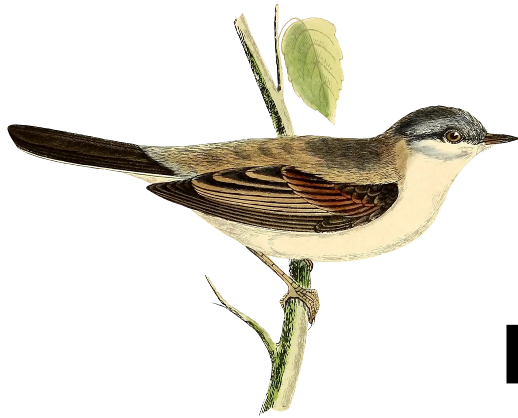
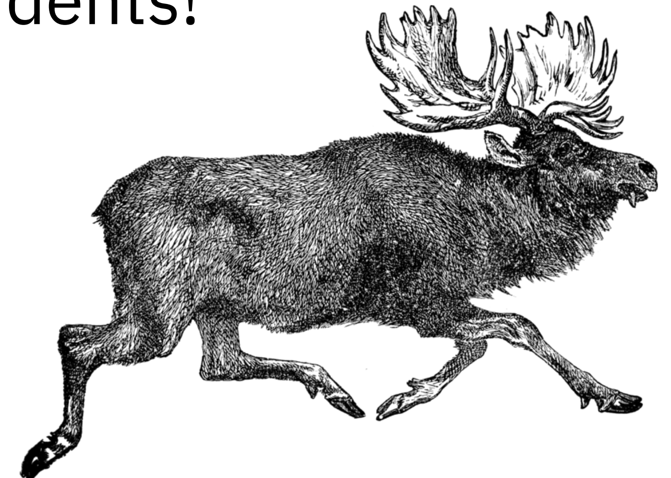


Semantics and algorithms for modern regexes

Clément Pit-Claudel, EPFL
Joint work with Aurèle Barrière, CNRS,
and many many students!

2026-04-15



Grandma's foolproof recipe to bake a verified engine

1. Find a formal regex semantics
2. Locate some nice linear-time algorithms
3. Implement fancy codegen
4. Proofs proofs proofs
5. ~~Profit!~~ Get tenure.

A quick primer on regexes

Constants: ε, `/abc/`

Sequence: $/r_1r_2/$

Alternation: $/r_1|r_2/$

Repetition: `/r*/`

Groups: `/(r)/`

…

Not a regular puzzle

Which of the following equalities hold in JavaScript?

```
1.       /a|ab/.exec('ab') =?= ['ab']
2.        /()?/.exec('')   =?= ['', '']
3.        /()|/.exec('')   =?= ['', '']
4. /(a?b??)*/.exec('ab') =?= ['ab', 'b']
```

The plan

1. What went wrong?
2. What can we do about it?
3. Some unexpected observations.
4. Help?

What went wrong: regular expressions vs regex

Just a few things:

- It’s a different problem!
- It’s a different language!
- Same syntax $\nRightarrow$ same semantics!
  - Corollary: Known algorithms can fail
- The usual assumptions don’t hold!
  - Corollary: Known techniques can fail

What went wrong: regular expressions vs regex

Just a few things:

- **It’s a different problem!**
- It’s a different language!
- Same syntax $\not\Rightarrow$ same semantics!
    - Corollary: Known algorithms can fail
- The usual assumptions don’t hold!
    - Corollary: Known techniques can fail

It’s a different problem!

Regex matching is a parsing problem
… not a decision problem

```
/^(.+)@(.+)[.](.+)$/
  .exec("cpc@epfl.ch")
    → [ "cpc@epfl.ch",
        "cpc", "epfl", "ch" ]
```

It's a different problem!

Regex matching is a parsing problem
... not a decision problem

```
/^(?<u>.+)@(?<d>.+)[.](?<ext>.+)$/
  .exec("cpc@epfl.ch").groups
    → { u: "cpc",
        d: "epfl",
        ext: "ch" }
```

It’s a different problem!

Regex matching is **an ambiguous** parsing problem
… not a decision problem

```
/^(?<u>.+)@(?<d>.+)[.](?<ext>.+)$/
  .exec("cpc@csail.mit.edu").groups
    → { u: "cpc",
        d: "csail.mit",
        ext: "edu" }
```

It’s a different problem!

Regex matching is an ambiguous parsing problem
… not a decision problem

```
/^(?<u>.+)@(?<d>.+?)[.](?<ext>.+)$/
  .exec("cpc@csail.mit.edu").groups

    → { u: "cpc",
        d: "csail",
        ext: "mit.edu" }
```

It's a different problem!

Common claim: "regex matching is NP-complete".

Wrong!

Any solution or *the highest-priority* one?

Both NP-hard, but…

`((r)|`ϵ`)` decides the negation of `r` → Co-NP?!

2025 conjecture: regex-matching is OptP-complete

(Aka: it solves Lexicographic-3SAT)

2026 result: regex-matching is PSPACE-complete.

(Aka: it solves QBF)

What went wrong: regular expressions vs regex

Just a few things:

- It’s a different problem!
- **It’s a different language!**
- Same syntax $\not\Rightarrow$ same semantics!
    - Corollary: Known algorithms can fail
- The usual assumptions don’t hold!
    - Corollary: Known techniques can fail

It’s a different language!

Lazy operators `??`, `*?`, `+?`

Lookaheads, lookbehinds (positive / negative)

```
"bananas".matchAll(/(?<=an)a/g)
"$3 = $ 3 * r $ €".matchAll(/[$](?![0-9])/g)
```

Assertions

```
/let\b/.exec("let letter = ")
```

Backreferences

```
/<(\w+)>(.*?)<[/]\1>/
```

Recursion, bounded repetitions, conditionals, possessive captures, variables, …

It’s a different language!

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])
*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t]
)*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n
)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\
031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \00
0-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?
:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@
,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))
|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]
))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))
|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|
\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r
\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\
r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(
?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)
)*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t]
)*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@
,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \0
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\
\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^
\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(
?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])
*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\
r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".
\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(
?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

https://pdw.ex-parrot.com/Mail-RFC822-Address/Mail-RFC822-Address.html

It’s a different language!

- Find duplicated words

```
/\b(\w+)\s+\1\b/
```

- Split camelCase words

```
/(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])/
```

- Tempered greedy tokens

```
/BEGIN(?:(?!END).)*END/
```

- Atomic groups (not in JS!)

```
/BEGIN(?>(?:.*?)END)(?! <skip>)/
```

- **Exercise:**

```
/^(.+?)\1+$/
```

What went wrong: regular expressions vs regex

Just a few things:

- It’s a different problem!
- It’s a different language!
- **Same syntax ⇏ same semantics!**
  - Corollary: Known algorithms can fail
- The usual assumptions don’t hold!
  - Corollary: Known techniques can fail

Same syntax ⇏ same semantics!

1. $\texttt{p|q} \not\equiv \texttt{q|p}$
2. $\texttt{p(q|r)} \not\equiv \texttt{(pq)|(pr)}$
3. $\texttt{(p|q)r} \equiv \texttt{(pr)|(qr)}$
4. $\texttt{r?} \not\equiv \texttt{r|}$
5. `r*` is stateful
6. `/((a)|(b))*/.exec("ab")`
   `→ ['ab', null, 'b']`

What went wrong: regular expressions vs regex

Just a few things:

- It’s a different problem!
- It’s a different language!
- Same syntax $\not\Rightarrow$ same semantics!
  - Corollary: Known algorithms can fail
- **The usual assumptions don’t hold!**
  - Corollary: Known techniques can fail

The usual assumptions don't hold!

- Modern regexes use backtracking
  (In specs and in implementations)
- "NFA → DFA → Transition table" is risky
  (OOM, probably)
- Lazy powerset construction? What "powerset"?
  (Powerlist, maybe? What's an additional $n!$ between 2^n friends?)

Do we care?

“Doctor, it hurts when I
study regular expressions.”

“Well, maybe don’t
study regular expressions?”

Do we care?

CLOUDFLARE

The CPU exhaustion was caused by a single WAF rule that contained a poorly written regular expression that ended up creating excessive backtracking. The regular expression that was at the heart of the outage is `(?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|true|false|null|undefined|symbol|math)|\`|\-|\+)+[)]*;?((?:\s|-|~|!|{}|\|\||\+)*.*(?:.*=.*)))`

Do we care?

Programming Techniques

This compiling scheme is very amenable to the extension of the regular expressions recognized. Special characters can be introduced to match special situations or sequences. Examples include: beginning of line character, end of line character, any character, alphabetic character, any number of spaces character, lambda, etc. It is also easy to incorporate new operators in the regular expression routine. Examples include: not, exclusive or, intersection, etc.

REFERENCES

matches the given regular expression. Examples, problems, and solutions are also presented.

KEY WORDS AND PHRASES: search, match, regular expression
CR CATEGORIES: 3.74, 4.49, 5.32

The compiler consists of three concurrently running stages. The first stage is a syntax sieve that allows only syntactically correct regular expressions to pass. This stage also inserts the operator "·" for juxtaposition of

The Linden project

Foundations and algorithms for modern regex

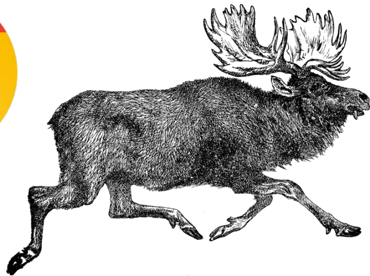
22.2.2.3 Runtime Semantics: CompileSubpattern

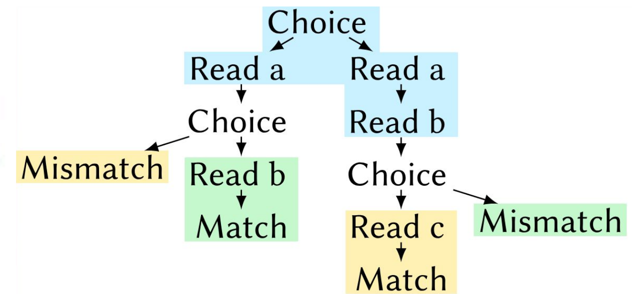
The syntax-directed operation CompileSubpattern takes arguments *rer* (a RegExp Record) and *direction* (forward or backward) and returns a Matcher.

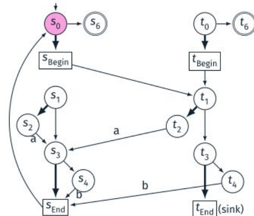
Disjunction :: *Alternative* | *Disjunction*

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*).

SpecMerger







WIP

- Algorithms:
  - What’s the complexity of the matching problem? *PSPACE*
  - What’s the right representation of groups? *Virtual trees*
  - How do we verify a full engine? *Prefix acc, MemoBT*
- Semantics:
  - Do our tree semantics generalize? *C#, XSD/I-Regexp, POSIX*
- Philosophy:
  - Is maximal munch a good thing?
  - What the right input & output syntax for grammars? *Railroads*
- Compilation
  - What a good IR for automata algorithms?
  - Can you mix simulation and backtracking?
  - Can you desynchronize VM threads?

Regex equivalence is subtle

Definition:

$$p \equiv q \coloneqq \forall\ s,\ p.\mathit{exec}(s) = q.\mathit{exec}(s)$$

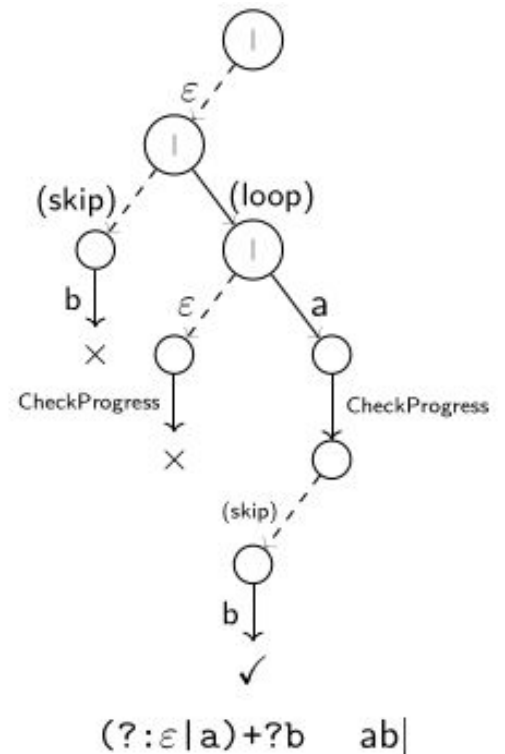
Thoughts?

By that definition:

$$(\epsilon|b) \equiv \epsilon$$ 🤔

And yet:

$$a(\epsilon|b)c \not\equiv a\epsilon c$$ 😱



Example industrial spec

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.

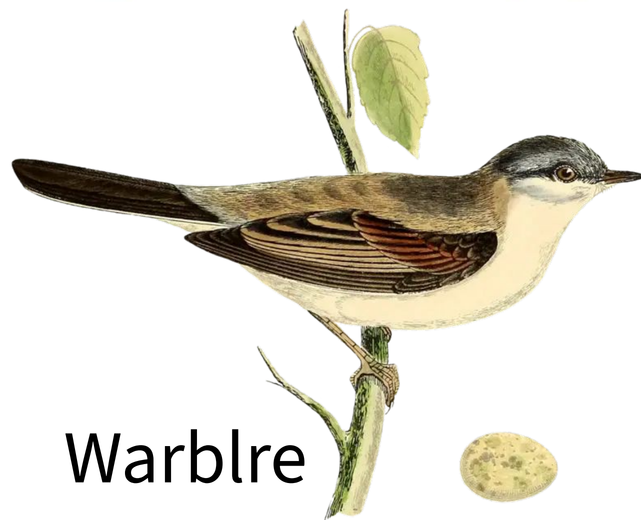
Disjunction :: *Alternative* | *Disjunction*

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

Our solution: Write ugly specs…

```
(*>> Disjunction :: Alternative | Disjunction <<*)
| Disjunction r1 r2 =>
  (*>> 1. Let m1 be CompileSubpattern of Alternative with arguments rer and direction. <<*)
  let! m1 =<< compileSubPattern r1 (Disjunction_left r2 :: ctx) rer direction in
  (*>> 2. Let m2 be CompileSubpattern of Disjunction with arguments rer and direction. <<*)
  let! m2 =<< compileSubPattern r2 (Disjunction_right r1 :: ctx) rer direction in
  (*>> 3. Return a new Matcher with parameters (x, c) that captures m1 and m2 and performs
          the following steps when called: <<*)
  (λ (x: MatchState) (c: MatcherContinuation) =>
    (*>> a. Assert: x is a MatchState. <<*)
    (*>> b. Assert: c is a MatcherContinuation. <<*)
    (*>> c. Let r be m1(x, c). <<*)
    let! r =<< m1 x c in
    (*>> d. If r is not failure, return r. <<*)
    if r is not failure then r
    (*>> e. Return m2(x, c). <<*)
    else m2 x c): Matcher
```



[De Santo, Barrière, Pit-Claudel @ ICFP’24]

... and prove pretty ones!

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

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

$$\frac{\text{advance}(cd, i, d) = \text{Some } (c, i') \qquad (l, i', gm, d) \Downarrow t}{(cd :: l, i, gm, d) \Downarrow \text{Read } c\ t}\ \textsc{Read} \qquad \frac{\text{advance}(cd, i, d) = \text{None}}{(cd :: l, i, gm, d) \Downarrow \text{Mismatch}}\ \textsc{ReadFail}$$

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

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

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

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

$$\frac{\text{advance_bkrf}(gm, g, i, d) = \text{Some } (s, i') \qquad (l, i', gm, d) \Downarrow t}{(\backslash g :: l, i, gm, d) \Downarrow \text{BackrefPass } s\ t}\ \textsc{Backref} \qquad \frac{\text{advance_bkrf}(gm, g, i, d) = \text{None}}{(\backslash g :: l, i, gm, d) \Downarrow \text{Mismatch}}\ \textsc{BackrefFail}$$

$$\frac{(r :: r\{min, \Delta, p\} :: l, i, \text{GM}_{\text{reset}}(gm, \mathcal{G}(r)), d) \Downarrow t}{(r\{min+1, \Delta, p\} :: l, i, gm, d) \Downarrow \text{Reset } \mathcal{G}(r)\ t}\ \textsc{Forced} \qquad \frac{(l, i, gm, d) \Downarrow t}{(r\{0, 0, p\} :: l, i, gm, d) \Downarrow t}\ \textsc{Done}$$

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

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

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

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

Fig. 4. Inductive tree semantics

- Just 1 page / 21 rules!
- Big-step, but *all* outcomes
- *Proved* equivalent to the original spec

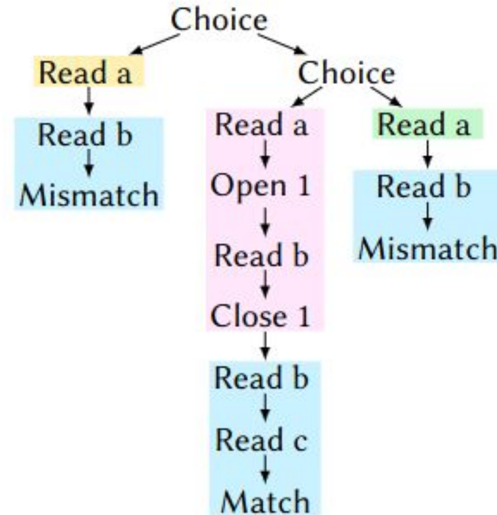


Fig. 2. Backtracking tree t of the regex $\langle a|\langle a(_1 b)|a\rangle\rangle bc$ on input "abbc"

[Barrière, Deng, Pit-Claudel @ POPL'26]