

Regular Expressions

Aayush Bajaj

July 10, 2026

Contents

1	quick reference	1
1.1	quantifiers and greed	2
2	three notations, one language class	2
2.1	thompson's construction: regex to NFA	3
2.2	subset construction: NFA to DFA, worked	3
3	what regular cannot do	4
3.1	the classic casualty: a^nb^n	5
4	engines: backtracking vs automata	5
4.1	catastrophic backtracking, measured	5
4.2	the features that break regularity	6
5	the everyday toolkit	6
5.1	grep	6
5.2	sed	7
5.3	awk	7
6	capture groups, named groups, substitution	7
7	when not to use regex	8
8	see also	9

You should not be permitted to write production code if you do not have an journeyman licence in regular expressions or floating point math.
—Rob Pike

a regular expression is two things wearing one syntax: a seventy-year-old theorem about finite automata, and the single most-used text-processing tool on unix. this page takes both seriously: the theory tells you exactly what the notation can and cannot express, and the theory's failure modes (backtracking blowups, backreference NP-hardness, the html-parsing folklore) are precisely where practitioners get burned.

kleene proved the theorem in 1951 in a paper about nerve nets, not text — the grep application came later

1 quick reference

```
1 . single character
2 \ escape
3 ? optional character
4 * zero or more of previous
5 + one ore more
```

```

6 | or on the word level
7 [] or for specific characters
8 ^ not
9 \d digit
10 \s whitespace (tabs, carriage returns, null)
11 \b word boundary
12 \d{4} equiv \d\d\d\d
13 parenthesis and ? makes the stuff in parens optional
14 same for + and *
15 \w{n} n letter words
16 \w{n,k} n or k letter words

```

1.1 quantifiers and greed

?*+ are “quantifiers”

*+ are greedy

*? is the non-greedy version

greedy quantifiers grab the longest match they can and give characters back only when forced; non-greedy ones grab the shortest and extend only when forced. the classic burn: `<.*>` applied to `<b>bold</b>` matches the *whole* string (the greedy `.*` runs to the last `>`), while `<.*?>` matches just `<b>`.

try not to use ranges: `[0-9]`. the reason is locales: under collation orders other than C, a range like `[a-z]` can match characters you did not intend (accented letters, sometimes even uppercase, depending on the locale’s sort order). posix classes `[:digit:]`, `[:alpha:]` or perl-style `\d`, `\w` say what you mean regardless of locale.

2 three notations, one language class

fix a finite alphabet Σ . **regular expressions** over Σ are built inductively: $\emptyset$, ε , and each literal $a \in \Sigma$ are regular expressions, and if r, s are, then so are the union $r \mid s$, the concatenation rs , and the kleene star r^* . that is the entire language — everything else in the quick reference above (`+`, `?`, `{n,k}`, character classes) is syntactic sugar: $r^+ = rr^*$, $r? = (r \mid \varepsilon)$, and `[abc]` is $(a \mid b \mid c)$ (Epp, Susanna S., 2019).

kleene’s theorem. the following three formalisms describe exactly the same class of languages — the *regular* languages:

- regular expressions,
- nondeterministic finite automata (NFAs), with or without ε -transitions,
- deterministic finite automata (DFAs).

each direction of the equivalence is a compilation algorithm you can actually run: regex to NFA is thompson’s construction, NFA to DFA is the subset construction, and DFA back to regex is state elimination. the string-matching-by-automaton connection is developed in (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009).

clrs chapter 32 runs the automaton idea as a string-search algorithm: compile the pattern to a dfa, then scan the text in linear time

2.1 thompson's construction: regex to NFA

compile the regex bottom-up over its parse tree; every subexpression becomes an NFA *fragment* with one start state and one accepting state, glued together with ε -transitions:¹

- **literal** a : two states joined by an edge labelled a .
- **union** $r \mid s$: a new start state with ε -edges into the fragments for r and s ; their accepting states get ε -edges into a new common accepting state.
- **concatenation** rs : the accepting state of r 's fragment is identified with (or ε -linked to) the start state of s 's fragment.
- **star** r^* : a new start-accept pair; ε -edges let you skip the fragment entirely (zero repetitions) or loop from its accept back to its start (more repetitions).

each operator adds at most two states, so a regex of length n compiles to an NFA with $O(n)$ states, each carrying at most two outgoing edges. simulating that NFA directly — carrying the *set* of live states across the text — matches a text of length m in $O(nm)$ time, no exponential anywhere.

*thompson's 1968 cacm paper
compiled the regex to ibm 7094
machine code on the fly. a jit
compiler, in 1968*

2.2 subset construction: NFA to DFA, worked

the DFA that simulates an NFA keeps one state per *set* of NFA states reachable on some input. compute only the reachable sets and the blowup is usually tame (worst case 2^n , and that worst case is achievable).

the standard example: $(a \mid b)^*abb$.¹ thompson's construction gives eleven states, numbered as in the dragon book:

0 $\xrightarrow{\text{eps}}$ 1, 7	1 $\xrightarrow{\text{eps}}$ 2, 4
2 $\xrightarrow{a}$ 3	4 $\xrightarrow{b}$ 5
3 $\xrightarrow{\text{eps}}$ 6	5 $\xrightarrow{\text{eps}}$ 6
6 $\xrightarrow{\text{eps}}$ 1, 7	7 $\xrightarrow{a}$ 8
8 $\xrightarrow{b}$ 9	9 $\xrightarrow{b}$ 10 (10 accepting)

the construction itself is a dozen lines of python — a worklist of subset states, ε -closure, move:

```
1 # thompson NFA for (a|b)*abb, dragon-book state numbering
2 eps = {0: {1, 7}, 1: {2, 4}, 3: {6}, 5: {6}, 6: {1, 7}}
3 sym = {(2, 'a'): 3, (4, 'b'): 5, (7, 'a'): 8, (8, 'b'): 9, (9, 'b'): 10}
4
5 def closure(S):
6     S, stack = set(S), list(S)
7     while stack:
8         for r in eps.get(stack.pop(), {}):
9             if r not in S:
10                 S.add(r); stack.append(r)
11     return frozenset(S)
12
13 def move(S, c):
14     return closure({sym[q, c] for q in S if (q, c) in sym})
15
16 start = closure({0})
17 names, work = {start: 'A'}, [start]
18 while work:
```

¹aho, lam, sethi and ullman, *compilers: principles, techniques, and tools* (the dragon book), 2nd ed., 2006 — section 3.7; the $(a \mid b)^*abb$ example and its state numbering are theirs.

```

19 S = work.pop(0)
20 row = [names[S], str(sorted(S))]
21 for c in 'ab':
22     T = move(S, c)
23     if T not in names:
24         names[T] = chr(ord('A') + len(names)); work.append(T)
25     row.append(names[T])
26 star = '*' if 10 in S else ' '
27 print(f"{row[0]}{star} {row[1]:<24} a->{row[2]} b->{row[3]}")

```

A	[0, 1, 2, 4, 7]	a→B	b→C
B	[1, 2, 3, 4, 6, 7, 8]	a→B	b→D
C	[1, 2, 4, 5, 6, 7]	a→B	b→C
D	[1, 2, 4, 5, 6, 7, 9]	a→B	b→E
E*	[1, 2, 4, 5, 6, 7, 10]	a→B	b→C

five subset states, E accepting (it contains NFA state 10). now look at rows **A** and **C**: identical transitions, both non-accepting — indistinguishable, so minimisation merges them. the minimal DFA has **four states**. relabelling the merged machine (A absorbs C; old D, E become C, D):

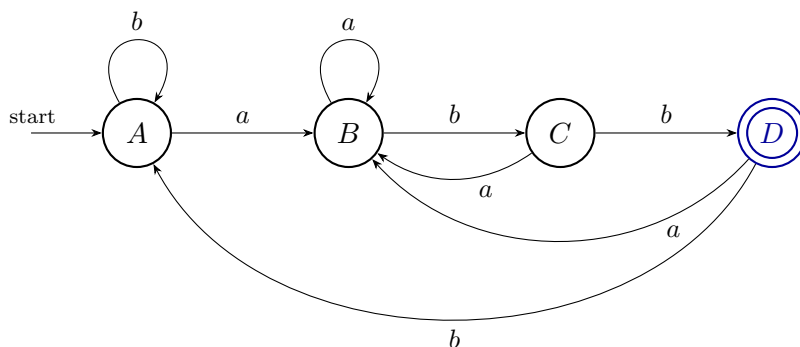


Figure 1: the minimal dfa for $(a|b)^*abb$: subset construction gives five states, merging the two indistinguishable ones leaves four. state *D* (double circle) accepts. intuition: each state remembers which prefix of *abb* you have just seen.

sanity check on *ababb*: $A \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{b} D$. accept, as it should — *ababb* ends in *abb*. and *abab* halts in *C*, non-accepting. the automaton’s state is literally “which prefix of *abb* did the last few characters spell”.

3 what regular cannot do

a DFA has finitely many states, so it can only *remember* finitely many things about the prefix it has read. the pumping lemma turns that observation into a weapon.

pumping lemma (regular languages). if L is regular, then there exists an integer $p \geq 1$ (the *pumping length*) such that every string $w \in L$ with $|w| \geq p$ can be written $w = xyz$ where

1. $|y| \geq 1$,
2. $|xy| \leq p$, and
3. $xy^iz \in L$ for every integer $i \geq 0$.

proof idea: take p to be the number of states of a DFA for L . reading the first p

characters of w visits $p+1$ states, so by pigeonhole some state repeats; the substring y consumed between the two visits traverses a loop, and the machine cannot tell whether you went around that loop zero times or a hundred.²

3.1 the classic casualty: $a^n b^n$

claim: $L = \{a^n b^n : n \geq 0\}$ is not regular. suppose it were, with pumping length p . take $w = a^p b^p \in L$; certainly $|w| = 2p \geq p$. any decomposition $w = xyz$ with $|xy| \leq p$ places x and y entirely inside the leading block of a 's: $x = a^s$, $y = a^k$ with $k \geq 1$ and $s+k \leq p$. pump up once ($i=2$):

$$xy^2z = a^{p+k} b^p \notin L \quad \text{since } k \geq 1, \quad (1)$$

contradicting condition 3. so no DFA recognises L : counting unboundedly requires unbounded memory.

*the same argument kills
balanced parentheses,
palindromes and well-nested
html. hold that thought for the
final section*

4 engines: backtracking vs automata

two implementation families, wildly different worst cases.³

- **backtracking engines** — perl, python `re`, pcre, java, javascript, ruby. compile the pattern to a tree or bytecode and run a recursive depth-first search for a match, retrying alternatives on failure. flexible (backreferences and lookahead come almost for free) but worst-case *exponential* in the input.
- **automata engines** — grep, awk, RE2, rust's `regex` crate, go's `regexp`. thompson NFA simulation with a lazily-built DFA. worst-case *linear* in the input, but they must forgo the non-regular features.

4.1 catastrophic backtracking, measured

the pattern `(a+)+$` against the string $a^n b$ is the canonical bomb: the string cannot match (it ends in `b`), but before giving up the backtracker must try every way of carving the run of a 's into “one-or-more groups of one-or-more” — and there are exponentially many carvings.

```

1 import re, time
2
3 pattern = re.compile(r'(a+)+$')
4 for n in range(16, 27, 2):
5     s = 'a' * n + 'b'
6     t0 = time.perf_counter()
7     pattern.match(s)
8     dt = time.perf_counter() - t0
9     print(f"n = {n:2d}    {dt:8.4f} s")

```

```

n = 16      0.0017 s
n = 18      0.0062 s
n = 20      0.0234 s
n = 22      0.0867 s
n = 24      0.5005 s

```

²sipser, *introduction to the theory of computation*, 3rd ed., 2013; also hopcroft and ullman, *introduction to automata theory, languages, and computation*, 1979.

³russ cox, *regular expression matching can be simple and fast*, 2007 — the essay behind RE2, with the thompson-vs-backtracking benchmark plots.

`n = 26` `1.4948 s`

adding *two characters* multiplies the time by three to four — clean exponential growth. extrapolating, $n = 40$ takes around a day and $n = 64$ outlives you. meanwhile the same query through `grep`’s automata engine, on an input nearly four times longer:

```
1 python3 -c "print('a'*100 + 'b')" > long.txt
2 time grep -cE '(a+)+$' long.txt
```

`0`

```
real 0m0.002s
user 0m0.001s
sys 0m0.001s
```

two milliseconds, mostly process startup. the asymmetry is not academic: cloudflare’s july 2019 global outage was a single catastrophically-backtracking regex in a waf rule.

the fix at scale is RE2 or the rust regex crate: refuse the non-regular features, guarantee linear time, sleep at night

4.2 the features that break regularity

- **backreferences** — `(\w+) \1` matches a repeated word. this steps outside the regular languages entirely (the language $\{ww : w \in \Sigma^*\}$ is not regular — pumping lemma again), and it costs: matching a pattern with backreferences against a string is NP-hard in general.⁴ every engine that supports them is a backtracker on those patterns, by necessity.
- **lookahead and lookbehind** — `(?=...)`, `(?!...)`, `(?<=...)`, `(?<!...)` assert without consuming. lookaround alone does not leave the regular class (regular languages are closed under intersection and complement), but backtracking implementations pay for it with extra search, and automata engines like RE2 simply refuse the syntax rather than surrender their linear-time guarantee.
- the lesson: pcre-style “regexes” are a superset of regular expressions in expressiveness and a downgrade in worst-case behaviour. know which features you are paying for.

5 the everyday toolkit

friedl’s *mastering regular expressions* is the standard practitioner’s book for everything below; the theory above is what it politely skips. all outputs below are real runs against this table:

```
1 printf 'alice 34 sydney\nbob 41 perth\ncarol 29 sydney\ndave 55 hobart\n' >
   ↪ people.txt
```

5.1 grep

`grep` prints lines matching a pattern; `-E` turns on extended syntax (the old `egrep`), `-c` counts, `-v` inverts, `-o` prints only the matching part.

⁴aho, *algorithms for finding patterns in strings*, handbook of theoretical computer science, vol. A, 1990 — matching with backreferences is NP-hard.

```

1 grep -E 'sydney$' people.txt      # lines ending in sydney
2 grep -Ec '^[a-z]+ [0-9]{2}' people.txt  # count lines shaped name-age-city

```

```

alice 34 sydney
carol 29 sydney
4

```

5.2 sed

the stream editor: apply an edit script to every line flowing past (Dougherty, Dale and Robbins, Arnold, 1997). the three idioms covering ninety percent of real usage — substitution, addressing, substitution-with-a-class:

```

1 printf 'the cat sat on the mat\nthe dog sat on the log\n' | sed 's/sat/stood/'
2 printf 'one\ntwo\nthree\nfour\n' | sed -n '2,3p'      # print only lines 2-3
3 printf 'field1,field2,field3\n' | sed -E 's/,/\t/g'    # csv to tsv

```

```

the cat stood on the mat
the dog stood on the log
two
three
field1 field2 field3

```

portability trap: in-place editing is `sed -i 's/x/y/' file` with GNU sed (linux) but `sed -i '' 's/x/y/' file` with BSD sed (macos) — BSD's `-i` requires a backup-suffix argument, so an empty one must be passed explicitly. scripts that must run on both usually write to a temp file and `mv`.

5.3 awk

awk splits each line into fields (`$1`, `$2`, ..., whitespace-delimited by default, `-F` to change) and runs *pattern { action }* rules over the stream — a full programming language that happens to fit on one line (Dougherty, Dale and Robbins, Arnold, 1997).

```

1 awk '{sum += $2} END {printf "total age: %d, mean: %.1f\n", sum, sum/NR}'
   ↪ people.txt
2 awk '$3 == "sydney" {print $1}' people.txt  # select rows, project a column
3 awk -F, '{print $2}' <<< 'a,b,c'           # comma-separated fields

```

```

total age: 159, mean: 39.8
alice
carol
b

```

the first line is `SELECT sum(age), avg(age)`; the second is a `WHERE` clause with projection. awk is relational algebra for people who refuse to leave the terminal.

6 capture groups, named groups, substitution

parentheses capture; `(?P<name>...)` captures with a name (python syntax); back-references reuse captures in the pattern or the replacement:

```

1 import re

```

```

2 log = "2026-07-07 14:32:01 ERROR disk full on /dev/sda1"
3
4
5 # positional groups
6 m = re.match(r'(\d{4})-(\d{2})-(\d{2})', log)
7 print(m.groups())
8 print(m.group(1))
9
10 # named groups
11 pat = re.compile(r'(?P<date>\S+) (?P<time>\S+) (?P<level>\w+) (?P<msg>.*\S+)')
12 m = pat.match(log)
13 print(m.group('level'), '->', m.group('msg'))
14 print(m.groupdict()['time'])
15
16 # re.sub with backreferences: yyyy-mm-dd becomes dd-mm-yyyy
17 print(re.sub(r'(\d{4})-(\d{2})-(\d{2})', r'\3-\2-\1', log))
18
19 # named backreference in pattern and replacement: collapse doubled words
20 print(re.sub(r'(?P<w>\b\w+) (?P=w)\b', r'\g<w>', "the the cat sat sat down"))

```

```

('2026', '07', '07')
2026
ERROR -> disk full on /dev/sda1
14:32:01
07-07-2026 14:32:01 ERROR disk full on /dev/sda1
the cat sat down

```

named groups cost nothing at match time and save every future reader from counting parentheses. use them in any pattern with more than two groups.

7 when not to use regex

- **html, xml, json, or anything nested.** balanced tags are the $a^n b^n$ language in a costume, and the pumping lemma already proved no regular expression recognises that. the folklore reference is the 2009 stackoverflow answer that descends into zalgo-text madness while explaining why regex cannot parse html — comedic in form, correct in substance.⁵ reach for an actual parser; every language ships one.
- **recursive structure in general** — matched parentheses, nested comments, indentation-scoped blocks. pcre's recursive patterns (`(?R)`) can technically handle some of this, at which point you have written an unreadable parser with an exponential worst case. write the readable one instead.
- **fixed-format fields** — if the input is genuinely columnar, awk field splitting or a `split(',', '')` is clearer and faster than a regex with seventeen groups.
- **when the pattern outgrows one line.** a regex you cannot re-read in six months is a liability; `re.VERBOSE` with comments is the halfway house, a real tokeniser is the cure.

the meta-rule: a regex is a specification of a *regular* language. the moment the thing you are matching needs a counter or a stack, the formalism is lying to you — it will half-work on your examples and fail in production on the nested case you did not test.

⁵stackoverflow question 1732348, answer by bobince, 2009: “you can’t parse [x]html with regex ... the center cannot hold it is too late.”

8 see also

- [linux](#) — the parent section
- [shell](#) — where these one-liners live
- [shell commands](#) — grep, sed and awk among their siblings
- [vim](#) — a second regex dialect you already half-know
- [computational complexity](#) — where claims like “NP-hard” live

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.

Dougherty, Dale and Robbins, Arnold (1997). *Sed & Awk*, O'Reilly Media.

Epp, Susanna S. (2019). *Discrete Mathematics with Applications*, Cengage Learning.