Systems and Formalisms Lab

TC39-TG5 meeting · August 27th, 2025

Exploring JavaScript Regular Expressions:

matching algorithms, mechanized semantics, formal verification

Aurèle Barrière Clément Pit-Claudel

SYSTEMF · EPFL

Supported by:

Swiss National Science Foundation (#10003649) and Open Research Data Program (ETH Domain)

Our specialty: formal methods for programming languages

Programming language semantics, compilation and execution techniques, formal verification.
We use proof assistants (Rocq/Coq) to reason about languages.

Our current project: formal methods for JavaScript regexes.

Our specialty: formal methods for programming languages

Programming language semantics, compilation and execution techniques, formal verification.
We use proof assistants (Rocq/Coq) to reason about languages.

Our current project: formal methods for JavaScript regexes.

Formal methods and JavaScript

- The ECMAScript specification is an asset for formal reasoning!
- Thanks to the specification, formal methods can be applied to JavaScript!
  Previous mechanization work: ESMeta, JSCert, KJS, λ_{JS} ...

Our specialty: formal methods for programming languages

Programming language semantics, compilation and execution techniques, formal verification.
We use proof assistants (Rocq/Coq) to reason about languages.

Our current project: formal methods for JavaScript regexes.

Formal methods and JavaScript

- The ECMAScript specification is an asset for formal reasoning!
- Thanks to the specification, formal methods can be applied to JavaScript!
  Previous mechanization work: **ESMeta**, JSCert, KJS, λ_{JS} ...

↓

Implementation tests, static analyzers...

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

Missing from previous mechanization work

Previous research work [PLDI'19, POPL'22, PLDI'23] is done on paper.

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

Missing from previous mechanization work

Previous research work [PLDI'19, POPL'22, PLDI'23] is done on paper.

JavaScript regexes face concrete problems where formal methods could help

Complexity:
ReDoS (regex denial-of-service) vulnerability: most engines have exponential complexity.

```
"a".repeat(15).match(/(a*)*b/) : 1 ms
```

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

Missing from previous mechanization work

Previous research work [PLDI'19, POPL'22, PLDI'23] is done on paper.

JavaScript regexes face concrete problems where formal methods could help

Complexity:
ReDoS (regex denial-of-service) vulnerability: most engines have exponential complexity.
`"a".repeat(15).match(/(a*)*b/)` : 1 ms
`"a".repeat(35).match(/(a*)*b/)` : 12 minutes

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

Missing from previous mechanization work

Previous research work [PLDI'19, POPL'22, PLDI'23] is done on paper.

JavaScript regexes face concrete problems where formal methods could help

Complexity:
ReDoS (regex denial-of-service) vulnerability: 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!

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

Missing from previous mechanization work

Previous research work [PLDI'19, POPL'22, PLDI'23] is done on paper.

JavaScript regexes face concrete problems where formal methods could help

Complexity:
ReDoS (regex denial-of-service) vulnerability: 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!

Question: Which regexes can we match without exponential complexity?

JavaScript regexes are widely used (30% of `npm` packages) [FSE'18].

Missing from previous mechanization work

Previous research work [PLDI'19, POPL'22, PLDI'23] is done on paper.

JavaScript regexes face concrete problems where formal methods could help

Complexity:
ReDoS (regex denial-of-service) vulnerability: 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!

Question: Which regexes can we match without exponential complexity?

Subtle Semantics:
Each modern regex language is different. These semantics are hard to grasp.

Question: How do JavaScript unique semantics affect algorithms?

Question: How can we make sure our algorithms are correct?

JavaScript regexes

Our Linear Algorithms

- Adapting standard linear algorithms.
- Linear-time lookarounds.
- Implementations.

JavaScript regexes

Our Linear Algorithms

- Adapting standard linear algorithms.
- Linear-time lookarounds.
- Implementations.

Our Mechanized Semantics

- Translation of the ECMAScript regex chapter into Rocq/Coq.
- New semantics for formal verification.

JavaScript regexes

Our Linear Algorithms

- Adapting standard linear algorithms.
- Linear-time lookarounds.
- Implementations.

Our Mechanized Semantics

- Translation of the ECMAScript regex chapter into Rocq/Coq.
- New semantics for formal verification.

Our Formal Verification Work

- Properties of the semantics.
- Linear matching algorithm.
- Regex equivalence.

JavaScript regexes

A quick refresher

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.

`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`
Disjunction	`r1 | r2`
Iteration	`r*`

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`
Disjunction	`r1 | r2`			
Iteration	`r*`			

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.

`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`	
Disjunction	`r1 | r2`				
Iteration	`r*`	`r+`	`r*?`	`r{n,m}`	⟶ Matches `r` from `n` to `m` times.

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`
Disjunction	`r1 | r2`			
Iteration	`r*`	`r+`	`r*?`	`r{n,m}`
Capture Groups	`(r)`			

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

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`
Disjunction	`r1 | r2`			
Iteration	`r*`	`r+`	`r*?`	`r{n,m}`
Capture Groups	`(r)`			
Anchors	`^`	`$`		

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`
Disjunction	`r1 | r2`			
Iteration	`r*`	`r+`	`r*?`	`r{n,m}`
Capture Groups	`(r)`			
Anchors	`^`	`$`		
Lookarounds	`(?=r)`	`(?!r)`		

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

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

Characters	`a`	`[a-z]`	`\d`	`.`
Disjunction	`r1 | r2`			
Iteration	`r*`	`r+`	`r*?`	`r{n,m}`
Capture Groups	`(r)`			
Anchors	`^`	`$`		
Lookarounds	`(?=r)`	`(?!r)`	`(?<=r)`	`(?<!r)`

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

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

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

Backtracking Semantics

Backtracking Semantics: Return the first match found by a backtracking algorithm.
`"abcd".match(/ab*|abc/) = "ab"` (same in Perl, Python, Java, .NET, Go, Rust).

Features (non-exhaustive)

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

Flags

i	Case Insentivity
u/v	Unicode Mode
m	Multiline Mode
g	Global
d	Return Group Indices
y	Sticky Matching
s	Dot Matches All

Features

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

How complex are these features?

Features

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

Backreferences make matching NP-hard
So we need an exponential algorithm.
But most engines are exponential even without backreferences! (e.g. `/(a*)*b/`)

Features

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

Some engines are safe against ReDoS!

Without backreferences or lookarounds, you can use linear-time algorithms!
RE2, Rust, Go, .NET, RE#, Hyperscan...

Backreferences make matching NP-hard.
So we need an exponential algorithm.
But most engines are exponential even without backreferences! (e.g. `/(a*)*b/`)

Features

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

Some engines are safe against ReDoS!

Without backreferences or lookarounds, you can use linear-time algorithms!
RE2, Rust, Go, .NET, RE#, Hyperscan...

Backreferences make matching NP-hard.
So we need an exponential algorithm.
But most engines are exponential even without backreferences! (e.g. `/(a*)*b/`)

For JavaScript: the V8 *Experimental* linear engine

An alternative engine behind a flag. PikeVM algorithm.
`"a".repeat(100).match(/(a*)*b/l)` : 0.01 ms.

Linear matching of JavaScript regexes

JS Regex Feature	State of the Art
Quantifiers (*,+)	incorrect

Adapting known algorithms for JavaScript

The PikeVM algorithm does not always work for JavaScript quantifiers!

JS Regex Feature	State of the Art	Our algorithms
Quantifiers (*,+)	incorrect	$O(\|r\| \times \|s\|)$

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

Adapting known algorithms for JavaScript

The PikeVM algorithm does not always work for JavaScript quantifiers!
We found an algorithm tweak that works, and implemented it in the V8 linear engine.

JS Regex Feature	State of the Art	Our algorithms
Quantifiers (*,+)	incorrect	$O(\|r\| \times \|s\|)$
Lookarounds	unsupported	$O(\|r\| \times \|s\|)$

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

Adapting known algorithms for JavaScript

The PikeVM algorithm does not always work for JavaScript quantifiers!
We found an algorithm tweak that works, and implemented it in the V8 linear engine.

Extending the set of linear features

We designed a linear algorithm for lookarounds!
Supports capture groups inside lookarounds (only for JavaScript).

JS Regex Feature	State of the Art	Our algorithms
Quantifiers (*,+)	incorrect	$O(\|r\| \times \|s\|)$
Lookarounds	unsupported	$O(\|r\| \times \|s\|)$

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

Adapting known algorithms for JavaScript

The PikeVM algorithm does not always work for JavaScript quantifiers!
We found an algorithm tweak that works, and implemented it in the V8 linear engine.

Extending the set of linear features

We designed a linear algorithm for lookarounds!
Supports capture groups inside lookarounds (only for JavaScript).

Without backreferences, we can match all JavaScript regexes with string-size linear complexity.

JS Regex Feature	State of the Art	Our algorithms
Quantifiers (*,+)	incorrect	$O(\|r\| \times \|s\|)$
Lookarounds	unsupported	$O(\|r\| \times \|s\|)$
Capture Groups in Quantifiers	$O(\|r\|^2 \times \|s\|)$	$O(\|r\| \times \|s\|)$
Nonnullable Plus	$O(2^{\|r\|} \times \|s\|)$	$O(\|r\| \times \|s\|)$
Nullable Greedy Plus	$O(2^{\|r\|} \times \|s\|)$	$O(\|r\| \times \|s\|)$

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

Adapting known algorithms for JavaScript

The PikeVM algorithm does not always work for JavaScript quantifiers!
We found an algorithm tweak that works, and implemented it in the V8 linear engine.

Extending the set of linear features

We designed a linear algorithm for lookarounds!
Supports capture groups inside lookarounds (only for JavaScript).

Without backreferences, we can match all JavaScript regexes with string-size linear complexity.

We also investigated regex-size complexity.

Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

JavaScript: Iterations of the $*$ cannot match empty.

$$(\quad (a \mid \varepsilon) \quad (\varepsilon \mid b) \quad)^* \quad \text{on "ab"}$$

Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

JavaScript: Iterations of the $*$ cannot match empty.

$$(\quad (a \mid \varepsilon) \quad (\varepsilon \mid b) \quad)^* \quad \text{on "ab"}$$

$\xrightarrow[a]{}$

Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

JavaScript: Iterations of the $*$ cannot match empty.

$$(\quad \underset{a}{\overset{(a \mid \varepsilon)}{\longrightarrow}} \quad \underset{\varepsilon}{\overset{(\varepsilon \mid b)}{\longrightarrow}} \quad)^* \quad \text{on "ab"}$$

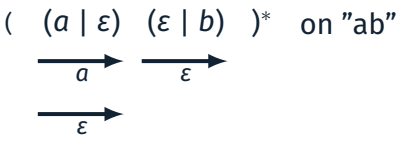
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

JavaScript: Iterations of the $*$ cannot match empty.



Preventing infinite repetition

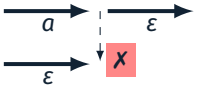
With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

JavaScript: Iterations of the $*$ cannot match empty.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on "ab"



Preventing infinite repetition

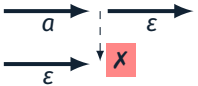
With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

JavaScript: Iterations of the $*$ cannot match empty.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on ”ab”



Result: 1 iteration, match ”a”.

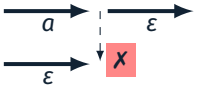
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 1 iteration, match "a".

JavaScript: Iterations of the $*$ cannot match empty.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"

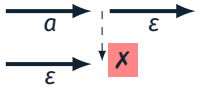
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 1 iteration, match "a".

JavaScript: Iterations of the $*$ cannot match empty.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"

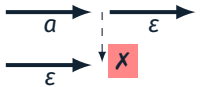
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on ”ab”



Result: 1 iteration, match ”a”.

JavaScript: Iterations of the $*$ cannot match empty.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on ”ab”

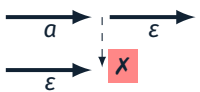
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on ”ab”



Result: 1 iteration, match ”a”.

JavaScript: Iterations of the $*$ cannot match empty.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on ”ab”

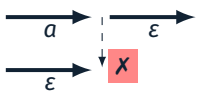
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on "ab"



Result: 1 iteration, match "a".

JavaScript: Iterations of the $*$ cannot match empty.

$($ $(a \mid \varepsilon)$ $(\varepsilon \mid b)$ $)^*$ on "ab"

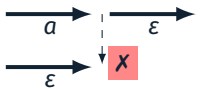
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

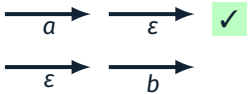
($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 1 iteration, match "a".

JavaScript: Iterations of the $*$ cannot match empty.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



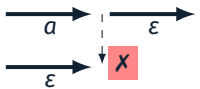
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

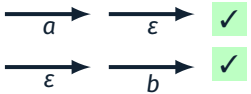
($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 1 iteration, match "a".

JavaScript: Iterations of the $*$ cannot match empty.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



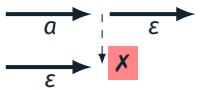
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

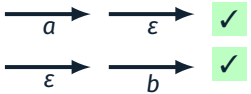
($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 1 iteration, match "a".

JavaScript: Iterations of the $*$ cannot match empty.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 2 iterations, match "ab".

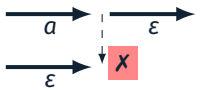
Preventing infinite repetition

With backtracking semantics, quantifiers are iterated as much as possible.
Semantics need to prevent infinite loops in cases like `/()*/`.

Two Ways of Avoiding Infinite Loops

Most languages: ε-loops are forbidden.

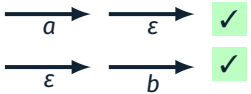
($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 1 iteration, match "a".

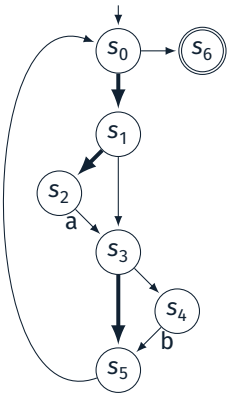
JavaScript: Iterations of the $*$ cannot match empty.

($(a \mid \varepsilon)$ $(\varepsilon \mid b)$)* on "ab"



Result: 2 iterations, match "ab".

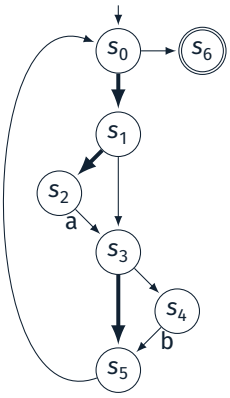
We added this example to Test262.



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

PikeVM algorithm

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

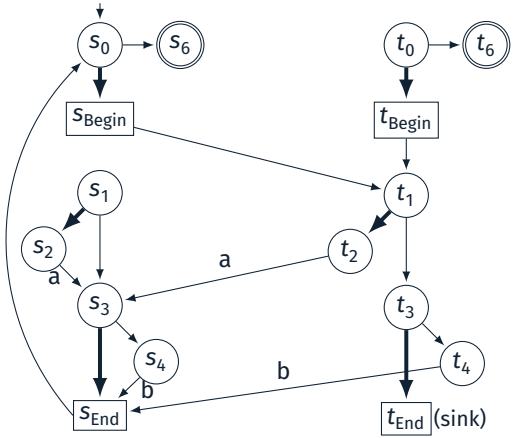


NFA for $((a \mid \varepsilon)\,(\varepsilon \mid b))^*$
with priority edges (**bold**).

PikeVM algorithm

- 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!

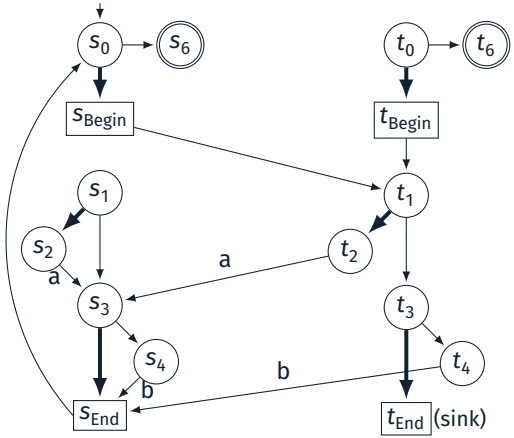


You can exit the star.

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.



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 semantics!
Still linear (even for nested stars).

Matching JavaScript Lookarounds in Linear Time

Linear engines do not support lookarounds.

But the semantics of EcmaScript makes it possible.

Key Insight 1 - Pre-computation

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

Key Insight 1 - Pre-computation

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

Our three-steps linear algorithm

- Pre-compute a table.

Key Insight 1 - Pre-computation

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

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

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

- Pre-compute a table.

Key Insight 1 - Pre-computation

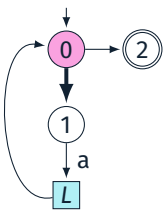
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

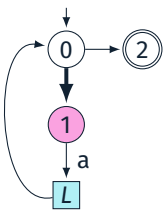
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ \ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

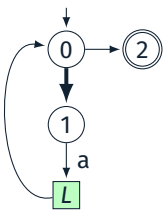
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

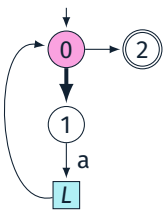
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

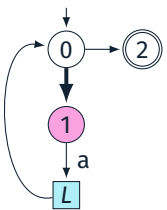
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ \ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

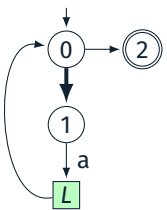
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

	a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

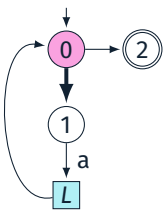
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

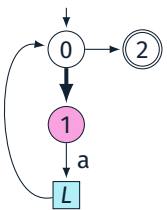
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ \ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

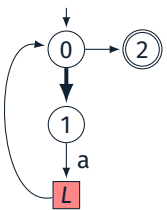
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

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

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

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

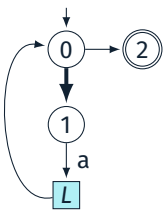
Example: $(a$ (?=(a | b)) $)^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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

What if lookarounds have capture groups?



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]
```

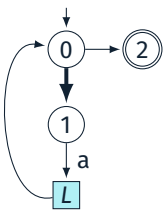
Example: $(a \;\; (?=(a \mid b)) \;)^*$ on string "aaac".

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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

What if lookarounds have capture groups?



Key Insight 1 - Pre-computation

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

Our three-steps linear algorithm

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

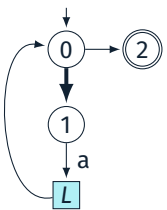
What if lookarounds have capture groups?

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.

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

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗



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.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

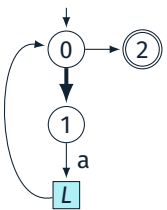
	a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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

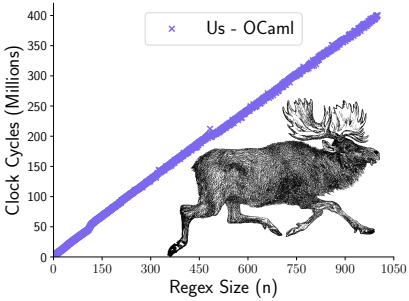
What if lookarounds have capture groups?

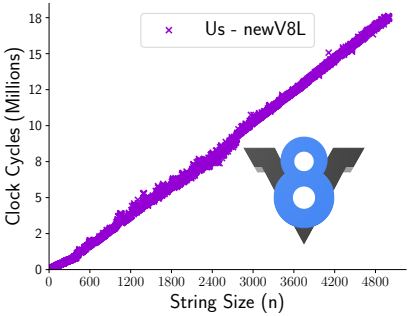
- Reconstruct groups in lookarounds:
  Match each lookaround once
  from where they were last used.



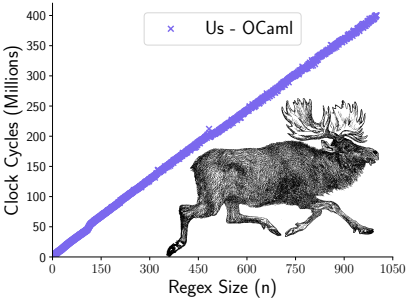
We implemented all our algorithms in our OCaml engine for JavaScript regexes (RegElk).
With Ludovic Mermod : we implemented most algorithms in V8.

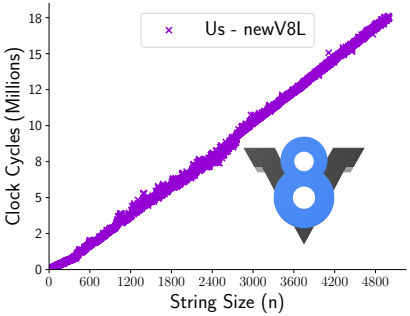
We implemented all our algorithms in our OCaml engine for JavaScript regexes (RegElk).
With Ludovic Mermod : we implemented most algorithms in V8.





We implemented all our algorithms in our OCaml engine for JavaScript regexes (RegElk).
With Ludovic Mermod : we implemented most algorithms in V8.





If lookbehinds do not have capture groups, our algorithm works for any regex language!

With Erik Giorgis : RE2 fork.

With Marcin Wojnarowski and Robin Hänni : Rust Regex fork.

Mechanized Semantics for JavaScript regexes

Linear algorithms are VERY different from the specification

Initial peer review: *"There may be problems in the techniques presented in Section 4.1, 4.4 and 4.5."*

Linear algorithms are VERY different from the specification

Initial peer review: *"There may be problems in the techniques presented in Section 4.1, 4.4 and 4.5."*

How can we formally reason about JavaScript regexes and their algorithms?
Let's **formally verify linear-time algorithms**.

Linear algorithms are VERY different from the specification

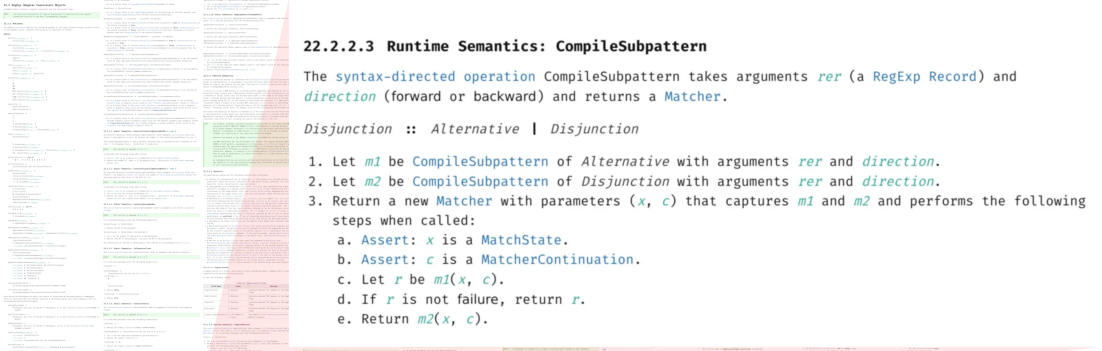
Initial peer review: *"There may be problems in the techniques presented in Section 4.1, 4.4 and 4.5."*

How can we formally reason about JavaScript regexes and their algorithms?
Let's **formally verify linear-time algorithms**.

Mechanizing the semantics

We need a version of the ECMAScript regex chapter in a proof assistant (Rocq).

The Warblre mechanization: we manually translated the regex chapter into Rocq definitions!



22.2.2.3 Runtime Semantics: CompileSubpattern

The syntax-directed operation CompileSubpattern takes arguments *rer* (a RegExp Record) and *direction* (forward or backward) and returns a Matcher.

Disjunction :: *Alternative* | *Disjunction*

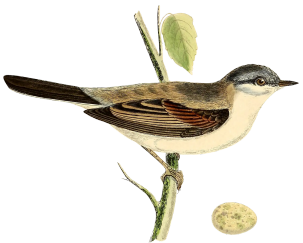
1. Let *m1* be CompileSubpattern of *Alternative* with arguments *rer* and *direction*.
2. Let *m2* be CompileSubpattern of *Disjunction* with arguments *rer* and *direction*.
3. Return a new Matcher with parameters (*x*, *c*) that captures *m1* and *m2* and performs the following steps when called:
   a. Assert: *x* is a MatchState.
   b. Assert: *c* is a MatcherContinuation.
   c. Let *r* be *m1*(*x*, *c*).
   d. If *r* is not failure, return *r*.
   e. Return *m2*(*x*, *c*).

33 pages of specification

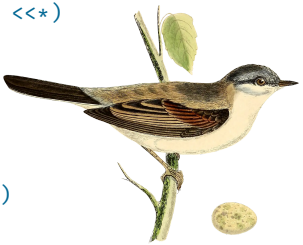
22.2.2.4.1 IsWordChar (rer, Input, e)

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.

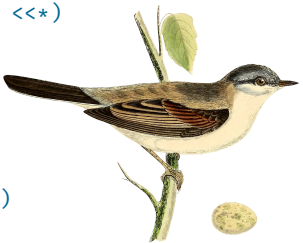


```
(** >>
    22.2.2.4.1 IsWordChar ( rer, Input, e )

    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. <<*)
```



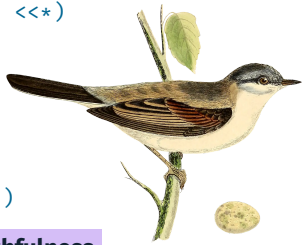
Translation done by Noé De Santo.

```
(** >>
    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:
<<*)
Definition isWordChar (rer: RegExpRecord) (Input: list Character) (e: integer): Result bool :=
  (*>> 1. Let InputLength be the number of elements in Input. <<*)
  let InputLength := List.length Input in
  (*>> 2. If e = -1 or e = InputLength, return false. <<*)
  if (e =? -1)%Z || (e =? InputLength)%Z then false
  else
  (*>> 3. Let c be the character Input[ e ]. <<*)
  let! c =<< Input[e] in
  (*>> 4. If WordCharacters(rer) contains c, return true. <<*)
  let! wc =<< wordCharacters rer in
  if CharSet.contains wc c then true
  else
  (*>> 5. Return false. <<*)
  false.
```



```
(** >>
    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:
<<*)
Definition isWordChar (rer: RegExpRecord) (Input: list Character) (e: integer): Result bool :=
  (*>> 1. Let InputLength be the number of elements in Input. <<*)
  let InputLength := List.length Input in
  (*>> 2. If e = -1 or e = InputLength, return false. <<*)
  if (e =? -1)%Z || (e =? InputLength)%Z then false
  else
  (*>> 3. Let c be the character Input[ e ]. <<*)
  let! c =<< Input[e] in
  (*>> 4. If WordCharacters(rer) contains c, return true. <<*)
  let! wc =<< wordCharacters rer in
  if CharSet.contains wc c then true
  else
  (*>> 5. Return false. <<*)
  false.
```

Translation done by Noé De Santo.



Checking faithfulness

With Martin Crettol: we built the SpecMerger tool, comparing Rocq comments to the ECMAScript HTML.

Some things are not easy to translate into Rocq:

CountLeftCapturingParensBefore (*node*)

1. Assert: *node* is an instance of a production in the RegExp Pattern grammar.
2. Let *pattern* be the *Pattern* containing *node*.
3. Return the number of *Atom* :: `(` $GroupSpecifier_{opt}$ *Disjunction* `)` Parse Nodes contained within *pattern* that either occur before *node* or contain *node*.

Some things are not easy to translate into Rocq:

CountLeftCapturingParensBefore (*node*)

1. Assert: *node* is an instance of a production in the RegExp Pattern grammar.
2. Let *pattern* be the *Pattern* containing *node*.
3. Return the number of *Atom* :: ($GroupSpecifier_{opt}$ *Disjunction*) Parse Nodes contained within *pattern* that either occur before *node* or contain *node*.

We need more information:

- the original regex (”Pattern containing node”)
- the position of the current node within the original regex

Some things are not easy to translate into Rocq:

CountLeftCapturingParensBefore (*node*)

1. Assert: *node* is an instance of a production in the RegExp Pattern grammar.
2. Let *pattern* be the *Pattern* containing *node*.
3. Return the number of *Atom* :: ($GroupSpecifier_{opt}$ *Disjunction*) Parse Nodes contained within *pattern* that either occur before *node* or contain *node*.

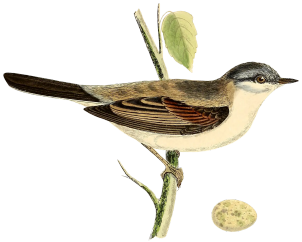
We need more information:

- the original regex (”Pattern containing node”)
- the position of the current node within the original regex

```
Definition countLeftCapturingParensBefore (node) (ctx: RegexContext):= ...
```

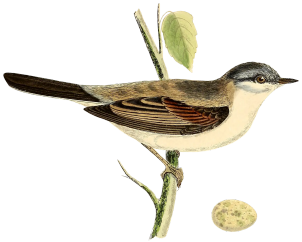
An executable mechanization

We ran the Test262 official JavaScript conformance test suite.
498 relevant regex tests.



An executable mechanization

We ran the Test262 official JavaScript conformance test suite.
498 relevant regex tests. 495 passed, 3 timeouts.



An executable mechanization

We ran the Test262 official JavaScript conformance test suite.
498 relevant regex tests. 495 passed, 3 timeouts.

Limitations

- ECMAScript 14th edition (2023): no v flag, no duplicate named groups...
- Parameterized by some unicode functions.
- We have not translated regex parsing.

An executable mechanization

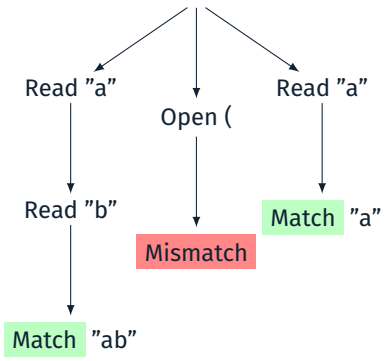
We ran the Test262 official JavaScript conformance test suite.
498 relevant regex tests. 495 passed, 3 timeouts.

Limitations

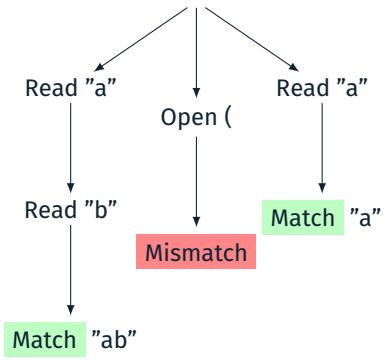
- ECMAScript 14th edition (2023): no v flag, no duplicate named groups...
- Parameterized by some unicode functions.
- We have not translated regex parsing.
- Not ideal for formal verification.

The ECMAScript style is a great intuitive specification for backtracking semantics.
But what would be the ideal semantics for formal verification?

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



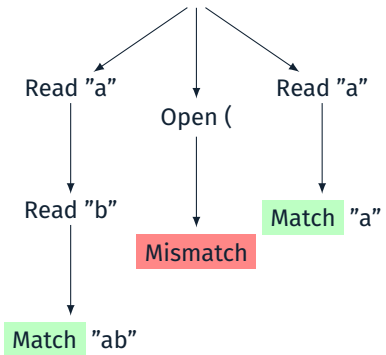
Backtracking Tree for $ab|(c)|a$ on string "ab"



Advantage #1: faithfulness

With Victor Deng, we have proved that the leftmost Match node of the tree is equal to the result of Warblre.

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



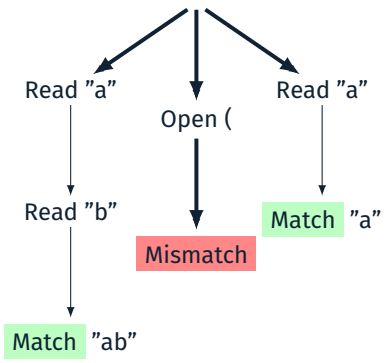
Advantage #1: faithfulness

With Victor Deng, we have proved that the leftmost Match node of the tree is equal to the result of Warblre.

Advantage #2: more behaviors

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

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



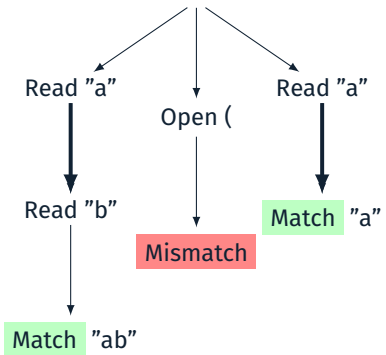
Advantage #1: faithfulness

With Victor Deng, we have proved that the leftmost Match node of the tree is equal to the result of Warblre.

Advantage #2: more behaviors

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

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



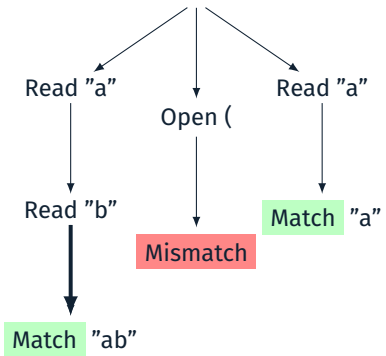
Advantage #1: faithfulness

With Victor Deng, we have proved that the leftmost Match node of the tree is equal to the result of Warblre.

Advantage #2: more behaviors

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

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



Advantage #1: faithfulness

With Victor Deng, we have proved that the leftmost Match node of the tree is equal to the result of Warblre.

Advantage #2: more behaviors

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

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

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

$$\frac{\text{advance\_backref}(gm, g, i, d) = \text{Some } (s, i') \quad (l, i', gm, d) \Downarrow t}{(\backslash g :: l, i, gm, d) \Downarrow \text{BackrefPass } s\ t}\ \textsc{Backref}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\text{check\_anchor}(a, i) = \top \quad (l, i, gm, d) \Downarrow t}{(a :: l, i, gm, d) \Downarrow \text{AnchorPass } a\ t}\ \textsc{Anchor}$$

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

Advantage #3: practicality

Only 21 inductive rules to support all features! And a convenient induction principle.

Formal Verification for JavaScript regexes

j. Let *ch* be the character *Input*[*index*].

6. Assert: $i \leq j$.

j. Let *ch* be the character *Input*[*index*]. 6. Assert: $i \leq j$.

Matching never fails

Every assertion holds, every array access is in bounds...

```
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.
```

j. Let *ch* be the character *Input*[*index*]. 6. Assert: $i \leq j$.

Matching never fails

Every assertion holds, every array access is in bounds...

```
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.
```

Matching always terminates

For each quantifier, we can compute a maximum number of iterations:

```
Theorem termination :
  (* For all valid regex r and string s *)
  ∀ r m s, compileSubPattern r = Success m →
  earlyErrors r = OK →
  (* the matcher cannot run out of fuel. *)
  m (init_state s) ≠ OutOfFuel.
```

Regex optimizers

Some optimizers, like the regex manipulation library `regexp-tree`, rewrite parts of a regex. `regexp-tree` replaces `r{min1,max1}r{min2,max2}` with `r{min1+min2,max1+max2}`.

Regex optimizers

Some optimizers, like the regex manipulation library `regexp-tree`, rewrite parts of a regex. `regexp-tree` replaces `r{min1,max1}r{min2,max2}` with `r{min1+min2,max1+max2}`.

Proving contextual equivalence

With Eugène Flesselle and Victor Deng : using our tree semantics, we can verify regex equivalence.

But we found counter-examples ! We reported the issue.

Regex optimizers

Some optimizers, like the regex manipulation library `regexp-tree`, rewrite parts of a regex. `regexp-tree` replaces `r{min1,max1}r{min2,max2}` with `r{min1+min2,max1+max2}`.

Proving contextual equivalence

With Eugène Flesselle and Victor Deng : using our tree semantics, we can verify regex equivalence.

But we found counter-examples ! We reported the issue.

We proved that under some conditions, it can be correct. And it's different inside a lookbehind!

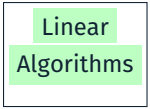
1st quantifier \ 2nd quantifier	`r{min2,min2}`	`r{0,max2}`	`r{0,max2}?`
`r{min1,min1}`	$\leftrightarrow^{\checkmark}$	$\rightarrow^{\checkmark}$ $\leftarrow^{\times}$	$\rightarrow^{\checkmark}$ $\leftarrow^{\times}$
`r{0,max1}`	$\leftarrow^{\checkmark}$ $\rightarrow^{\times}$	$\leftrightarrow^{\checkmark}$	n/a
`r{0,max1}?`	$\leftarrow^{\checkmark}$ $\rightarrow^{\times}$	n/a	$\leftrightarrow^{\times}$

The Linden project: Algorithms you can trust

We verified the PikeVM linear algorithm, used in V8's *Experimental* engine!
With the fix to support JavaScript quantifiers.

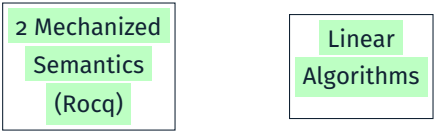
```
Theorem pike_vm_correctness:
  (* For any regex and string *)
  ∀ (r: regex) (s: string),
  (* The result of the PikeVM is equal to Warblre's *)
  pike_vm (NFA r) (init_state s) = (Warblre.compile r) s.
```

Conclusion



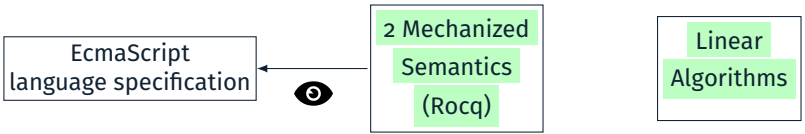
Done:

- Linear algorithms



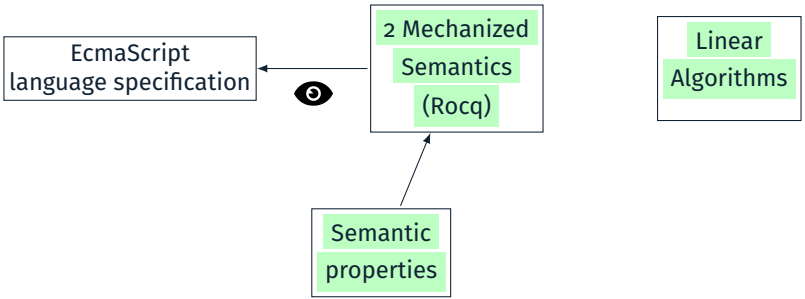
Done:

- Linear algorithms
- Rocq mechanized semantics



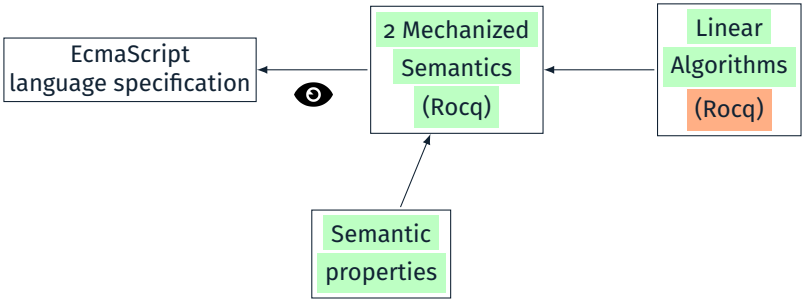
Done:

- Linear algorithms
- Rocq mechanized semantics



Done:

- Linear algorithms
- Rocq mechanized semantics
- Semantic properties

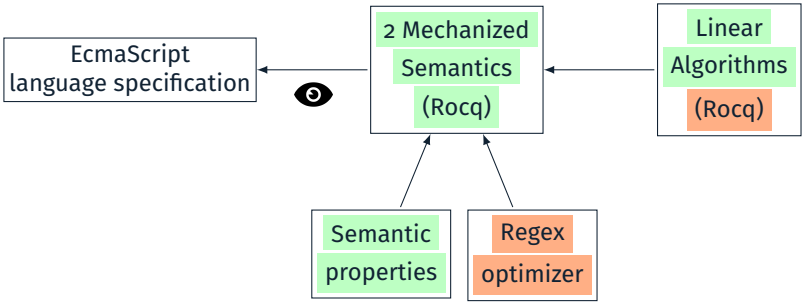


Done:

- Linear algorithms
- Rocq mechanized semantics
- Semantic properties

Ongoing:

- Algorithms proofs

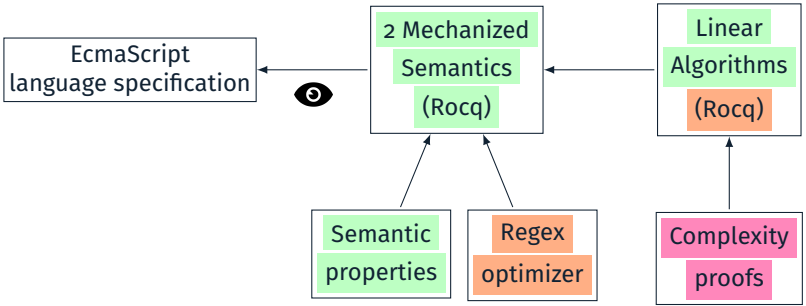


Done:

- Linear algorithms
- Rocq mechanized semantics
- Semantic properties

Ongoing:

- Algorithms proofs
- Regex equivalence proofs



Done:

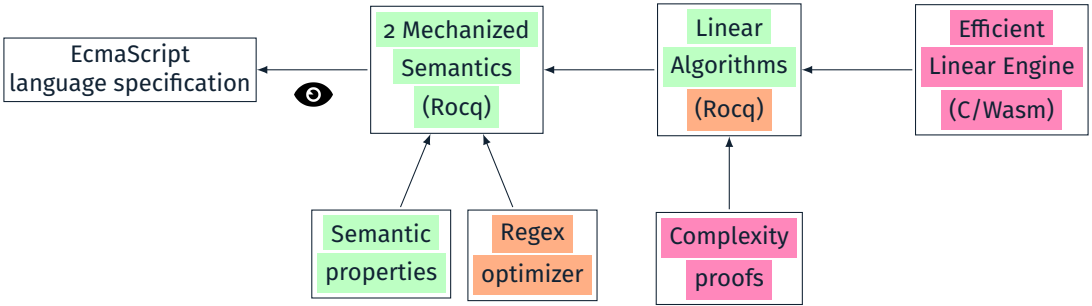
- Linear algorithms
- Rocq mechanized semantics
- Semantic properties

Ongoing:

- Algorithms proofs
- Regex equivalence proofs

Next steps:

- Update to ECMAScript 2026
- Complexity proofs



Done:

- Linear algorithms
- Rocq mechanized semantics
- Semantic properties

Ongoing:

- Algorithms proofs
- Regex equivalence proofs

Next steps:

- Update to ECMAScript 2026
- Complexity proofs
- A verified, efficient, linear JavaScript regex engine in C/Wasm.

Verify the implementation, not just the algorithm!

Generate C/Wasm code for the PikeVM engine, with a proof between ECMAScript and C/Wasm semantics.

Verify the implementation, not just the algorithm!

Generate C/Wasm code for the PikeVM engine, with a proof between ECMAScript and C/Wasm semantics.

Linearity is not speed

On average, a backtracking implementation performs much better than the PikeVM.
Other linear engines (Rust `Regex`, RE2, RE#...) achieve both linearity and efficiency.

Verify the implementation, not just the algorithm!

Generate C/Wasm code for the PikeVM engine, with a proof between ECMAScript and C/Wasm semantics.

Linearity is not speed

On average, a backtracking implementation performs much better than the PikeVM.
Other linear engines (Rust `Regex`, RE2, RE#...) achieve both linearity and efficiency.

- Let's verify common linear engine optimizations: prefix acceleration, vectorization.

Verify the implementation, not just the algorithm!

Generate C/Wasm code for the PikeVM engine, with a proof between ECMAScript and C/Wasm semantics.

Linearity is not speed

On average, a backtracking implementation performs much better than the PikeVM.
Other linear engines (Rust `Regex`, RE2, RE#...) achieve both linearity and efficiency.

- Let's verify common linear engine optimizations: prefix acceleration, vectorization.
- Let's verify other linear algorithms: DFA, OnePass, memoized backtracking.
  With Sophie Ammann: verified memoized backtracking.

Verify the implementation, not just the algorithm!

Generate C/Wasm code for the PikeVM engine, with a proof between ECMAScript and C/Wasm semantics.

Linearity is not speed

On average, a backtracking implementation performs much better than the PikeVM.
Other linear engines (Rust `regex`, RE2, RE#...) achieve both linearity and efficiency.

- Let's verify common linear engine optimizations: prefix acceleration, vectorization.
- Let's verify other linear algorithms: DFA, OnePass, memoized backtracking.
  With Sophie Ammann: verified memoized backtracking.
- Let's explore new optimizations!
  With Zacharie Tevaearai: native compilation for the PikeVM.

We want to implement a formally verified, linear-time, efficient engine for JavaScript regexes.

Linear algorithms · PLDI 24 Paper

Linear Matching of JavaScript Regular Expressions.
`https://github.com/epfl-systemf/RegElk`

Mechanized Semantics · ICFP24 Paper

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

Formal Verification · 2025 Preprint

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

We want to implement a formally verified, linear-time, efficient engine for JavaScript regexes.

Linear algorithms · PLDI 24 Paper

Linear Matching of JavaScript Regular Expressions.
`https://github.com/epfl-systemf/RegElk`

Mechanized Semantics · ICFP24 Paper

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

Formal Verification · 2025 Preprint

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

Our Questions

- How to expose linearity to users?
- What properties/algorithms would you like to see verified?
- What were the reasons behind the unique semantic choices?

Appendix

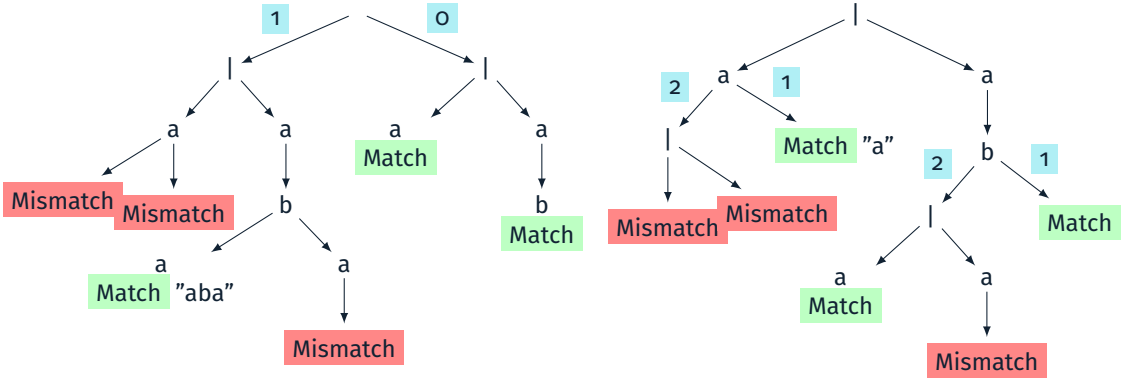
[FSE'18] The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee.

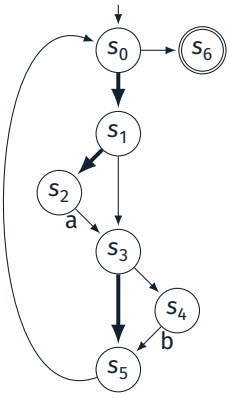
[PLDI'19] Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. Blake Loring, Duncan Mitchell, and Johannes Kinder.

[POPL'22] Solving string constraints with Regex-dependent functions through transducers with priorities and variables. Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu.

[PLDI'23] Repairing Regular Expressions for Extraction. Nariyoshi Chida and Tachio Terauchi.

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".

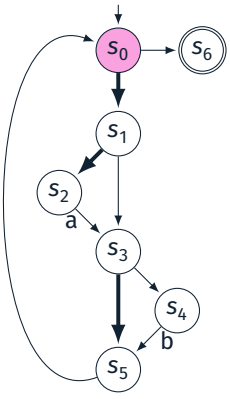




NFA for $((a \mid \varepsilon)\,(\varepsilon \mid b))^*$
with priority edges (**bold**).

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)

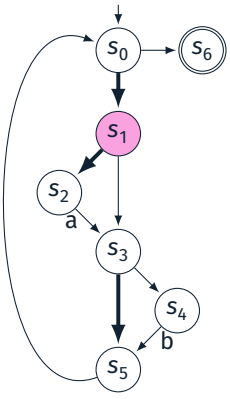


NFA for $((a \mid \varepsilon)\,(\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

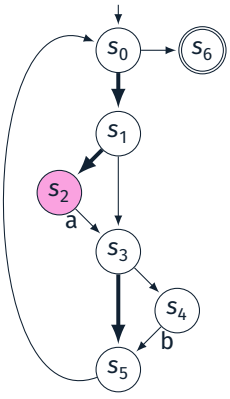


NFA for $((a \mid \varepsilon)\,(\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

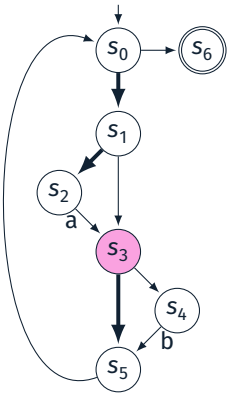


NFA for $((a \mid \varepsilon)\ (\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

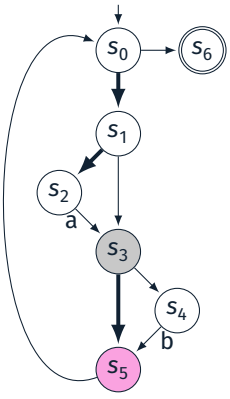


NFA for $((a \mid \varepsilon)\,(\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

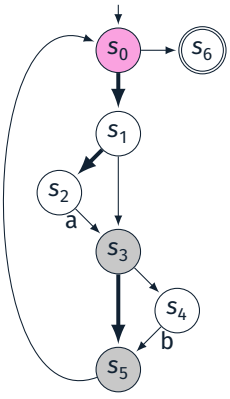


NFA for $((a \mid \varepsilon)\ (\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

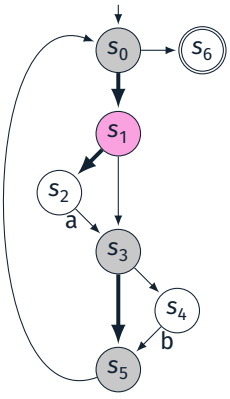


NFA for $((a \mid \varepsilon)\,(\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

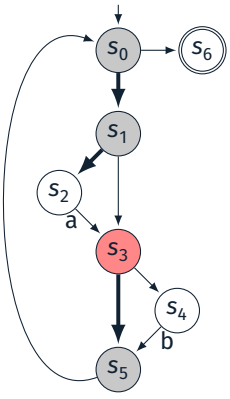


NFA for $((a \mid \varepsilon)\ (\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

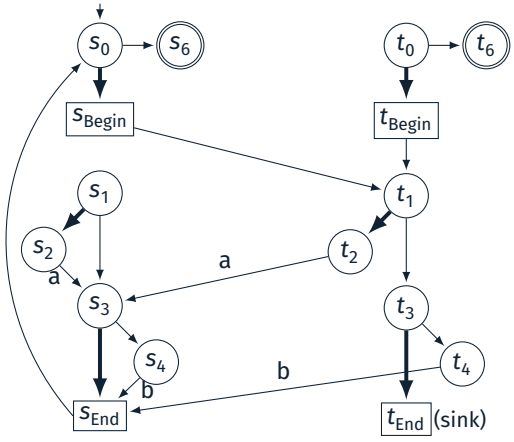


NFA for $((a \mid \varepsilon)\ (\varepsilon \mid b))^*$
with priority edges (**bold**).

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!

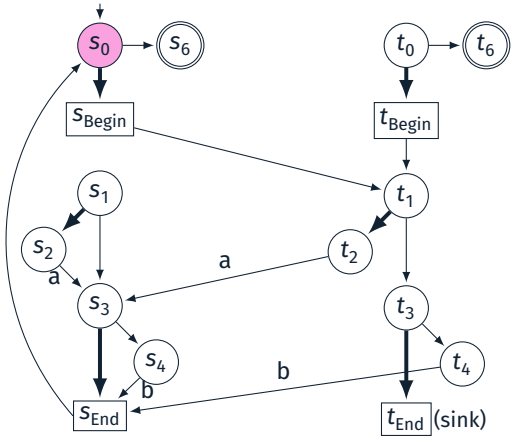


You can exit the star.

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.

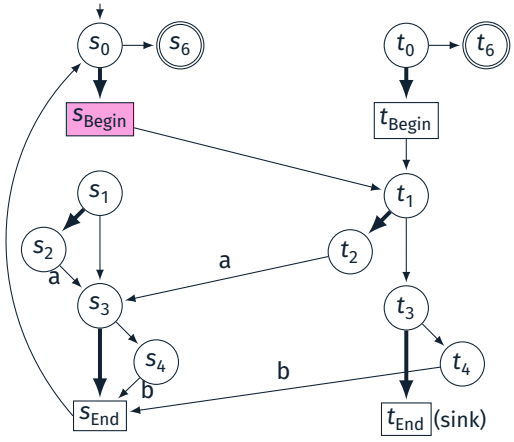


You can exit the star.

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.

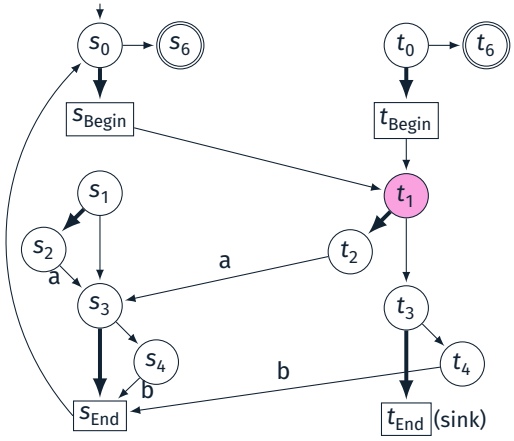


You can exit the star.

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.

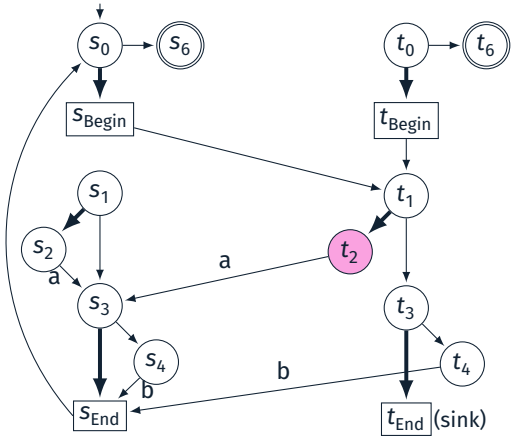


You can exit the star.

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.

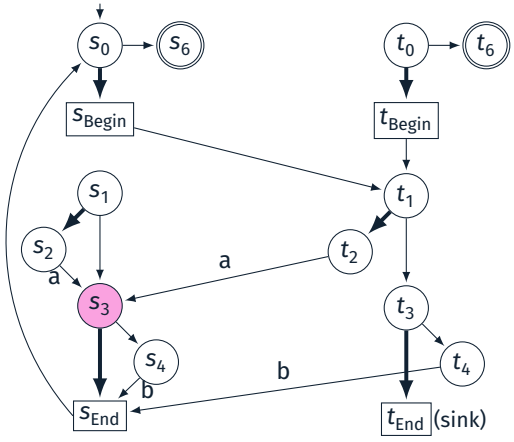


You can exit the star.

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.

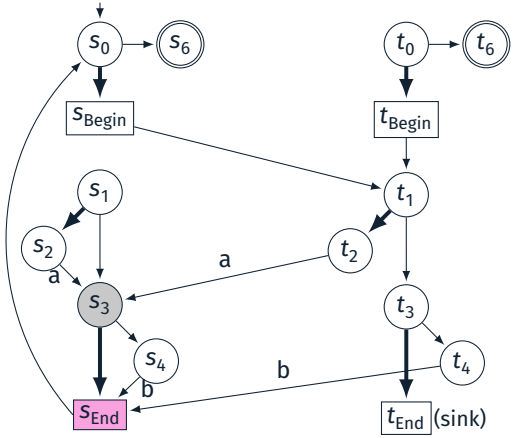


You can exit the star.

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.

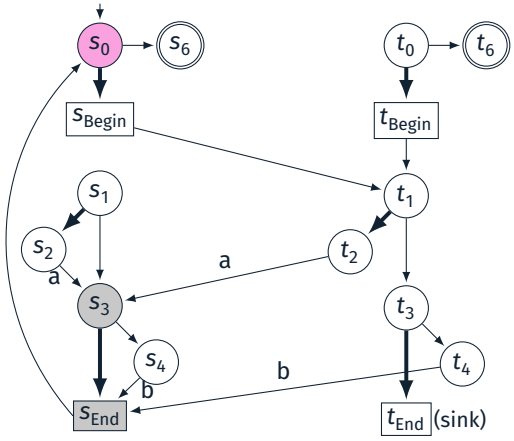


You can exit the star.

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.

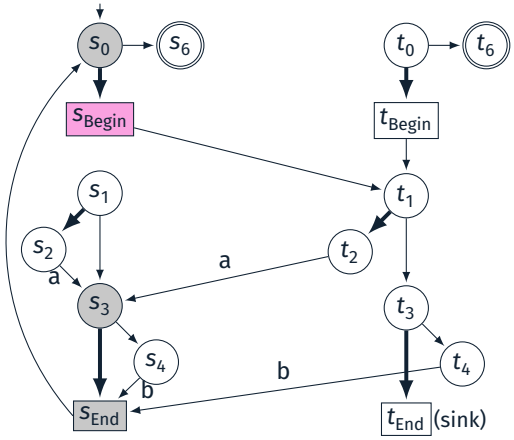


You can exit the star.

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.

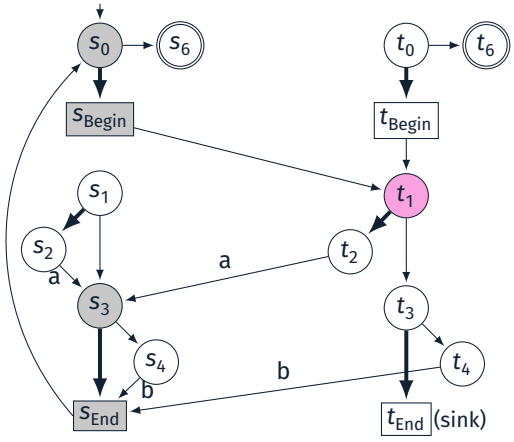


You can exit the star.

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.

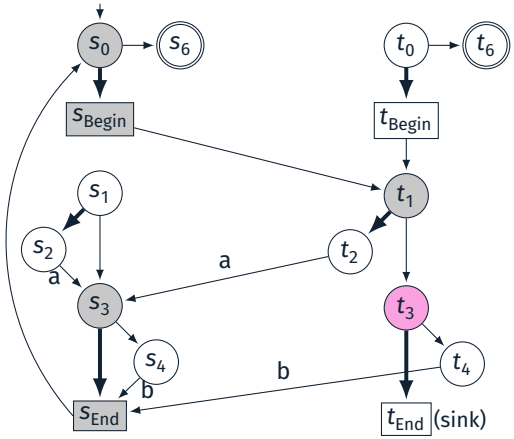


You can exit the star.

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.

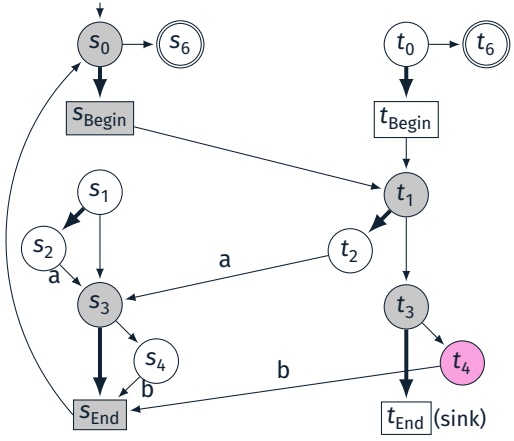


You can exit the star.

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.

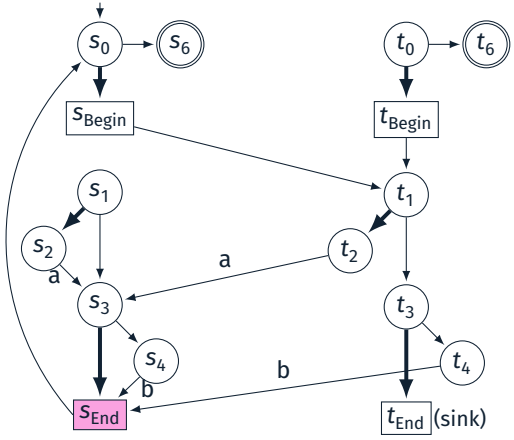


You can exit the star.

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.

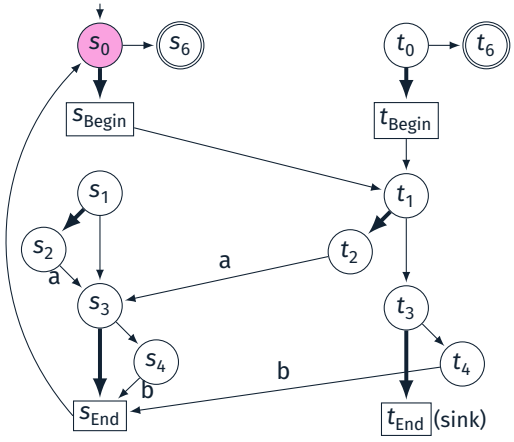


You can exit the star.

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.

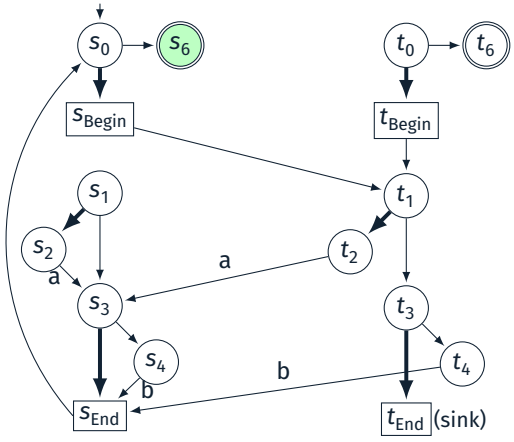


You can exit the star.

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.

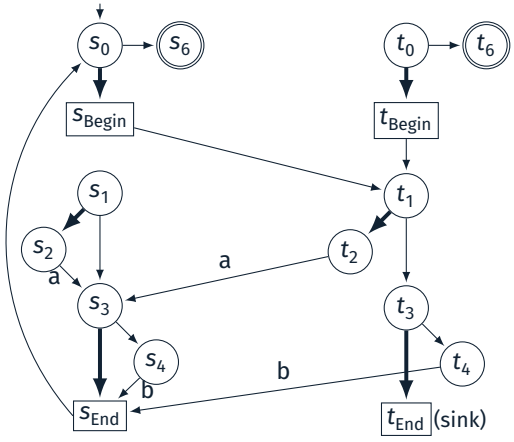


You can exit the star.

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.



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 1 - Pre-computation

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

Our three-steps linear algorithm

- Pre-compute a table.

Key Insight 1 - Pre-computation

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

Example: $(a\ (?=(a \mid b))\ )^*$ on string ”aaac”.

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

- Pre-compute a table.

Key Insight 1 - Pre-computation

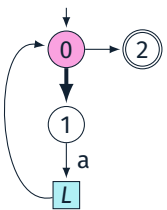
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ \text{(?=(a | b))})^*$ on string ”aaac”.

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

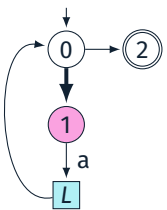
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a$ (?=(a | b)) $)^*$ on string ”aaac”.

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

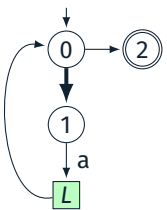
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

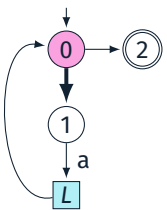
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ \text{(?=(a | b))})^*$ on string ”aaac”.

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

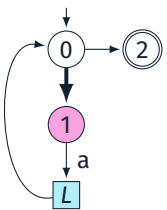
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a$ (?=(a | b)) $)^*$ on string ”aaac”.

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

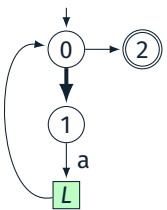
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

	a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

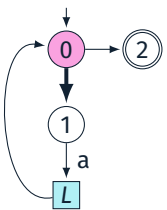
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a\ \text{(?=(a | b))})^*$ on string ”aaac”.

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

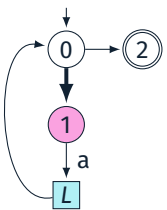
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a$ (?=(a | b)) $)^*$ on string ”aaac”.

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

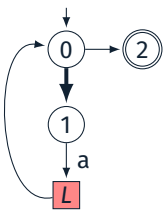
In linear time, we can pre-compute every position where each lookaround holds.
By *reversing* the regex.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

		a	a	a	c
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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



Key Insight 1 - Pre-computation

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

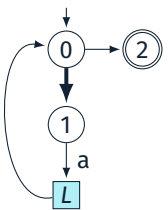
Example: $(a\ (?=(a \mid b))\ )^*$ on string "aaac".

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

Our three-steps linear algorithm

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

What if lookarounds have capture groups?



Key Insight 1 - Pre-computation

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

Our three-steps linear algorithm

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

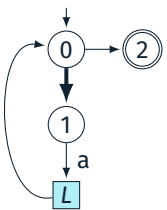
What if lookarounds have capture groups?

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.

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

	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗



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.

Example: $(a \ (?=(a \mid b))\ )^*$ on string "aaac".

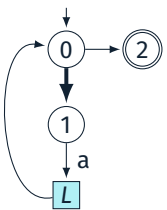
	a	a	a	c	
(a | b)	✓	✓	✓	✗	✗

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.



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.

Example: $(a\ \ (?=(a \mid b))\ )^*$ on string "aaac".

	a	a	a	c
(a | b)	✓	✓	✓	✗	✗

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.