute it later. Winning threads of these new simulations define the missing capture group values. In the worst-case, each lookaround needs to be executed once in this phase, which has time complexity $O\left(\sum_{i=1}^{\ell(r)} ||r_{\leqslant i}|| \times |s|\right)$, or $O\left(|r| \times |s|\right)$ using the intrinsic equality.

```
case WriteOracle l ⇒
  oracle[l][i] = true
  active.pop()
case CheckOracle l ⇒
  if oracle[l][i] then
    t.pc = t.pc + 1
  else
    active.pop()
  end
```

Fig. 11. Oracle

Figure 11 shows how to augment Algorithm 1 to support the new instructions `WriteOracle` and `CheckOracle`. Note that, unlike an `Accept`, `WriteOracle` does not kill lower priority treads but only the current one (and writes to the oracle), in accordance with Observation 1.

Correctness. Observation 1 follows from the uniform-futures property, and Observation 2 can be proved by induction on the regex. The correctness of the first two phases directly follows from Observations 1 and 2 and the correctness of the standard NFA simulation algorithm. The third phase follows from the Capture Reset property: capture groups inside lookarounds can only be defined by the last iteration of this lookaround in the winning thread.

4.4 Matching Captureless Lookbehinds with an NFA Simulation

The previous algorithm of Section 4.3 requires precomputing the entire oracle table. For lookbehinds without capture groups inside, this is not needed. In this section, we present a novel separate *streaming* algorithm to extend the NFA simulation algorithm to handle captureless lookbehinds.In

[3]As an optimization, note that capture groups do not matter in that step, and we can freely remove them from each lookaround subexpression. We can even use a Lazy DFA matcher instead of the NFA simulation for faster matching.

particular, all negative lookbehinds are supported, as they cannot define capture groups (see Section 3.2). Compared to the previous section, this algorithm needs no additional space complexity. It does not use the Capture Reset property and is thus applicable to any regex language, not just JavaScript. This algorithm supports nested lookbehinds and capture groups outside of lookbehinds.

```
case WriteLB b ⇒
    LBtable[b] = true
    active.pop()
case CheckLB b ⇒
    if LBtable[b] then
        t.pc = t.pc + 1
    else
        active.pop()
    end
```

Fig. 12. LBtable instructions

Unlike lookaheads, deciding whether a lookbehind is satisfied only depends on the part of the string that has already been read. This suggests that we can merge together the first two steps of the algorithm in Section 4.3, building the oracle as we match the main expression. As we synchronize both steps, it turns out that only the oracle column corresponding to the current string position is used. This new algorithm leverages this observation. It maintains an array of booleans, `LBtable` of length $\ell(\text{r})$. This `LBtable` corresponds to the column of the oracle of Section 4.3 of the current string position. We show below how an NFA simulation engine can update this array in such a way that, at string position i, for a lookbehind b, `LBtable[`b`]` contains 1 if and only if the lookbehind $r_{\lessgtr b}$ holds at position i. When a thread encounters a lookbehind, one then simply needs to check the corresponding entry in `LBtable`. We implement this with a new bytecode instruction, `CheckLB b`, killing the current thread if `LBtable[b]` is 0. For negative lookbehinds, we use a similar `NegCheckLB b` instruction. Figure 12 shows how to augment Algorithm 1 for these new instructions. At each new character, the `LBtable` is also reset to an array of `false`. Compared to Figure 11, the two instructions do not use the current string position `i`, since the `LBtable` corresponds only to the current column of the `oracle`.

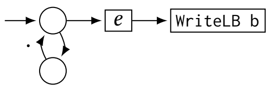


Fig. 13. NFA of `(?<=`$_{\lessgtr b}$`e)`

To update this `LBtable`, it suffices to compile each lookbehind separately and run them in lockstep with the main regex. Each of these lookbehind automata ends with a new `WriteLB b` bytecode instruction, writing to `LBtable` that the lookbehind b is satisfied at the current string position. Each lookbehind automaton also starts with a `.*?` prefix, allowing the match to begin at any character. Figure 13 shows the NFAs generated for captureless lookbehinds; since thread priority is irrelevant in these automata, we do not use dashed arrows. Each time the simulation reads a new character, the contents of `LBtable` are reset to 0.[4]

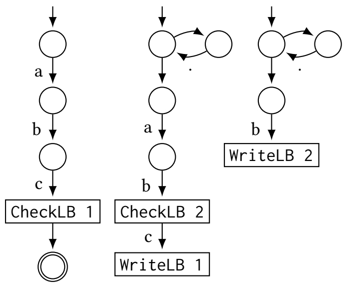


Fig. 14. Lookbehind automata for `/abc(?<=ab(?<=b)c)/`

To run these multiple automata in lockstep, it suffices to add their initial state to the list of initial states of the NFA simulation.[5] This list of initial states is ordered such that the initial state of a lookbehind comes before the initial state of a parent lookbehind in the regex AST, to ensure that each `WriteLB` instruction is executed before the corresponding `CheckLB` instruction. As an example, consider the regex `/abc(?<=ab(?<=b)c)/`. Its NFA is shown on Figure 14. On the string `"abc"`, the NFA simulation will start executing all three automata, starting with the one on the right. After reading `"a"`, none of the automata have yet reached a final state, and `LBtable` contains only 0s. This is expected, since none of the lookbehinds hold after reading a single `"a"`. After reading `"b"` however, the automaton on the right reaches the

[4]One can even avoid resetting by storing the last position where each lookbehind was satisfied instead of booleans.
[5]Alternatively, since the lookbehinds considered in this algorithm do not contain groups, one could also run these lookbehind automata with a Lazy DFA matcher in lockstep with an NFA simulation.

`WriteLB 2` instruction, and writes to `LBtable`. Just after this write, the middle automaton reaches the `CheckLB 2` instruction, and since `LBtable[2]` was just written to, this thread is kept alive. Similarly after reading `"c"`, the middle automaton will write to the table and the main automaton on the left will finally reach an accepting state, indicating that a match was found.

`LBtable` has size $\ell(r)$, bounded by $O(|r|)$. Since each lookbehind is compiled exactly once, the total generated bytecode has size $O(|r|)$. Each `WriteLB` and `CheckLB` instruction can be executed with $O(1)$ complexity, and resetting `LBtable` at each new character has a total time complexity of $O(|r| \times |s|)$, leaving the total complexity of the NFA simulation unchanged.

Correctness. This algorithm follows from Observation 1 and the following property: for any lookbehind `/(?<=r)/`, any string s and index i, there exists a match of `/.*?r/` on s ending at position i if and only if `/(?<=r)/` holds in string s at position i.

4.5 Linear Matching of the JavaScript Plus

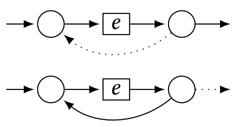


Fig. 15. Usual NFA for e+ and e+? in linear engines.

The traditional construction for a regex plus (as shown in Figure 15 and used in RE2 or Rust) generates linear-sized NFAs. Unfortunately, this construction does not implement the JavaScript plus semantics. To see where this construction fails, consider the regex `/⦇ϵ|.⦈+/` on string `"a"`. In JavaScript, an engine returns a match on the whole string, doing a first iteration matching nothing then another matching `"a"`. With the usual construction of Figure 15 however, an NFA simulation's highest priority match does a single iteration matching only the empty string. Doing another iteration is invalid, since it would go back to the beginning of the plus without having consumed any character. While this is reminiscent of the problem solved in Section 4.1, wrapping the body in `BeginLoop` and `EndLoop` instructions would not be correct, as it would prevent a plus from ever matching the empty string. As seen in Section 3.2, the first mandatory repetition of the plus can match the empty string, but not the following ones. To reflect this difference, the solution implemented by V8Linear was simply to expand `/e+/` into `/ee*/` and `/e+?/` into `/ee*?/`. However, this leads to regex-exponential complexity: for instance, `/⦇⦇a+⦈+⦈+/` gets expanded into `/⦇⦇aa*⦈+⦈+/`, then `/⦇⦇aa*⦈⦇aa*⦈*⦈+/` and `/⦇⦇aa*⦈⦇aa*⦈*⦈⦇⦇aa*⦈⦇aa*⦈*⦈*/`. To prevent exponential explosion, V8Linear used to simply reject regexes with too many nested plusses. In this section, we present two new constructions to implement any nonnullable or greedy plus without such bytecode duplication and in linear time.

$$\begin{gathered}
\text{null}(.) = \text{null}(c) = \textbf{NN} \\
\text{null}(\epsilon) = \textbf{CIN} \\
\text{null}(e_1\ e_2) = \max(\text{null}(e_1), \text{null}(e_2)) \\
\text{null}(e_1|e_2) = \min(\text{null}(e_1), \text{null}(e_2)) \\
\text{null}(e*) = \text{null}(e*?) = \textbf{CIN} \\
\text{null}(e+) = \text{null}(e+?) = \text{null}(e) \\
\text{null}((e)) = \text{null}(e) \\
\text{null}(lk\ e) = \textbf{CDN}
\end{gathered}$$

Using the order **NN** > **CDN** > **CIN**.

Fig. 16. Nullability analysis

4.5.1 Nullability. We now define three notions of nullability. A regex can be nonnullable (or NN), meaning that it cannot ever match the empty string (*e.g.* `/a/`). Otherwise, we define *Context-Independent Nullable* (CIN) and *Context-Dependent Nullable* (CDN) regexes. A CIN can always match the empty string (*e.g.* `/a|ϵ/`). A CDN however can match the empty string depending on the surrounding context (when there are lookarounds or if the engine supports anchors like `\b` or `^`). For instance, `/a|(?=b)/` is only nullable when the next character is a `"b"`. Figure 16 presents a syntax-directed nullability analysis. This is an approximation, in the sense that a nonnullable or a CIN can be classified as CDN. As the analysis does not explore lookarounds it does not detect for instance, that `/a|(?=b)(?!b)/` is nonnullable or that `/(?=a)|(?!a)/` is a CIN. This isn't an issue: the solution we present for CDNs is also correct for CINs and nonnullables, albeit more complex.

4.5.2 NonNullable Plus. The simplest case happens when compiling `/e+/` or `/e+?/` where `e` is nonnullable. In that case, it turns out that the usual NFA construction shown in Figure 15 correctly implements the JavaScript semantics. Since `e` cannot match the empty string, it does not matter that the first repetition is allowed to match it while the others are not. We have implemented and merged this case in V8Linear. For counted repetitions `/e{n,}/`, when $n > 0$ and `e` is nonnullable we can similarly avoid one repetition of the bytecode of `e` by replicating it $n-1$ times, followed by Figure 15, instead of the usual n repetitions followed by `e*`.

4.5.3 CIN and CDN Plus. We now present a way to match nullable greedy plusses linearly. Matching nullable *lazy* plusses in linear time remains an open problem, although we only found them in 0.003% of regexes (see Section 5.1). Our solution leverages the following observation: the only way to match the empty string with a greedy nullable JavaScript plus is to do a single iteration, and this has the *lowest priority*. For instance, consider the regex `/⦇a|ϵ|b⦈+/`, a CIN. First, the middle branch of the plus body is only allowed in the first iteration of the plus. So we can expand and rewrite this regex as `/⦇a|ϵ|b⦈⦇a|b⦈*/`, removing the empty path from the star. This regex can only match the empty string if it skips the star, but doing so has the lowest priority since the star is greedy. Even if the string starts with a `"b"`, then matching empty in the first iteration then consuming `"b"` in the greedy star has higher priority than just consuming empty in the first iteration and skipping the star. We can then further rewrite the regex to `/⦇a|b⦈⦇a|b⦈*|ϵ/`, or even `/⦇a|b⦈+|ϵ/`, where the plus is now nonnullable.

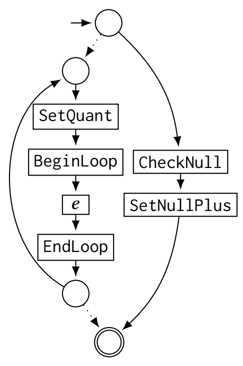


Fig. 17. NFA for a CDN e+

As a result, we compile CINs and CDNs as a `Fork`, where the left branch contains the nonnullable paths, and the right one corresponds to matching the empty string. However, the rewriting transformation in the example above cannot be generalized. First, extracting the nullable path from a regex cannot always be done this easily (consider `/⦇⦇a|ϵ⦈⦇b|ϵ⦈⦈+/`). Second, for CDNs, the nullable path should only be taken when the current string position allows it. Finally, the nullable path of a regex may define capture groups. These groups can either be empty groups (*e.g.* in `/⦇a|(ϵ)|b⦈+/`), or nonempty groups if the regex has lookarounds (*e.g.* the CIN `/⦇a|(?=(c))|ϵ|b⦈+/`).

The first issue can be dealt with by adding `BeginLoop` and `EndLoop` instructions (see Section 4.1) around the body of the plus, effectively removing the nullable paths. The second issue for CDNs can be solved by adding a new `CheckNull` instruction on the nullable path checking that the nulled plus is in fact nullable at that point. We leave implementation details out of this explanation, but computing the nullability of all plusses (even nested) in a regex r for a particular string position can be done with time complexity $\mathcal{O}(|r|)$, for a total added complexity of $\mathcal{O}(|r| \times |s|)$.[6]

Finally, when taking the nulled path of `/e+/`, one needs to set the capture groups defined along the top-priority nullable path in `e` at that string position. This problem is similar to reconstructing capture groups inside lookarounds in Section 4.3. From the Capture Reset property, we know that these capture groups are only defined if the last time the plus was matched, the winning thread went through its nullable path. Like for lookarounds, we record the last string position at which each plus was nulled with a new bytecode instruction `SetNullPlus` inserted along the nullable path. The resulting linear-sized NFA for `/e+/` can be seen on Figure 17. Like a `SetQuant` instruction, this instruction records in the current thread memory the current clock (see Section 4.2), but it also records that the last time this plus was matched, it was nulled and at which string position. After a

[6]For instance, by memoizing checks for the nullability of nested CDN plusses.

winning thread is found, one can reconstruct the missing groups a posteriori. To do so, one starts the simulation again on the body of each nulled plus, at the string position where they were last nulled, without consuming any character of the string. One only needs to run this other simulation once per CIN or CDN of the main expression `r` just like for the lookarounds of the third stage of Section 4.3 but only the empty string. This last phase has a total time complexity of $O(|\texttt{r}|)$,

```
case SetQuant ⇒
    t.qclocks[r] = clk
    t.plusnulled[q] = -1
    t.pc = t.pc+1
case SetNullPlus q ⇒
    t.qclocks[q]=clk
    t.plusnulled[q] = i
    t.pc = t.pc + 1
case CheckNull q ⇒
    if nullable(q,i) then
        t.pc = t.pc + 1
    else
        active.pop()
    end
```

Fig. 18. Nullable plus instructions

Figure 18 shows how to extend Algorithm 1 for these new instructions. Threads contain an additional set of registers, `plusnulled`, indicating for each quantifier `q` if its last iteration consisted in taking the nullable path of a nullable plus (in which case it contains the string position when it last happened), or not (in which case it contains `-1`). These registers are used at the end of the match to figure out which CDN plus needs its capture groups to be reconstructed. We modify the `SetQuant` instruction of Figure 10 so that this register is reset to `-1` when the quantifier is taken without going through the nullable path. The `nullable(q,i)` function checks the nullability of the `q` quantifier at the current string position `i`. When `q` contains lookarounds, this reads from the oracle of Section 4.3.

This algorithm also supports nested CIN or CDN plusses with capture groups inside. Consider for instance the regex $/((\epsilon)+_2)+_1/$ being matched on the string `"a"`. After doing the NFA simulation, a match is found where the winning thread took the nullable path of the outer plus, $+_1$. In this thread, `plusnulled[1]` is `0`, so we reconstruct the capture groups inside $+_1$ at string position `0`. To do so, we interpret $((\epsilon)+_2)$ [7]. Once again, a match is found where the outer capture group has been defined, and now in the winning thread `plusnulled[2]` is `0`. Consequently, we then interpret (ϵ) from string position `0`, defining the inner capture group.

Correctness. The new NFA construction relies on the fact that matching the empty string with a greedy plus has the lowest priority: if at the current position, the regex inside the plus can match any other string s with lower priority, then matching empty in the first iteration then matching s in a second iteration has more priority than doing a single empty iteration since the plus is greedy. The capture group reconstruction algorithm uses the Capture Reset property: any capture group in a plus can only be defined by the last iteration of that plus in the winning thread. The `SetNullPlus` instruction ensures that missing groups are reconstructed.

4.6 A Space-Time Complexity Tradeoff for Capture Groups in an NFA Simulation

	Time	Space
Array	$O(\lvert r\rvert^2 \times \lvert s\rvert)$	$O(\lvert r\rvert^2)$
Linked List	$O(\lvert r\rvert \times \lvert s\rvert)$	$O(\lvert r\rvert \times \lvert s\rvert)$
Balanced Tree	$O(\lvert r\rvert \times log\lvert r\rvert \times \lvert s\rvert)$	$O(\lvert r\rvert^2)$

Fig. 19. NFA simulation complexity using different thread register data-structures.

Recall the execution of the `Fork` instruction in Algorithm 1. As threads contain string indices for each capture group (and the clock values of Section 4.2), this instruction requires copying $O(|r|)$ data in the worst case. All the NFA simulation implementations we found (including Rust, RE2 and V8Linear) use an array for thread registers and copy the array when executing a `Fork`. As a result, the entire simulation execution can have worst-case time complexity $O\left(|r|^2 \times |s|\right)$. In most practical cases where the string is significantly bigger than the regex, this is not an issue. However, in the case of user-provided regexes, this quadraticity can become a security concern. We present two alternative data-structures for thread registers.

[7]For this step, as we reconstruct groups that were taken along a nullable path, we can omit the non-nullable path on the left of Figure 17 for nested CIN/CDN plusses. This ensures linearity: each plus is compiled at most once during this phase.

If one needs strict regex-size linearity, one can store each update done to a thread's registers in an immutable linked list. Storing a new value (`SetReg`) then consists in adding a new cell to the list. Threads can be forked in $O(1)$ time by simply sharing the tail of the list across multiple threads. Threads registers are often written to during execution, but only read from at the very end of the match. Extracting the final value of each register for the winning thread can be done by traversing its list of updates. This comes with an additional space complexity for bigger strings: we know that there can be as many as $O(|r| \times |s|)$ executions of `SetReg` instructions in a match and as a result as many allocations. [8]

If an $O(|r| \times log|r| \times |s|)$ time complexity is acceptable, one can instead store thread registers in an immutable balanced tree. Forking is then achieved in $O(1)$ time. However, executing `SetReg` instructions now requires $O(log(|r|))$ time. The worst-case space complexity of $O\left(|r|^2\right)$ is achieved when $O(|r|)$ threads are alive without sharing any sub-trees.

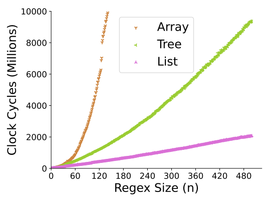


Fig. 20. r_n matching time

We experimentally validated that RE2 and Rust exhibit such regex-quadratic complexity, although Rust limits the maximum number of capture groups in a regex. We implemented these three solutions in our prototype OCaml engine and experimentally validated their time complexities. For instance, we define the following regex family: r_0 =`/⦇(a)?⦈*/`, r_1 =`/⦇(a)?(a)?⦈*/`, r_2 =`/⦇(a)?(a)?(a)?⦈*/` and so on. Figure 20 shows the execution time, measured in CPU clock cycles, of matching r_n on the string "$\texttt{a}^{1000}$". The experimental setup is described in Section 5.3. We believe that regex engines would benefit from simple heuristics, for instance switching to a linked list implementation if the number of capture groups is bigger than the size of the input string.

4.7 Composing Our NFA Simulation Extensions

All the JavaScript regex features showcased on Figure 3 can be supported together with linear time and space guarantees. The algorithms we presented in Sections 4.1, 4.2, 4.3, 4.5 are designed to be composed together. The NFA simulation algorithm 1 can be extended with Figures 7, 10, 11 and 18. Three of these algorithms perform matching in several passes: both the algorithms of Sections 4.3 and 4.5 reconstruct capture groups a posteriori, and the algorithm of Section 4.2 filters capture groups after a match is found. These different phases can be organized as follows. In the first phase of Section 4.3, when building the oracle, there is no need to reconstruct the groups inside plusses since the oracle does not remember any capture group information. Next, in the second phase, we first match the main expression, then reconstruct the groups inside nulled CDN plusses as in the second phase of Section 4.5. Finally in the third phase of Section 4.3, we can match each lookaround subexpression, then reconstruct the capture groups inside each nulled CDN plus. The clock filtering can be done at the very end, once a final match with all captures is found.

Each of the three register implementations of Section 4.6 are compatible with all the algorithms presented in this paper. While the streaming algorithm of Section 4.4 can be composed with the algorithms of Section 4.1 and 4.2, it cannot be composed with the algorithm of Section 4.5 for CDNs if the CDNs contain capture groups. In our CDN algorithm, we need to evaluate the nullability of each CDN plus, and this may depend on knowing if a lookaround holds at a previous string position, for which the full oracle is needed.

[8]This can be seen as a specialization of the persistent array implementation of [Conchon and Filliâtre 2007], where the `Diff` list is the list of register updates and a single *rerooting* is performed at the end for the winning thread.

5 EVALUATION

5.1 Regex Usage Statistics

Nullable Quantifiers	5729	0.37%
Capture in Quantifiers	102283	6.67%
Nonnullable Plus (+ or +?)	405179	26.38%
CIN and CDN greedy +	1041	0.07%
CIN and CDN lazy +?	50	0.003%
Lookarounds	79754	5.19%
Captureless Lookbehinds	22734	1.48%

Fig. 21. Number of regexes using each feature

How often do developers use each of the features for which we presented new algorithms? To answer this question, we parsed and analyzed large corpora of regexes from previous related work [Davis et al. 2018, 2019]. These consists of regexes scrapped online on StackOverflow, RegexLib, NPM or Pypi packages and others. Some of these regexes may be written for other regex languages (like Python), but the syntax is mostly similar for the features that we studied. We parsed them all with a JavaScript regex parser we wrote in OCaml, following the ECMA grammar [ECMA-262 2024] but rejecting unsupported features like backreferences. In total, we parsed and analyzed 1536196 out of 1755587 regexes (87.5%). Figure 21 reports how many regexes include each feature.

Note that not all nullable quantifiers can exhibit the semantic mismatch presented in Section 4.1. For instance, we believe that `/(a?b?)*/` would return the same result in both semantics for any input string. Finding a more precise syntactic characterization of quantifiers that require `BeginLoop`/`EndLoop` instructions is left as future work. For regexes with capture groups inside quantifiers, our algorithm of Section 4.2 reduces the bytecode size. More than a fourth of regexes include a nonnullable plus, for which previous techniques unnecessarily duplicate bytecode. The more elaborate CIN and CDN technique of Section 4.5.3 is only required for a thousand regexes, and only 50 use a nullable lazy plus, for which we haven't found a linear algorithm yet. Around 5% of all regexes use lookarounds of any kind, and more than a fourth of them only use captureless lookbehinds and could be handled by our algorithm without any additional memory complexity.

5.2 Implementation

Each of our algorithms has been implemented in a standalone OCaml NFA simulation engine of around 3.5K lines of code (of which 350 are for parsing), available as an artifact. This engine supports all the features presented here, as well as character classes, backslash sequences, anchors and counted repetition. However regex flags, named groups, Unicode, hexadecimal and octal escapes have not yet been implemented, although they do not represent additional algorithmic complexity. Backreferences are not supported. The three different capture register implementations of Section 4.6 can be used. The engine comes with a differential fuzzer that randomly generates regexes and strings and compares the result to Irregexp. This helped us discover the semantic bug of Section 4.1. Since then, all our algorithms went through hours of fuzzing and millions of tests without reporting any mismatch.

We implemented several of our algorithms in V8Linear. Three of them have been merged into V8Linear: the semantic adaptation for nullable quantifiers (Section 4.1), the linear construction for nonnullable plus (Section 4.5), and the captureless lookbehind algorithm (Section 4.4). These three algorithms have been reviewed by V8 maintainers and tested against various JavaScript test suites, including browser test suites and the official Test262 JavaScript conformance test suite [TC39 2010]. We also contributed some of our own tests to the V8 test suite covering our contributions and documenting previous bugs of V8Linear. We are currently working on implementing the other algorithms, starting with the clock algorithm for Capture Reset (Section 4.2).

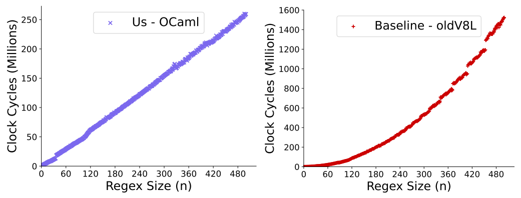


Fig. 22. Capture Reset Complexity

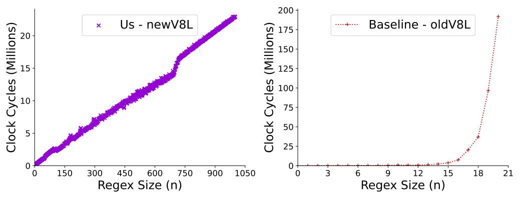


Fig. 23. NonNullable Plus Complexity

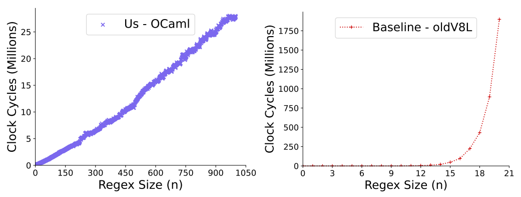


Fig. 24. CDN Complexity

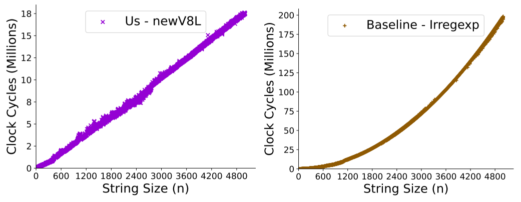


Fig. 25. Captureless Lookbehind Complexity

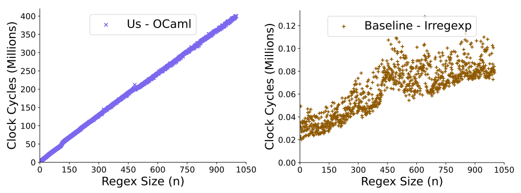


Fig. 26. Lookarounds Regex-Complexity

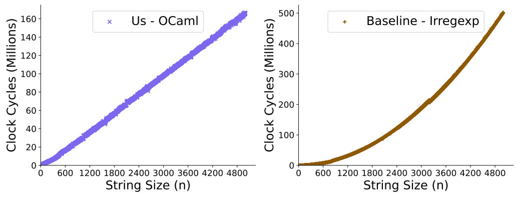


Fig. 27. Lookarounds String Complexity

5.3 Experimental validation of complexity claims

Previous sections provide theoretical arguments to support our algorithmic-complexity claims. Here, we exemplify these claims on specific families of regexes and strings to confirm that, in practice, our implementations verify the following: **(C1)** our capture reset algorithm is linear in $|r|$ when the previous V8Linear algorithm is quadratic; **(C2)** our nonnullable plus construction is linear in $|r|$ when the previous V8Linear algorithm is exponential; **(C3)** our CIN/CDN plus algorithm is linear in $|r|$ when the previous V8Linear algorithm is exponential; **(C4)** our captureless lookbehind algorithm is linear in both $|r|$ and $|s|$ while backtracking is not linear in $|s|$; **(C5)** our unrestricted lookaround algorithm is linear in both $|r|$ and $|s|$ while backtracking is not linear in $|s|$. We compare the following configurations when applicable: [OCaml] our prototype OCaml engine with the linked list implementation of Section 4.6; [oldV8L] V8Linear before our changes; [newV8L] V8Linear after our changes; and [Irregexp] the V8 backtracking engine (with compilation to native code enabled). We modified V8Linear to allow any number of nested plus. For all engines, measurements are done with the `rdtsc` instruction to estimate CPU cycles. We measure 10 repetitions of `match()` after 10 warmup repetitions and take the median of 5 measurements. We do not measure bytecode compilation.

(C1) Capture Reset We evaluate the regex family $r_0 = $ `/a/`, $r_{n+1} = $ `/(`r_n`)*/` on string "$\texttt{a}^{100}$", a case where the bytecode grows quadratically if one uses the `ClearReg` instructions of V8Linear. We compare [OCaml] to [oldV8L] on Figure 22 and confirm that our algorithm behaves linearly. Note that here [oldV8L] suffers from two sources of regex-size quadraticity: the `ClearReg` instructions and the array implementation (Section 4.6).

(C2) NonNullable Plus We consider the regex family $r_0 = $ `/a/`, $r_{n+1} = $ `/`r_n`+/` on string "$\texttt{a}^{100}$". We compare [newV8L] to [oldV8L] on Figure 23 and confirm that the construction of Figure 15 avoids the exponential complexity of [oldV8L].

(C3) Nullable Plus We consider the regex family r_0 = `/a|(^)/`, r_{n+1} = `/`r_n`+/` on string `"b"`. This is a worst case for our algorithm, since each plus is a CDN for which we need to compute the nullability. After a match is found, each plus also needs to be executed again to reconstruct the capture group inside all of them. We compare [OCaml] to [oldV8L] on Figure 24. Even in this worst case, our algorithm exhibits regex-size linearity.

(C4) Captureless Lookbehinds To show linearity in $|s|$, we consider the regex `/b⦇a(?<=ba*)⦈*/` and the string family s_n =`"ba`^{n}`"`. On Figure 25, we compare [newV8L] to the backtracking engine [Irregexp], since [oldV8L] did not support lookbehinds. The backtracking engine exhibits quadratic complexity, since at each string position it tries to match the lookbehind and reads the string backward until the beginning. Our algorithm does a single pass over the entire string.

(C5) Lookarounds To show linearity in $|r|$, we first consider the regex family r_0 = `/(a*)b/`, r_{n+1} = `/a(?=`r_n`)/` on string `"a`^{1000}`b"` on Figure 26. This is a worst case for our algorithm, since each nested lookahead needs to be run in the third step to reconstruct the capture group inside. However, our algorithm still exhibits regex-size linearity, like the backtracking algorithm which does not even need to backrtack in this particular case. To show linearity in $|s|$ in Figure 27, we consider the regex `/c⦇a(?=a*(?<=c(a*))b)⦈*/` on the string family s_n =`"ca`^{n}`b"` and compare [OCaml] to [Irregexp]. Since there are two lookarounds, three passes over the entire string are enough to find a match and our engine exhibits string-size linearity.

6 RELATED WORK

Lookarounds While captureless lookarounds are reminiscent of regex intersection, it has been shown that constructing the NFA for a regex with one or more intersections comes with an exponential size increase in some cases [Gelade and Neven 2012]. Both lookbehinds and lookaheads are commonly known to only be supported by backtracking engines [RE2 2017]. Recent work has managed to integrate lookarounds into a derivative-based engine in .NET7 [Moseley et al. 2023]. When matching a lookaround, their algorithm interrupts the matching of the main expression to start a new engine on the lookaround. Consequently, nesting lookarounds in quantifiers results in polynomial complexity, with the exponent increasing with nesting. Even without lookarounds, their engine achieves linearity in $|s|$ but not in $|r|$. We achieve linearity in both, even in the presence of lookarounds. Another work [Davis et al. 2021] has shown that memoized backtracking engines can support captureless lookaheads in linear time, but with an additional memory overhead $O(|r| \times |s|)$. In contrast, our captureless lookbehind algorithm has no memory overhead. Our unrestricted lookaround algorithm also has a similar space complexity, but it allows captures inside lookarounds for the first time.

Our captureless lookbehind algorithm was inspired by two observations from related work. First, lookaheads can be encoded with Alternating Finite Automata (AFAs) [Berglund et al. 2021]. Second, AFAs can represent LTL formulas and be model-checked linearly if one reads the string (or the LTL *trace*) backwards and reverses the direction of the automaton [Finkbeiner and Sipma 2004]. Reversing the regex and reading the string backwards is usually incompatible with capture priority, but captureless lookbehinds lend themselves well to this treatment. These two observations led to the development of our lookaround matching algorithms and to their first public description, in the issue tracker of V8[9]. Later, related work independently proposed a similar algorithm to match regexes with lookarounds [Mamouras and Chattopadhyay 2024]. In essence, that algorithm is a simplified version of the first two phases of our algorithm in Section 4.3. Just like in our first phase, an oracle is built by reversing regexes to indicate the positions at which each lookaround holds, but unlike our algorithms there is no support for capture groups (neither inside lookarounds nor in

[9]https://bugs.chromium.org/p/v8/issues/detail?id=14099

the main regex). Their optimization to reduce memory usage for lookbehinds is a special case of the one we present in Section 4.4 when the main expression does not contain capture groups. The corresponding optimization for lookaheads is not directly applicable when the main regex contains capture groups (reversing the regex does not preserve capture group priority, so we do not apply any such transformation).

Mitigating ReDoS ReDoS is a serious security concern for many applications [Davis et al. 2018; Staicu and Pradel 2018]. There exist two main kinds of mitigations. The first one consists in *detecting* regexes for which exponential backtracking can happen [Kirrage et al. 2013; Liu et al. 2021; Parolini and Miné 2022; Shen et al. 2018; Weideman et al. 2016; Wüstholz et al. 2017], and *repairing* them when possible (generating equivalent regexes without backtracking explosion) [Chida and Terauchi 2022, 2023; van der Merwe et al. 2017]. The other consists in only using linear engines and avoid catastrophic backtracking altogether; this has become the default practice in Rust, Go and for anyone using the RE2 library. By adding lookarounds to linear engines, our work extends the applicability of this second ReDoS mitigation. Even simple regexes where backtracking is exponential because of other constructs (like `/⦇a*⦈*(?=b)/` on a string of `"a"`s) can now be supported by linear engines.

Other Regex Features Recent work [Glaunec et al. 2023; Holík et al. 2023; Turonová et al. 2020] has explored matching counted repetition with a time complexity independent of the counters. However, these solutions do not preserve the priority between threads that is needed to support capture groups. Other work [Schmid 2019] has defined subsets of regexes with backreferences that can be matched in polynomial time. There exist multiple semantics for backreferences, and [Berglund and van der Merwe 2017] explored their relative expressive powers. [Borsotti and Trofimovich 2021] has presented algorithms to handle capture groups in longest-match POSIX semantics.

7 CONCLUSION

We have presented novel algorithms for matching most JavaScript regex linearly in the sizes of both the regex and the input string. These guarantees are crucial for many applications working with user-provided regexes or strings: without them, applications are susceptible to regex-based denial-of-service attacks. In the process, we have highlighted several incorrect assumptions in state-of-the-art linear engines, affecting both complexity and correctness. We have presented the first algorithm to match unrestricted lookarounds in a linear-time engine, and thereby reduced the expressivity gap between backtracking and linear approaches. Parts of our work, including our captureless lookbehind algorithm, are applicable to other linear regex engines, and could be included in Rust or RE2. We have implemented and experimentally validated the practicality of all our algorithms, and merged some of them in the V8 JavaScript engine, putting them in the hands of millions of developers and making it more practical to chose secure-by-default execution.

Our work sheds light on the importance of seemingly minor semantic design choices. For instance, JavaScript's semantics for nullable quantifiers requires more complex algorithms than other regex languages. On the contrary, its capture-reset property enables us to support, for the first time in any regex language, linear-time matching of capturing lookarounds.

Because of backreferences, JavaScript regexes cannot be fully supported by linear engines. The current trend in modern languages is to move away from backreferences and provide secure regex matching, either by default (Rust and Go) or as an alternative (.NET [Moseley et al. 2023]). Our work shows how to bring these benefits to JavaScript, a language particularly affected by ReDoS [Staicu and Pradel 2018]. We show that this requires sacrificing very few features (backreferences, counted repetition, lazy nullable plus). As a future direction, we note that the surprising amount of non-linearities and semantic subtleties that we uncovered suggests that JavaScript regexes would strongly benefit from a systematic formalization of the regex language and its engines.

ACKNOWLEDGMENTS

We thank Ludovic Mermod for working with us on implementing our algorithms in the V8 engine. We thank V8 developers Patrick Thier, Jakob Linke and Olivier Flückiger for helpful discussions and code reviews. We also thank Nate Foster and Viktor Kuncak for the feedback on this paper.

ARTIFACT AVAILABILITY

A virtual machine image with our implementations is available as an artifact [Barrière and Pit-Claudel 2024]. It contains our OCaml implementations, our V8 patches, scripts to reproduce our benchmarks and analyses, and a script to rebuild and provision the virtual machine image itself. The OCaml engine implementing our algorithms is also available online: https://github.com/epfl-systemf/RegElk.

REFERENCES

Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 255–300.

Aurèle Barrière and Clément Pit-Claudel. 2024. Artifact for "Linear Matching of JavaScript Regular Expressions" at PLDI 2024. https://doi.org/10.5281/ZENODO.10806044

Martin Berglund and Brink van der Merwe. 2017. Regular Expressions with Backreferences Re-examined. In *Proceedings of the Prague Stringology Conference 2017, Prague, Czech Republic, August 28-30, 2017*. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 30–41. http://www.stringology.org/event/2017/p04.html

Martin Berglund, Brink van der Merwe, and Steyn van Litsenborgh. 2021. Regular Expressions with Lookahead. *J. Univers. Comput. Sci.* 27, 4 (2021), 324–340. https://doi.org/10.3897/jucs.66330

Angelo Borsotti and Ulya Trofimovich. 2021. Efficient POSIX submatch extraction on nondeterministic finite automata. *Softw. Pract. Exp.* 51, 2 (2021), 159–192. https://doi.org/10.1002/SPE.2881

Carl Chapman and Kathryn T. Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 282–293. https://doi.org/10.1145/2931037.2931073

Nariyoshi Chida and Tachio Terauchi. 2022. Repairing DoS Vulnerability of Real-World Regexes. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2060–2077. https://doi.org/10.1109/SP46214.2022.9833597

Nariyoshi Chida and Tachio Terauchi. 2023. Repairing Regular Expressions for Extraction. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1633–1656. https://doi.org/10.1145/3591287

Chromium. 2009. Irregexp, Google Chrome's New Regexp Implementation. https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html.

Cloudflare. 2019. Details of the Cloudflare outage. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*. ACM, 37–46. https://doi.org/10.1145/1292535.1292541

Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast. https://swtch.com/~rsc/regexp/regexp1.html.

Russ Cox. 2009. Regular Expression Matching: the Virtual Machine Approach. https://swtch.com/~rsc/regexp/regexp2.html.

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*. ACM, 246–256. https://doi.org/10.1145/3236024.3236027

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 443–454. https://doi.org/10.1145/3338906.3338909

James C. Davis, Francisco Servant, and Dongyoon Lee. 2021. Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS). In *42nd IEEE Symposium on Security and Privacy, SP 2021*. IEEE, 1–17. https://doi.org/10.1109/SP40001.2021.00032

Mark Jason Dominus. 2000. Perl Regular Expression Matching is NP-Hard. https://perl.plover.com/NPC/NPC-3SAT.html.

DukTape. 2013. DukTape Regular Expressions. https://github.com/svaarala/duktape/blob/master/doc/regexp.rst.

ECMA-262. 2024. RegExp (Regular Expression) Objects. https://262.ecma-international.org/13.0/#sec-regexp-regular-expression-objects.

Bernd Finkbeiner and Henny Sipma. 2004. Checking Finite Traces Using Alternating Automata. *Formal Methods Syst. Des.* 24, 2 (2004), 101–127. https://doi.org/10.1023/B:FORM.0000017718.28096.48

Andrew Gallant. 2014. Crate regex: An implementation of regular expressions for Rust. https://docs.rs/regex/latest/regex/.

Andrew Gallant. 2023. Regex engine internals as a library. https://blog.burntsushi.net/regex-internals/.

Wouter Gelade and Frank Neven. 2012. Succinctness of the Complement and Intersection of Regular Expressions. *ACM Trans. Comput. Log.* 13, 1 (2012), 4:1–4:19. https://doi.org/10.1145/2071368.2071372

Alexis Le Glaunec, Lingkun Kong, and Konstantinos Mamouras. 2023. Regular Expression Matching using Bit Vector Automata. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 492–521. https://doi.org/10.1145/3586044

V. M. Gluškov. 1961. Abstract theory of automata. *Uspehi Mat. Nauk* 16, 5(101) (1961), 3–62.

Google. 2022. RE2: A fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in PCRE, Perl, and Python. https://github.com/google/re2.

Google. 2023. RE2 Wiki. https://github.com/google/re2/wiki/Glossary.

Hermes. 2022. Hermes Regex Engine. https://hermesengine.dev/docs/regexp/.

Lukás Holík, Juraj Síc, Lenka Turonová, and Tomás Vojnar. 2023. Fast Matching of Regular Patterns with Synchronizing Counting. In *Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023 (Lecture Notes in Computer Science, Vol. 13992)*. Springer, 392–412. https://doi.org/10.1007/978-3-031-30829-1_19

Iain Ireland. 2020. A New RegExp Engine in SpiderMonkey. https://hacks.mozilla.org/2020/06/a-new-regexp-engine-in-spidermonkey/.

James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013 (Lecture Notes in Computer Science, Vol. 7873)*. Springer, 135–148. https://doi.org/10.1007/978-3-642-38631-2_11

Ville Laurikari. 2000. NFAs with Tagged Transitions, Their Conversion to Deterministic Automata and Application to Regular Expressions. In *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000*. IEEE Computer Society, 181–187. https://doi.org/10.1109/SPIRE.2000.878194

Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1468–1484. https://doi.org/10.1109/SP40001.2021.00062

Konstantinos Mamouras and Agnishom Chattopadhyay. 2024. Efficient Matching of Regular Expressions with Lookaround Assertions. *Proc. ACM Program. Lang.* 8, POPL (2024), 2761–2791. https://doi.org/10.1145/3632934

Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1026–1049. https://doi.org/10.1145/3591262

MuJS. 2014. MuJS Regex Engine. https://github.com/ccxvii/mujs/blob/master/regexp.c.

Francesco Parolini and Antoine Miné. 2022. Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks. In *Theoretical Aspects of Software Engineering - 16th International Symposium, TASE 2022 (Lecture Notes in Computer Science, Vol. 13299)*. Springer, 73–91. https://doi.org/10.1007/978-3-031-10363-6_6

Rob Pike. 1987. The text editor sam. http://doc.cat-v.org/plan_9/4th_edition/papers/sam/.

QuickJS. 2020. QuickJS Regex Engine. https://github.com/bellard/quickjs/blob/master/libregexp.c.

RE2. 2017. GitHub Issue: Please Support Negative Lookahead. https://github.com/google/re2/issues/156.

Markus L. Schmid. 2019. Regular Expressions with Backreferences: Polynomial-Time Matching Techniques. (2019). arXiv:1903.05896

Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*. ACM, 225–235. https://doi.org/10.1145/3238147.3238159

Stack Exchange. 2016. Outage Postmortem. https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016.

Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium, USENIX Security 2018*. USENIX Association, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

TC39. 2010. test262. https://github.com/tc39/test262

Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* (1968). https://doi.org/10.1145/363347.363387

TIOBE. 2023. Programming Community Index for April 2023. https://www.tiobe.com/tiobe-index/.

Stephen Toub. 2022. Regular Expression Improvements in .NET 7. https://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7/.

Lenka Turonová, Lukás Holík, Ondrej Lengál, Olli Saarikivi, Margus Veanes, and Tomás Vojnar. 2020. Regex matching with counting-set automata. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 218:1–218:30. https://doi.org/10.1145/3428286

V8. 2021. An Additional Non-backtracking RegExp Engine. https://v8.dev/blog/non-backtracking-regexp.

Brink van der Merwe, Nicolaas Weideman, and Martin Berglund. 2017. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017*. ACM, 38:1–38:10. https://doi.org/10.1145/3129416.3129440

WebKit. 2018. JavaScriptCore RegExp Processing. https://trac.webkit.org/wiki/JSCRegExpProcessingAndJSCGoals.

Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. 2016. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016 (Lecture Notes in Computer Science, Vol. 9705)*. Springer, 322–334. https://doi.org/10.1007/978-3-319-40946-7_27

Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017 (Lecture Notes in Computer Science, Vol. 10206)*. 3–20. https://doi.org/10.1007/978-3-662-54580-5_1

A EXAMPLE: EXECUTING OUR LOOKAROUNDS ALGORITHM

Consider the regex $r =$ `/(c)⦇a(?=a*(?<=c(a*))b)⦈*/` on string `"caab"`. This regex gets annotated as follows: `/(c)`$_{\#1}$`⦇a(?=`$_{\leqslant 1}$`a*(?<=`$_{\leqslant 2}$`c(a*)`$_{\#2}$`)b)⦈*/`.

A.1 First step - Building the oracle

```
0: Fork 3 1
1: ConsumeAny
2: Jump 0
3: Consume c
4: Fork 5 7
5: Consume a
6: Jump 4
7: WriteOracle 2
```

Fig. 28. Oracle Bytecode for $r_{\leqslant 2}$

In our first step, we build the oracle. The oracle has a two rows (there are two lookarounds) and 5 columns (the string has 4 characters). For the row of lookaround ≼2, we consider $r_{\leqslant 2}$ = `/c(a*)/`. In that step, we remove the capture group and `SetQuant` instructions, use a `WriteOracle` instruction and add a `.*?` prefix. The resulting bytecode is shown on Figure 28. The first three instructions are for the lazy prefix. Executing an NFA simulation, in the forward direction, on string `"caab"` finds matches at positions 1, 2 and 3 (every position between the `"c"` and the `"b"`).

```
0: Fork 3 1
1: ConsumeAny
2: Jump 0
3: Consume b
4: CheckOracle 2
5: Fork 6 8
6: Consume a
7: Jump 5
8: WriteOracle 1
```

Fig. 29. Oracle Bytecode for $r_{\leqslant 1}$

For the first row of the oracle (lookaround ≼1), we consider $r_{\leqslant 1}$ = `/a*(?<=`$_{\leqslant 2}$`)b/`. Since this is a lookahead, we compute $rev(r_{\leqslant 1})$ = `/b(?<=`$_{\leqslant 2}$`)a*/` and add a `.*?` prefix. The resulting bytecode is shown on Figure 29. The inner lookaround ≼2 is simply compiled to a single `CheckOracle` instruction which reads the previous row of the oracle we just computed before. Executing an NFA simulation, in the backward direction, on string `"caab"` also finds matches at positions 1, 2 and 3.

The final oracle is shown on the following table:

String Position	0	1	2	3	4
$r_{\leqslant 1}$	X	✓	✓	✓	X
$r_{\leqslant 2}$	X	✓	✓	✓	X

A.2 Second step - Matching the main expression and groups `#0` and #1

```
0: Fork 3 1
1: ConsumeAny
2: Jump 0
3: SetReg #0:entry
4: SetReg #1:entry
5: Consume c
6: SetReg #1:exit
7: Fork 8 12
8: SetQuant 1
9: Consume a
10: CheckOracle 1
11: Jump 7
12: SetReg #0:exit
13: Accept
```

Fig. 30. Bytecode of r

In the second step, we compile and match the main expression. Note that the entire lookahead will be compiled to a single bytecode instruction `CheckOracle 1`. We compile the regex `/.*?((c)`$_{\#1}$`⦇a(?=`$_{\leqslant 1}$`a*(?<=`$_{\leqslant 2}$`c(a*)`$_{\#2}$`)b)⦈*)`$_{\#0}$`/` and show its bytecode on Figure 30.

Executing an NFA simulation forward on this expression will find a best match where group`#0` is `"caa"`, group#1 is `"c"` and group#2 is undefined. The values for groups `#0` and #1 are correct because these groups are defined inside the main expression. However, group#2 is defined inside a lookahead.

A.3 Final step - Getting the value of group#2

```
0: Fork 1 4
1: SetQuant 2
2: Consume a
3: Jump 0
4: CheckOracle 2
5: Consume b
6: Accept
```

Fig. 31. Reconstruction Bytecode of $r_{\leqslant 1}$

We now reconstruct the value of group#2. We can see that in the match of the second step, the winning thread last used the oracle for lookahead ⩽1 at position 3. We compile the lookahead $r_{\leqslant 1}$ to bytecode. This time, we do not reverse it and keep capture groups. Its bytecode is shown on Figure 31. We start the NFA simulation in a forward direction at position 3. A match is found, where the lookbehind ⩽2 was last used at position 3, meaning that we need to run it too in order to get the value of the group#2.

```
0: Fork 1 6
1: SetQuant 3
2: SetReg #2:entry
3: Consume a
4: SetReg #2:exit
5: Jump 0
6: Consume c
7: Accept
```

Fig. 32. Reconstruction Bytecode of $r_{\leqslant 2}$

We reverse and compile $r_{\leqslant 2}$ to bytecode, without removing capture groups. Its bytecode is shown on Figure 32. We start the NFA simuation in a backward direction, starting at position 3. A match is found where capture group#2 has value `"a"`. We now have the full results of matching r on string `"caab"`: group#`0` has value `"caa"`, group#1 has value `"c"` and group#2 has value `"a"`.