ECMAScript Regexes: Linear Matching, Formal Verification

Consider the Impact of Regex Proposals to Linear Implementations

Aurèle Barrière Clément Pit-Claudel Michael Ficarra Mikhail Barash

Our specialty: formal methods for programming languages

Semantics, compilation and execution techniques, formal verification (Rocq/Coq proof assistant).
In the last 3 years: formal methods for modern regexes.

In this talk

Linear-Time Regex Matching

- Linear-time matching solves concrete issues.
- ECMAScript is surprisingly adapted for linear-time matching.
  Only language for which we can support **all** lookarounds in linear-time.
- This property should not be relinquished without great consideration of the consequences.

Formal Verification

- Formal proofs can bring a lot to modern regexes.
- ECMAScript regexes have an elegant semantics, suitable for proofs.
- We are verifying properties and developing verified tools/engines.

Most engines have exponential complexity

`"a".repeat(15).match(/(a*)*b/)`: 1 ms
`"a".repeat(35).match(/(a*)*b/)`: 12 minutes
`"a".repeat(100).match(/(a*)*b/)` would take 10^{14} years!

Regex performance matters

ReDoS Vulnerability: Regular Expression Denial Of Service

- 12% of JavaScript-based web servers were found vulnerable [USENIX'18].
- Major services (Cloudflare, StackOverflow) brought down due to regex performance issues.
- It's easy to run into these issues: `/.*a.*b/` has quadratic complexity.
- Also an issue when using regexes as a target language.
  E.g. Glob patterns to regexes: [CVE-2026-27903, CVE-2026-27904, CVE-2026-26996, CVE-2026-33671]
- More 2026 ReDoS: [CVE-2026-41040, CVE-2026-40319, CVE-2026-39320, CVE-2026-35611, CVE-2026-35458, CVE-2026-35213, CVE-2026-35041, CVE-2026-34939, CVE-2026-33169, CVE-2026-33079, CVE-2026-30925, CVE-2026-30837, CVE-2026-29856, CVE-2026-28356, CVE-2026-26936, CVE-2026-26006, CVE-2026-24001, CVE-2026-23956, CVE-2026-22809, CVE-2026-22178, CVE-2026-21868, CVE-2025-9670, CVE-2026-8159, CVE-2026-5986, CVE-2026-4926, CVE-2026-4923, CVE-2026-4867, CVE-2026-4539, CVE-2026-3293, CVE-2026-2327, CVE-2026-1388, CVE-2026-0967, CVE-2026-0668, CVE-2026-0621]. (1 CVE every 2.7 days)

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"`.

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`	
Disjunction	`r1 | r2`				
Quantifiers	`r*`	`r+`	`r*?`	`r{n,m}`	⟶ Matches `r` from `n` to `m` times.
Capture Groups	`(r)`				
Anchors	`^`	`$`			
Lookarounds	`(?=r)`	`(?!r)`	`(?<=r)`	`(?<!r)`	
Backreferences	`\1`	`\k<name>`			

Returns the substring last matched by subexpressions in parentheses:
`"abcd".match(/a(b*)|abc/) = ["ab", "b"]`

Makes matching NP-Hard!

Adds a condition in a regex:
`/a(?=b)/` matches `"a"` only if followed by `"b"`.

Matches the string backwards:
`"TC39".match(/(?<=TC)\d+/) = "39"`.

Some languages (Rust, Go) omit features (backreferences, lookarounds) and guarantee linear-time.
Others (.NET) provide both an exponential-time engine for all features, and a linear one for a subset.

The V8 Experimental Engine

Behind V8 flag `--enable-experimental-regexp-engine` and regex flag `/l`.
`"a".repeat(100).match(/(a*)*b/l)`: 0.01 ms.
Some unsupported features: u and v flags, duplicate named groups...

Recent proposal (Feb 2026 – Stage 0)

https://github.com/monolithed/proposal-regular-expression-linear-time-flag

Libraries for linear matching

- `node-re2`: Node.js bindings for the RE2 linear engine.
- `regolith`: TypeScript/JavaScript server-side lib using the Rust linear engine.
- `re2js`: RE2 ported to JavaScript.
  Uses our linear-time algorithm for captureless lookbehinds.
- `@iter-tools/regex` and `@bablr/regex-vm`: streaming regex implementation.

Missing features. Common features may have different semantics!

Can we match ECMAScript regexes in Linear Time?

Yes! Even more so than in other languages.

22.2.2.3.1 RepeatMatcher (*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*)

The abstract operation RepeatMatcher takes arguments *m* (a Matcher), *min* (a non-negative integer), *max* (a non-negative integer or +∞), *greedy* (a Boolean), *x* (a MatchState), *c* (a MatcherContinuation), *parenIndex* (a non-negative integer), and *parenCount* (a non-negative integer) and returns either a MatchState or FAILURE. It performs the following steps when called:

1. If *max* = 0, return *c*(*x*).
2. Let *d* be a new MatcherContinuation with parameters (*y*) that captures *m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, and *parenCount* and performs the following steps when called:
   a. Assert: *y* is a MatchState.
   b. If *min* = 0 and *y*.[[EndIndex]] = *x*.[[EndIndex]], return FAILURE.
   c. If *min* = 0, let *min2* be 0; otherwise let *min2* be *min* - 1.
   d. If *max* = +∞, let *max2* be +∞; otherwise let *max2* be *max* - 1.
   e. Return RepeatMatcher(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*).
3. Let *cap* be a copy of *x*.[[Captures]].
4. For each integer *k* in the inclusive interval from *parenIndex* + 1 to *parenIndex* + *parenCount*, set *cap*[*k*] to **undefined**.
5. Let *Input* be *x*.[[Input]].
6. Let *e* be *x*.[[EndIndex]].
7. Let *xr* be the MatchState { [[Input]]: *Input*, [[EndIndex]]: *e*, [[Captures]]: *cap* }.
8. If *min* ≠ 0, return *m*(*xr*, *d*).
9. If *greedy* is **false**, then
   a. Let *z* be *c*(*x*).
   b. If *z* is not FAILURE, return *z*.
   c. Return *m*(*xr*, *d*).
10. Let *z* be *m*(*xr*, *d*).
11. If *z* is not FAILURE, return *z*.
12. Return *c*(*x*).

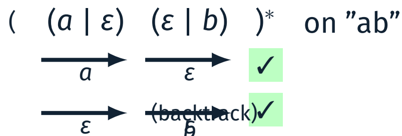
Invalid Iterations

Optional (min=0) iterations that do not advance in the string are invalid.

Capture Reset

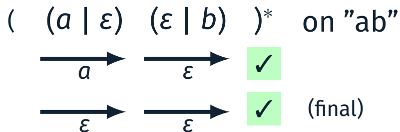
At each iteration, reset the values of capture groups inside the quantifier.

ECMAScript: Empty iterations are invalid.



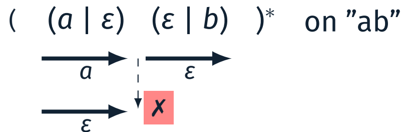
Result: 2 iterations, match "ab".

PCRE/Python/Java/.NET: Empty iterations are final.



Result: 2 iterations, match "a".

Rust/Go/RE2: ε-loops are forbidden.



Result: 1 iteration, match "a".

$|r|$: regex size
$|s|$: string size

We needed new algorithms for the star!

Standard linear-time algorithms support the Rust/Go/RE2 star with complexity $O(|r| \times |s|)$.
We designed a variant of these algorithms for the ECMAScript star, with complexity $O(|r| \times |s|)$.
We fixed V8 Experimental.
The best we found for the PCRE/Python/Java/.NET star has $O\left(|r|^2 \times |s|\right)$ complexity.

Regex-size complexity

One case where ECMAScript semantics is worse: the lazy nullable plus.
Can we deal with the lazy nullable plus in regex-size linear time?
(appears in 0.003% of regexes)

Capture Reset

At each quantifier iteration, reset the value of groups inside.

```
"ab".match(/(?:(a)|b)*/) = ["ab", undefined]
```

Consequence: Only the last iteration of a quantifier can define groups.
Consequence: Each capture group inside a lookaround can only be defined by the last time the lookaround was used.

Our linear algorithm

- 1. Match without capturing inside lookarounds.
- 2. Reconstruct groups in lookarounds:
  Match each lookaround **once** from where they were last used.

This algorithm has now been implemented in V8 Experimental.
Supports **all** lookarounds, even unbounded ones.

Current state of linearity

- Without backreferences, everything can be matched in string-size linear time.
- Without counted quantifiers and the lazy nullable plus, everything can be matched in regex-size linear time.

Current proposals that should not affect linearity

- Stage 3: Legacy Features
- Stage 2: Buffer Boundaries
- Stage 1: Extended mode
- Stage 1: \R escape

Can we handle this in linear time?

- Stage 1: Atomic operators

Recent work [ESOP'24] presents a linear-time algorithm for atomic groups, but it does not work if the inner regex is nullable!

Would you agree that:

It would be valuable to provide users with a way to ensure their regex is executed in linear time?

Formal Verification for ECMAScript Regexes

Modern regexes are difficult

- **Designing a correct algorithm is hard.**
  We found bugs in engines.
  Initial peer review: *"There may be problems in the techniques presented in Section 4.1, 4.4 and 4.5."*
- **The semantics is counter-intuitive.**
  We found errors in papers.
  `r?` is not equivalent to `r|` and `r??` is not equivalent to `|r`.
  Counter-example: `"abac".match(/(?:(?=(a))??)ab\1c/)`
- **Optimizing regexes is hard**
  We found bugs in optimizers.
  You can't optimize `r{min1,max1}r{min2,max2}` into `r{min1+min2,max1+max2}`.
  Counter-example: `"aba".match(/(?:a|ab){0,1}(?:a|ab){1,1}/)`

Let's **prove** algorithms, properties and optimizations!
We needed to define ECMAScript regex semantics **in a proof assistant**.

22.2.2.4.1 IsWordChar (rer, Input, e)

The abstract operation IsWordChar takes arguments rer (a RegExp Record), Input (a List of characters), and e (an integer) and returns a Boolean.
It performs the following steps when called:

1. Let InputLength be the number of elements in Input.
2. If e = -1 or e = InputLength, return false.
3. Let c be the character Input[e].
4. If WordCharacters(rer) contains c, return true.
5. Return false.

```
(** >>
   22.2.2.4.1 IsWordChar ( rer, Input, e )
```

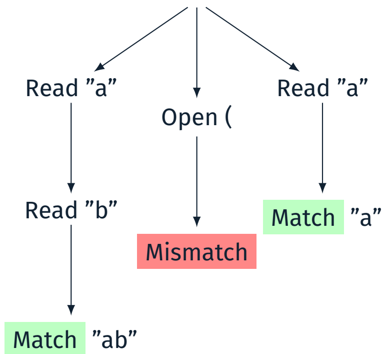
An executable mechanization

We ran Test262 with this code.
498 relevant regex tests. 495 passed, 3 timeouts.
Limitations: 14th edition, missing some unicode functions and parsing.

Reasoning about ECMAScript regexes

The ECMAScript style is a great intuitive specification for backtracking semantics.
But what would be the ideal semantics for formal verification?
The best way to reason about backtracking semantics is not with a backtracking algorithm.

Backtracking Tree for $ab|(c)|a$ on string "ab"



Play around with the tree semantics:
https://lin.den.re/viz/

More behaviors

- Represents all paths explored by linear algorithms.
- Useful to reason about subregex equivalence.

Faithfulness

We proved that the leftmost Match node of the tree is equal to the result of the previous mechanization.

A more succinct inductive rephrasing of the semantics

$$\frac{}{([\,], i, gm, d) \Downarrow \mathsf{Match}}\ \textsc{Match}$$

$$\frac{(l, i, \mathsf{GM}_{\mathsf{close}}(gm, g, \mathsf{idx}(i)), d) \Downarrow t}{(\mathsf{Aclose}\ g :: l, i, gm, d) \Downarrow \mathsf{Close}\ g\ t}\ \textsc{Close}$$

$$\frac{\mathsf{advance_backref}(gm, g, i, d) = \mathsf{Some}\ (s, i') \qquad (l, i', gm, d) \Downarrow t}{(\backslash g :: l, i, gm, d) \Downarrow \mathsf{BackrefPass}\ s\ t}\ \textsc{Backref}$$

$$\frac{\mathsf{advance_backref}(gm, g, i, d) = \mathsf{None}}{(\backslash g :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{BackrefFail}$$

$$\frac{i_{check} <_d i \qquad (l, i, gm, d) \Downarrow t}{(\mathsf{Acheck}\ i_{check} :: l, i, gm, d) \Downarrow \mathsf{Progress}\ t}\ \textsc{Check}$$

$$\frac{\neg(i_{check} <_d i)}{(\mathsf{Acheck}\ i_{check} :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{CheckFail}$$

$$\frac{(r :: r\{min, \Delta, p\} :: l, i, \mathsf{GM}_{\mathsf{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t}{(r\{min + 1, \Delta, p\} :: l, i, gm, d) \Downarrow \mathsf{Reset}\ \mathcal{G}(r)\ t}\ \textsc{Forced}$$

$$\frac{(l, i, gm, d) \Downarrow t}{(r\{0, 0, p\} :: l, i, gm, d) \Downarrow t}\ \textsc{Done}$$

$$\frac{\mathsf{advance}(cd, i, d) = \mathsf{Some}\ (c, i') \qquad (l, i', gm, d) \Downarrow t}{(cd :: l, i, gm, d) \Downarrow \mathsf{Read}\ c\ t}\ \textsc{Read}$$

$$\frac{\mathsf{advance}(cd, i, d) = \mathsf{None}}{(cd :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{ReadFail}$$

$$\frac{(l, i, gm, d) \Downarrow t_{skip} \qquad (r :: \mathsf{Acheck}\ i :: r\{0, \Delta, \top\} :: l, i, \mathsf{GM}_{\mathsf{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t_{iter}}{(r\{0, \Delta + 1, \top\} :: l, i, gm, d) \Downarrow \mathsf{Choice}\ (\mathsf{Reset}\ \mathcal{G}(r)\ t_{iter})\ t_{skip}}\ \textsc{Greedy}$$

$$\frac{(r_1 :: l, i, gm, d) \Downarrow t_1 \qquad (r_2 :: l, i, gm, d) \Downarrow t_2}{((r_1 | r_2) :: l, i, gm, d) \Downarrow \mathsf{Choice}\ t_1\ t_2}\ \textsc{Disj}$$

$$\frac{(l, i, gm, d) \Downarrow t_{skip} \qquad (r :: \mathsf{Acheck}\ i :: r\{0, \Delta, \bot\} :: l, i, \mathsf{GM}_{\mathsf{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t_{iter}}{(r\{0, \Delta + 1, \bot\} :: l, i, gm, d) \Downarrow \mathsf{Choice}\ t_{skip}\ (\mathsf{Reset}\ \mathcal{G}(r)\ t_{iter})}\ \textsc{Lazy}$$

$$\frac{(r_1 :: r_2 :: l, i, gm, \rightarrow) \Downarrow t}{(r_1\ r_2 :: l, i, gm, \rightarrow) \Downarrow t}\ \textsc{SeqForward}$$

$$\frac{(r_2 :: r_1 :: l, i, gm, \leftarrow) \Downarrow t}{(r_1\ r_2 :: l, i, gm, \leftarrow) \Downarrow t}\ \textsc{SeqBackward}$$

$$\frac{\mathsf{dir}(lk) = d' \qquad \mathsf{lk_result}(lk, t_{look}, gm, \mathsf{idx}(i)) = \mathsf{Some}\ gm' \qquad ([r], i, gm, d') \Downarrow t_{look} \qquad (l, i, gm', d) \Downarrow t}{((?lk\ r) :: l, i, gm, d) \Downarrow \mathsf{LK}\ lk\ t_{look}\ t}\ \textsc{Lookaround}$$

$$\frac{(l, i, gm, d) \Downarrow t}{(\varepsilon :: l, i, gm, d) \Downarrow t}\ \textsc{Epsilon}$$

$$\frac{(r :: \mathsf{Aclose}\ g :: l, i, \mathsf{GM}_{\mathsf{open}}(gm, g, \mathsf{idx}(i)), d) \Downarrow t}{((_g\ r) :: l, i, gm, d) \Downarrow \mathsf{Open}\ g\ t}\ \textsc{Group}$$

$$\frac{\mathsf{dir}(lk) = d' \qquad \mathsf{lk_result}(lk, t_{look}, gm, \mathsf{idx}(i)) = \mathsf{None} \qquad ([r], i, gm, d') \Downarrow t_{look}}{((?lk\ r) :: l, i, gm, d) \Downarrow \mathsf{LKMismatch}\ lk\ t_{look}}\ \textsc{LookaroundFail}$$

$$\frac{\mathsf{check_anchor}(a, i) = \top \qquad (l, i, gm, d) \Downarrow t}{(a :: l, i, gm, d) \Downarrow \mathsf{AnchorPass}\ a\ t}\ \textsc{Anchor}$$

$$\frac{\mathsf{check_anchor}(a, i) = \bot}{(a :: l, i, gm, d) \Downarrow \mathsf{Mismatch}}\ \textsc{AnchorFail}$$

Only 21 inductive rules to support all the control-flow! And we can reason by induction.
Open question: Could we write the spec in such a way that we can derive both styles?

Properties of the spec

- Matching never fails (all assertions hold, arrays are accessed in range...).
- Matching always terminates.
- Complexity class: ECMAScript regex matching is PSPACE-complete.
  Without lookarounds, finding the top-priority match is OPTP-complete.

Regex Equivalence and Translations

- In some cases, `r{min1,max1}r{min2,max2}` is equivalent to `r{min1+min2,max1+max2}` !
  For instance, when inside a lookbehind and `min2=max2`.
- We are proving the translation from the Interoperable Regex RFC 9485.
  https://datatracker.ietf.org/doc/rfc9485/ We found some small mistakes.

```
Theorem no_failure:
  (* For all valid regex r and string s *)
  ∀ r m s, compileSubPattern r = Success m →
  earlyErrors r = OK →
  (* the matcher cannot fail. *)
  m (init_state s) ≠ Failure.
```

Algorithms

- We proved the PikeVM linear algorithm, used in V8 Experimental engine!
- We proved the prefix acceleration optimization, and improved it.

A formally verified, linear-time, efficient engine

- Generate verified C code that executes our algorithms.
- Linearity is not speed: verify optimizations.

Other uses

- **Encode and compare many languages:** we're extending our semantics for other regex flavors.
- **Fearless refactoring:** prove a new version of the spec is equivalent to the old one.
- **Feature encodings:** we proved that anchors can be expressed with lookarounds. Should atomic groups be next?

What other properties would you like to see verified?

Conclusion

For most JavaScript regexes, you could use a formally verified, linear-time, efficient engine!
We want to implement and prove that engine.

Linear algorithms · PLDI 24 Paper

Linear Matching of JavaScript Regular Expressions.
`https://github.com/LindenRegex/RegElk`

Mechanized Semantics · ICFP 24 Paper

A Coq Mechanization of JavaScript
Regular Expression Semantics.
`https://github.com/LindenRegex/Warblre`

Formal Verification · POPL 26 Paper

Formal Verification for JavaScript Regular Expressions:
a Proven Semantics and its Applications.
`https://github.com/LindenRegex/Linden`

Our project:
`https://lin.den.re/`

Our work – Summary

`https://lin.den.re/`

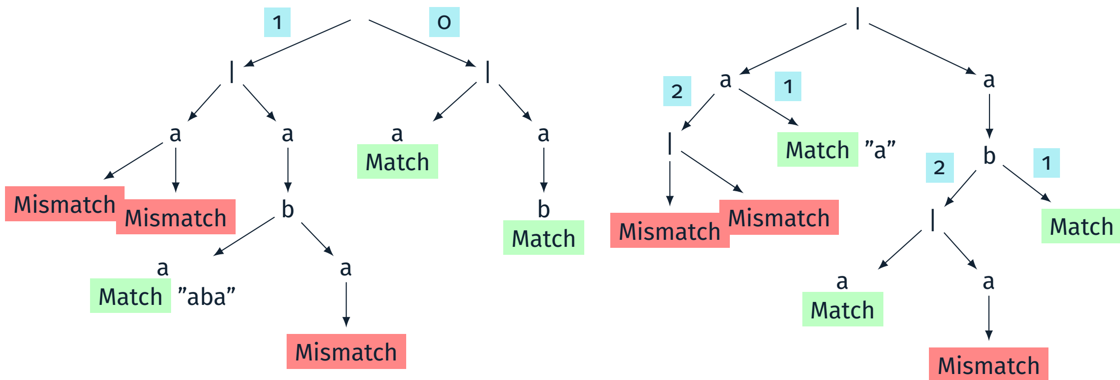
- ECMAScript is in a unique position for linear-time matching.
- We can support lookarounds in linear time.
- We have mechanized semantics for proofs:
  properties, equivalences, algorithms…
- We are developing a formally verified, linear-time engine.

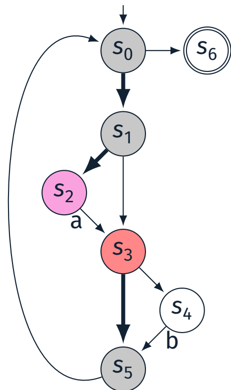
Discussion: how to consider linearity?

- Can complexity be considered for new proposals?
- How should linearity be exposed to users?
  The spec already has complexity constraints (Maps, Sets, WeakMaps, WeakSets).
  Should it be user-controlled (`l` flag) or heuristics-based?
  Suggestion by Ron Buckton: a getter that exposes if the engine will perform a linear match.

Appendix

Rewriting `r{min1,max1}r{min2,max2}` into `r{min1+min2,max1+max2}` can be wrong!
Counter-example: `(?:a|ab){0,1}(?:a|ab){1,1}` and `(?:a|ab){1,2}` on string "aba".



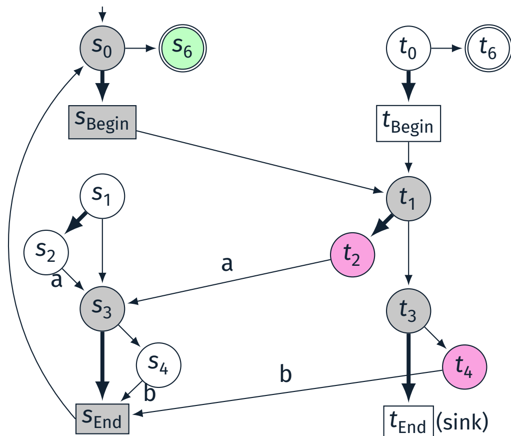


NFA for $((a \mid \varepsilon)\ (\varepsilon \mid b))^*$
with priority edges (**bold**).

PikeVM algorithm on string "ab"

- Build NFA.
- Find the top priority path.
- To ensure linearity: never visit the same *configuration* twice.
  (configuration = NFA state & string position)

ε-loops visit the same configuration twice.
The PikeVM cannot find the top-priority path of the JavaScript semantics!



You can exit the star.

You cannot exit the star.

Our new NFA construction

- Copy the NFA on the side.
- New Begin and End states.
- Begin moves to the right.
- Reading moves to the left.

This restores correctness for the JavaScript star!

Key Insight 1 - Pre-computation

In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Key Insight 2 - JavaScript unique semantics

Only the last iteration of a quantifier can define groups.

```
"ab".match(/(?:(a)|b)*/) = ["ab", undefined]
```

Each capture group inside a lookaround can only be defined by the last time the lookaround was used.

Our three-steps linear algorithm

- Pre-compute a table.
- Match the main expression.

What if lookarounds have capture groups?

- Reconstruct groups in lookarounds:
  Match each lookaround once.
  from where they were last used.

Example: (*a* (?=(a | b)))* on string "aaac".

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗