Module 10 of 16

Strings, Unicode, regular expressions and parsing

UTF-16 code units, code points and graphemes; normalisation, collation and Intl formatting; tagged templates; regular expressions from syntax to lookarounds, Unicode properties, escaping and ReDoS; tokenizers and recursive-descent parsers.

Text is where most programs meet the real world, and JavaScript's text model has a gap at its centre: a string is a sequence of UTF-16 code units, while users type characters. Every "impossible" string bug — a length of 2 for one emoji, identical-looking strings that compare unequal, a reversed name full of question marks — comes from that gap, and every fix is knowing which layer a task needs: code units, code points or grapheme clusters.

The module opens with that model and the tools at each layer, then the string toolbox with its exact semantics, tagged templates as the mechanism behind `html` / `sql` tags, and `Intl` for anything shown to a reader. Two lessons cover regular expressions: the complete syntax with flags, groups and the five methods (and the `lastIndex` trap), then the practical layer — lookarounds, `\p{L}`, function replacers, escaping input, and the catastrophic-backtracking patterns that turn one request into a denial of service. The last lesson goes past regex: a sticky-flag tokenizer, a recursive-descent parser where precedence is structural, CSV's quoting rules as a state machine, and errors that carry positions.

The exercises measure surrogate pairs and graphemes, normalise and collate, format a table with `Intl`, write `html` and `sql` tags, drive every regex method and watch `lastIndex`, extract dates with named groups, insert thousands separators with a lookahead, escape a search query, parse CSV and arithmetic — and finish with a Unicode-aware word-statistics tool, a template renderer, and a calculator with variables and right-associative exponents.

CONTENTS

1	Unicode and how JavaScript strings really work	13 min
2	The string toolbox, template literals, tagged templates and Intl formatting	12 min
3	Regular expressions — syntax, flags, groups and the five methods	14 min

4	Regex in practice — lookarounds, Unicode properties, escaping input, ReDoS, and knowing when to stop	13 min
5	Parsing text by hand — tokenizers, recursive descent, CSV and error positions	14 min
✓	Checkpoint — Strings, Unicode, regular expressions and parsing	25 min

Exercises in this module compile on Node 16. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Unicode and how JavaScript strings really work

A JavaScript string is a sequence of **UTF-16 code units**, not of characters — and every string bug that looks impossible ("length is 2 for one emoji", "reverse() broke my text", "these two identical strings are not equal") comes from that gap. This lesson gives you the three layers — code units, code points, grapheme clusters — the APIs that work at each, normalisation, case mapping and comparison across languages, and the rules for which layer a given task needs. It matters for anything that stores names, counts characters, truncates text, sorts, or searches.

Three layers

Layer	What it is	How JavaScript exposes it
Code unit	16-bit number; strings are made of these	`str.length`, `str[i]`, `charCodeAt`, `charAt`, `slice` indices
Code point	a Unicode scalar value, U+0000–U+10FFFF; one or two code units	`codePointAt`, `String.fromCodePoint`, `for…of`, `[...str]`, `\u{1F600}`, the regex `u` flag
Grapheme cluster	what a reader sees as one character (base + combining marks, emoji sequences)	`Intl.Segmenter` (granularity `grapheme`) — nothing else

```
const s = "😀";           // U+1F600
s.length;                  // 2 — two code units (a surrogate pair: 0xD83D 0xL
[...s].length;             // 1 — one code point
s.codePointAt(0).toString(16);   // "1f600";  s.charCodeAt(0) is 0xd83d — the
String.fromCodePoint(0x1f600) === s;   // true; String.fromCharCode needs both

const family = "👨‍👩‍👧";      // man + ZWJ + woman + ZWJ + girl: 5 code points, 8
[...new Intl.Segmenter("en", { granularity: "grapheme" }).segment(family)].ler
```

Code points above U+FFFF (emoji, many CJK characters, mathematical symbols) take **two** code units, a *surrogate pair*. Indexing into the middle of one gives a lone surrogate — an invalid string that prints as `�`. `"abc😀".slice(0, 4)` cuts the emoji in half;

`[...str].slice(0, n).join("")` truncates by code point; only a segmenter truncates by what the user sees.

Iteration is by code point

`for (const ch of str)`, `[...str]`, `Array.from(str)` and the regex `u` flag all step by code point. `str.split("")`, `str[i]` and a `for` loop over `length` step by code unit. Use the former for anything that must treat an emoji or a rare character as a unit; use `length`/indices only for ASCII-known data or when you truly mean code units (storage limits in UTF-16).

Combining marks and normalisation

"é" can be **one** code point (U+00E9, precomposed) or **two** (`e` U+0065 + combining acute U+0301). They render identically, compare unequal, and have different lengths:

```
"é" === "é";                          // false
"é".normalize("NFC") === "é".normalize("NFC");   // true — NFC composes; NFD c
```

Normalise (`NFC` is the usual choice) before comparing, hashing, deduplicating or storing user text; keyboards, operating systems and copy-paste produce both forms. `NFKC`/`NFKD` additionally fold compatibility characters (`ﬁ` → `fi`, full-width digits) — for search matching, not for storage.

Case mapping

`toUpperCase`/`toLowerCase` are Unicode-aware and can change **length**: `"straße".toUpperCase()` is `"STRASSE"`. They are locale-independent; `toLocaleUpperCase("tr")` turns `i` into `İ` (dotted capital I). Case-insensitive comparison in general is `a.toLowerCase() === b.toLowerCase()` for ordinary text, or `Intl.Collator` with `sensitivity: "accent"`/`"base"` for user-facing matching. Note `.toLowerCase()` on user input is not a normaliser — normalise too.

Comparing and sorting text

`<` and `sort()` compare code units — uppercase before lowercase, accents sorted to the end, `"Z" < "a"`. For human order:

```
names.sort((a, b) => a.localeCompare(b));                              // locale-
const collator = new Intl.Collator("en", { sensitivity: "base", numeric: true
names.sort(collator.compare);                                          // "file2"
```

`numeric: true` sorts embedded numbers naturally; `sensitivity: "base"` ignores case and accents. Always pass a locale in servers so results do not depend on the machine's default.

Segmentation

`Intl.Segmenter` (Node 16+, all modern browsers) splits by `grapheme`, `word` or `sentence` according to Unicode rules — the only correct way to count visible characters, to truncate "at 20 characters", or to split words in languages without spaces:

```
const words = [...new Intl.Segmenter("en", { granularity: "word" }).segment(te
```

`str.split(" ")` is fine for ASCII prose and wrong for Thai, Japanese, or text with punctuation.

Immutability and building

Strings are immutable values: every method returns a new string; `str[0] = "x"` does nothing. Repeated `+=` in a loop is fine in modern engines for moderate sizes (they use ropes internally); for very large output collect parts in an array and `join("")` once, or write to a stream. Comparing strings with `===` is by value and O(n) worst case; interning is the engine's business.

Escapes and literals

`"\n" "\t" "\\" "é" "\u{1F600}" "\x41"`; template literals keep raw newlines; `String.raw\`\n\` keeps the backslash. A string can contain lone surrogates and other invalid sequences;` str.isWellFormed() / toWellFormed() `(Node 20) detect and repair them — on 16, check with a regex for unpaired surrogates if it matters (`/\uD800-\uDBFF|(?<![\uD800-\uDBFF])[\uDC00-\uDFFF]/`).

Common mistakes

Counting or truncating with `length` / `slice` on text that may contain emoji or non-BMP characters.

`str.split("").reverse().join("")` — breaks surrogate pairs and combining sequences.

Comparing user text without `normalize("NFC")`.

`sort()` on names; `toLowerCase()` as a full case-fold; assuming case mapping preserves length.

Splitting words on spaces for non-Latin text; regexes without the `u` flag on Unicode text (next lessons).

Interview angle

"Why is `'🤪'.length` 2?" Strings are UTF-16 code units; the emoji is a surrogate pair — one code point, two units.

"How do you count characters correctly?" Code points with `[...str].length`; visible characters with `Intl.Segmenter` graphemes.

"Why are two visually identical strings unequal?" Precomposed versus combining sequences — normalise with `NFC` before comparing.

"How do you sort strings for humans?" `localeCompare` or a reused `Intl.Collator` with `numeric` / `sensitivity` options.

"Can `toUpperCase` change a string's length?" Yes — `ß` → `SS`.

Key takeaways

Code units (`length`, indices) ≠ code points (`for…of`, spread, `codePointAt`) ≠ graphemes (`Intl.Segmenter`); pick the layer the task needs.

Non-BMP characters are surrogate pairs; never slice through them; reverse and truncate by code point or grapheme.

Normalise (`NFC`) before comparing or storing; `NFKC` for search folding.

Case mapping is Unicode-aware and length-changing; locale matters (`tr`).

Sort and compare with `Intl.Collator` / `localeCompare` and an explicit locale; segment words with `Intl.Segmenter`.

EXERCISE 1

Code units, code points, graphemes

The input is a list of hexadecimal code points; build the string with `String.fromCodePoint`. Print `length=<code units> codePoints=<[...s].length> graphemes=<Intl.Segmenter grapheme count>`; `units=<each charCodeAt in hex>`; `codePoint0=<hex> charCode0=<hex> isPair=<first code point above 0xFFFF>`; `reverseNaive=<code points of s.split("").reverse().join("")>` and `reverseByCodePoint=<code points of [...s].reverse().join("")>` (both as hex code points via the starter's `hex`); and `sliceHalf=<hex of s.slice(0, 1)> at(-1)=<hex of s.at(-1)> last=<hex of [...s].at(-1)>`.

Example: `1F600 61` (😀 a) →

```
length=3 codePoints=2 graphemes=2
units=d83d de00 61
codePoint0=1f600 charCode0=d83d isPair=true
reverseNaive=61 de00 d83d
reverseByCodePoint=61 1f600
sliceHalf=d83d at(-1)=61 last=61
```

The naive reverse produced two lone surrogates; `slice(0, 1)` cut the emoji in half.

INPUT 1

```
1F600 61
```

OUTPUT 1

```
length=3 codePoints=2 graphemes=2
units=d83d de00 61
codePoint0=1f600 charCode0=d83d
isPair=true
reverseNaive=61 de00 d83d
reverseByCodePoint=61 1f600
sliceHalf=d83d at(-1)=61 last=61
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const hex = (s) => [...s].map((c) => c.codePointAt(0).toString(16)).join(" ");
const s = String.fromCodePoint(...lines[0].trim().split(/\s+/).map((h) => parseInt(h
const graphemes = [...new Intl.Segmenter("en", { granularity: "grapheme" }).segment(
console.log(`length=${s.length} codePoints=${[...s].length} graphemes=${graphemes}`)
console.log(`units=${Array.from({ length: s.length }, (_, i) => s.charCodeAt(i).toSt
console.log(`codePoint0=${s.codePointAt(0).toString(16)} charCode0=${s.charCodeAt(0)
const naive = s.split("").reverse().join("");
const byCodePoint = [...s].reverse().join("");
console.log(`reverseNaive=${hex(naive)}`);
console.log(`reverseByCodePoint=${hex(byCodePoint)}`);
console.log(`sliceHalf=${hex(s.slice(0, 1))} at(-1)=${hex(s.at(-1))} last=${hex([...
```

EXERCISE 2

Normalise, then compare like a human

Line 1 holds the same word twice — precomposed and decomposed (`é` as U+00E9 versus `e` + U+0301). Line 2 is a word list. Print `equal=<a === b> normalizedEqual=<NFC forms equal> lengths=<a.length>,<b.length> nfdLengths=<NFD lengths>`; `sortDefault=<[...words].sort()>`; `sortCollator=<sorted with new Intl.Collator("en", { numeric: true }).compare>`; `baseEqual(Apple,apple)=<bool> baseEqual(e,é)=<bool>` using a collator with `sensitivity: "base"`; and `upper=<"straße".toUpperCase()> lengthChange=<before⟶<after> turkishI=<"i".toLocaleUpperCase("tr") code point in hex>`.

Example: `café café` / `banana Apple apple Éclair zebra file10 file2` →

```
equal=false normalizedEqual=true lengths=4,5 nfdLengths=5,5
sortDefault=Apple apple banana file10 file2 zebra Éclair
sortCollator=apple Apple banana Éclair file2 file10 zebra
baseEqual(Apple,apple)=true baseEqual(e,é)=true
upper=STRASSE lengthChange=6→7 turkishI=130
```

Default sort put the accented word last and `file10` before `file2`; the collator fixed both.

INPUT 1

```
café café
banana Apple apple Éclair zebra file10
file2
```

OUTPUT 1

```
equal=false normalizedEqual=true
lengths=4,5 nfdLengths=5,5
sortDefault=Apple apple banana file10
file2 zebra Éclair
sortCollator=apple Apple banana Éclair
file2 file10 zebra
baseEqual(Apple,apple)=true
baseEqual(e,é)=true
upper=STRASSE lengthChange=6→7
turkishI=130
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const [a, b] = lines[0].trim().split(/\s+/);      // the same word, precomposed and
const words = lines[1].trim().split(/\s+/);
console.log(`equal=${a === b} normalizedEqual=${a.normalize("NFC") === b.normalize('
console.log(`sortDefault=${[...words].sort().join(" ")}`);
const collator = new Intl.Collator("en", { numeric: true });
console.log(`sortCollator=${[...words].sort(collator.compare).join(" ")}`);
const base = new Intl.Collator("en", { sensitivity: "base" });
console.log(`baseEqual(Apple,apple)=${base.compare("Apple", "apple") === 0} baseEquc
const sharp = `stra${String.fromCodePoint(0xdf)}e`;
console.log(`upper=${sharp.toUpperCase()} lengthChange=${sharp.length}→${sharp.toUp
```

QUIZ

1. `"😜".length` is 2 because…

A. Emoji are two characters

B. Strings are UTF-16 code units and the emoji is a surrogate pair — one code point, two units

C. It counts the variation selector

D. It is a bug

2. Two strings render identically but `===` is false. The likely cause is…

A. Different fonts

B. Precomposed versus decomposed characters — normalise both with `normalize("NFC")` before comparing

C. Trailing whitespace only

D. Case

3. `["b", "a", "B", "É"].sort()` puts `É` …

A. First

B. Last — default sort compares code units, and accented letters sit above ASCII; use `localeCompare` / `Intl.Collator`

C. Between `a` and `b`

D. It throws

4. Counting visible characters correctly needs…

A. `length`

B. `Intl.Segmenter` with `granularity: "grapheme"` — code points still split combined emoji

C. `split("")`

D. `charCodeAt`

5. `"straße".toUpperCase()` …

A. Is `"STRAßE"`

B. Is `"STRASSE"` — case mapping is Unicode-aware and can change length

C. Throws

D. Depends on the locale

ANSWER KEY · UNICODE AND HOW JAVASCRIPT STRINGS REALLY WORK

1. B `[..."😜"].length` is 1; a family emoji is several code points but one grapheme.

2. B Keyboards, OSes and copy-paste produce both forms.

3. B `numeric: true` also fixes `file10` sorting before `file2`.

4. B Code points (`[...s]`) are right for many tasks; graphemes for what the user sees.

5. B Locale matters elsewhere: `"i".toLocaleUpperCase("tr")` is `İ`.

LESSON 2 · 12 MIN

The string toolbox, template literals, tagged templates and Intl formatting

Most string work is a dozen methods used well: find, slice, split, join, trim, pad, replace. The rest is *formatting* — turning numbers, dates and lists into text that is right for a reader — which is where hand-written code goes wrong in every locale but the author's, and where `Intl` gets it right for free. This lesson covers the methods with their exact semantics (`slice` versus `substring`, `indexOf` versus `includes`, `split` corner cases), template literals and the tagged-template mechanism that lets a function process a literal (the basis of `html`, `sql` and `css` tags), and the `Intl` formatters for numbers, currencies, dates and lists.

The methods, precisely

```
s.slice(2, 5); s.slice(-3);                 // [start, end) — negatives count fr
s.substring(2, 5);                          // swaps arguments if start > end; t
s.indexOf("x"); s.lastIndexOf("x"); s.includes("x"); s.startsWith("x", from);
s.at(-1);                                   // last code unit (Node 16.6+)
s.split(",");  s.split(",", 2);  s.split("");  s.split(/\s+/);
"a,b,".split(",");                          // ["a", "b", ""] — a trailing separ
"".split(",");                              // [""] — not []
s.trim(); s.trimStart(); s.trimEnd();       // whitespace incl. Unicode spaces (
s.padStart(8, "0"); s.padEnd(20);           // to a target LENGTH (code units),
s.repeat(3);                                // RangeError for negative or Infini
s.replace("a", "b");                        // first occurrence only for a strin
s.replaceAll("a", "b");                     // all (Node 15+); with a regex it r
s.toUpperCase(); s.toLowerCase(); s.localeCompare(t); s.normalize();
s.concat(t) === s + t;  s.charAt(i) === s[i] (but "" instead of undefined out
```

`split` with a regex containing a capture group includes the captures in the result — `"a1b2c".split(/(\d)/)` is `["a", "1", "b", "2", "c"]`, occasionally useful. `replace` with a string pattern and a string replacement interprets `$` sequences (`$$` for a literal dollar) — pass a function (`() => text`) when the replacement is user data.

Template literals

```
const line = `${user.name} has ${items.length} item${items.length === 1 ? "" :
const multi = `first line
second line`;                                          // real newlines pres
`${1 + 1}`;  `${obj}`  /* calls toString */;  `${[1, 2]}` /* "1,2" */;  `${nu
```

Interpolation calls `String(value)` (via the ToPrimitive rules of module 5): objects print `[object Object]` unless they define `toString`, arrays join with commas, `undefined` prints as `undefined`. For anything shown to users, format explicitly — an interpolated `Date` prints its long default form; an interpolated number prints without separators.

Tagged templates

A function placed before a template literal receives the **literal parts** and the **values** separately, before any concatenation:

```
function html(strings, ...values) {
  return strings.reduce((out, str, i) => out + str + (i < values.length ? esca
}
const safe = html`<p>${userInput}</p>`;      // strings = ["<p>", "</p>"], val
String.raw`C:\new\table`;                    // "C:\\new\\table" — the raw tex
```

The tag decides what interpolation means: escaping (`html`), parameterising (`sql\` … WHERE id = ${id}` → placeholders + parameters, no injection), styling (`css`, `styled.div`), localisation, GraphQL parsing. `strings` has a `.raw` property with the unescaped source text. Because the literal parts are known statically, a tag can cache parsing per call site — `strings` is the same frozen array object on every evaluation of the same literal.

Intl: numbers, currency, dates, lists

```
new Intl.NumberFormat("en-US").format(1234567.891);
new Intl.NumberFormat("de-DE").format(1234567.891);
new Intl.NumberFormat("en-US", { style: "currency", currency: "EUR" }).format(
new Intl.NumberFormat("en", { notation: "compact" }).format(1_500_000);
new Intl.NumberFormat("en", { style: "percent", maximumFractionDigits: 1 }).fo
new Intl.NumberFormat("en", { style: "unit", unit: "kilometer-per-hour" }).for
new Intl.DateTimeFormat("en-GB", { dateStyle: "long", timeZone: "UTC" }).forma
new Intl.RelativeTimeFormat("en", { numeric: "auto" }).format(-1, "day");
new Intl.ListFormat("en", { type: "conjunction" }).format(["a", "b", "c"]);
new Intl.PluralRules("en").select(1);
(1234.5).toLocaleString("en-US", { minimumFractionDigits: 2 });
```

Create a formatter **once** and reuse it — construction is expensive, `format` is cheap. Always pass a locale and, for dates, a `timeZone`; defaults come from the machine and differ between your laptop and the server. `toFixed(2)` is for machine-facing decimals (and rounds half-away-from-zero on the binary value, so `(1.005).toFixed(2)` is `"1.00"`); `Intl` is for humans.

Building text

Concatenation with `+` / `+=` and template literals is fine for ordinary sizes. For thousands of pieces, `parts.push(…)` then `parts.join("\n")` is clearer and allocation-friendly; for output measured in megabytes, write to a stream. `Array.prototype.join` on numbers calls `String` on each; `[1, null, undefined].join("-")` is `"1--"` — nullish elements become empty strings.

Escaping

Text inserted into another language needs that language's escaping: HTML (`& < > " '` → entities), URLs (`encodeURIComponent` for components — never `encodeURI` for a value, never hand-rolled), shell (do not — use argument arrays), SQL (parameters, never strings), regex (`escapeRegExp`, module 10 lesson 4), JSON (`JSON.stringify`, which also gives you a correctly quoted string literal). A tagged template per target language is the tidy way to make escaping automatic.

Common mistakes

`substring` with negative arguments or reversed indices (silently "fixed") — use `slice`.

`"".split(",")` expected to be `[]`; a trailing separator's empty element ignored.

`replace` when all occurrences were meant; `$` sequences in a replacement built from user data.

Formatting numbers/dates by hand (`toFixed`, manual comma insertion) for a UI; creating an `Intl` formatter per call in a loop.

Interpolating objects/dates into templates and shipping `[object Object]` or a machine-locale date.

Escaping for the wrong target, or not at all, when building HTML/URLs.

Interview angle

"`slice` versus `substring`?" `slice` supports negatives and returns `""` for reversed indices; `substring` clamps negatives to 0 and swaps reversed indices.

"What is a tagged template?" A function called with the literal's string parts and interpolated values, so it can escape, parameterise or transform them — `html`, `sql`, `styled`.

"How do you format 1234567.891 for German users?" `new Intl.NumberFormat("de-DE").format(n)` — never manual.

"`replace` versus `replaceAll`?" String pattern: first occurrence versus all; with a regex, `replaceAll` requires `g`.

"Why cache `Intl` formatters?" Construction is slow (locale data), `format` is fast.

Key takeaways

`slice`, `includes` / `startsWith`, `split` (know the empty-string cases), `trim`, `padStart`, `replaceAll`, `at`.

Template literals interpolate via `String()`; format explicitly for users.

Tagged templates receive `(strings, ...values)` — the mechanism behind `html` / `sql` / `css` tags and `String.raw`.

`Intl.NumberFormat` / `DateTimeFormat` / `RelativeTimeFormat` / `ListFormat` / `PluralRules` with an explicit locale (and timeZone), created once.

Escape for the target language — entities, `encodeURIComponent`, parameters, `escapeRegExp` — or use a tag that does it.

EXERCISE 1

An aligned report with Intl

Each line is `name amount`. Print an aligned table: a header `Name | Amount | Share` (name column padded to the longest name, at least 4; amount column right-aligned to the widest formatted amount, at least 6), a rule of dashes, one row per entry with the amount from `Intl.NumberFormat("en-US", { style: "currency", currency: "USD" })` and the share of the total as a percent with at most one decimal (padded to 5), a `Total` row, and finally `compact=<total in compact notation> de=<total as EUR in de-DE> toFixed=<total.toFixed(2)>`.

Example: `rent 1200`, `coffee 47.5`, `cloud 1350000` →

```
Name   |        Amount | Share
-------+---------------+------
rent   |     $1,200.00 |  0.1%
coffee |        $47.50 |    0%
cloud  | $1,350,000.00 | 99.9%
Total  | $1,351,247.50 |  100%
compact=1.4M de=1.351.247,50 € toFixed=1351247.50
```

INPUT 1

```
rent 1200
coffee 47.5
cloud 1350000
```

OUTPUT 1

```
Name   |        Amount | Share
-------+---------------+------
rent   |     $1,200.00 |  0.1%
coffee |        $47.50 |    0%
cloud  | $1,350,000.00 | 99.9%
Total  | $1,351,247.50 |  100%
compact=1.4M de=1.351.247,50 €
toFixed=1351247.50
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const money = new Intl.NumberFormat("en-US", { style: "currency", currency: "USD" })
const percent = new Intl.NumberFormat("en-US", { style: "percent", maximumFractionDi
const compact = new Intl.NumberFormat("en-US", { notation: "compact" });
const rows = lines.map((l) => { const [name, amount] = l.trim().split(/\s+/); returr
const total = rows.reduce((s, r) => s + r.amount, 0);
const nameWidth = Math.max(4, ...rows.map((r) => r.name.length));
const amounts = rows.map((r) => money.format(r.amount));
const amountWidth = Math.max(...amounts.map((a) => a.length), money.format(total).le
console.log(`${"Name".padEnd(nameWidth)} | ${"Amount".padStart(amountWidth)} | Share
console.log(`${"-".repeat(nameWidth)}-+-${"-".repeat(amountWidth)}-+------`);
rows.forEach((r, i) => console.log(`${r.name.padEnd(nameWidth)} | ${amounts[i].padSt
console.log(`${"Total".padEnd(nameWidth)} | ${money.format(total).padStart(amountWid
const de = new Intl.NumberFormat("de-DE", { style: "currency", currency: "EUR" }).fo
console.log(`compact=${compact.format(total)} de=${de} toFixed=${total.toFixed(2)}`)
```

EXERCISE 2

html and sql tags

Write two tag functions. `html` joins the literal parts and HTML-escapes every interpolated value (`&` `< > " '`). `sql` replaces every interpolation with `$1` , `$2` , … and returns `{ text, values }` . Input: three lines — a name, a comment, an id. Print `html\` <p class="by">${name}</p> <blockquote>${comment}</blockquote>` , then the` sql `query` SELECT * FROM comments WHERE author = ${name} AND id = ${Number(id)} `as` <text> -- <JSON values> `, then a diagnostic tag applied to` `a\n${1}b${2}` `printing` strings=<JSON> raw=<JSON of strings.raw> values=<JSON> `, then` raw=<String.raw`C:\new\table`> interpolated=<length of `C:\new`>`.

Example: `Ada <script>alert(1)</script>` / `I "love" it & more` / `42` →

```
<p class="by">Ada &lt;script&gt;alert(1)&lt;/script&gt;</p><blockquote>I &quot;love&
SELECT * FROM comments WHERE author = $1 AND id = $2 -- ["Ada <script>alert(1)</scri
strings=["a\n","b",""] raw=["a\\n","b",""] values=[1,2]
raw=C:\new\table interpolated=5
```

INPUT 1

```
Ada <script>alert(1)</script>
I "love" it & more
42
```

OUTPUT 1

```
<p class="by">Ada
&lt;script&gt;alert(1)&lt;/script&gt;
</p><blockquote>I &quot;love&quot; it
&amp; more</blockquote>
SELECT * FROM comments WHERE author = $1
AND id = $2 -- ["Ada <script>alert(1)
</script>",42]
strings=["a\n","b",""] raw=
["a\\n","b",""] values=[1,2]
raw=C:\new\table interpolated=5
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const escapeHtml = (v) => String(v).replace(/[&<>"']/g, (ch) => ({ "&": "&amp;", "<"
function html(strings, ...values) {
  return strings.reduce((out, str, i) => out + str + (i < values.length ? escapeHtml
}
function sql(strings, ...values) {
  const text = strings.reduce((out, str, i) => out + str + (i < values.length ? `$${
  return { text, values };
}
const [name, comment, id] = lines.map((l) => l.trim());
console.log(html`<p class="by">${name}</p><blockquote>${comment}</blockquote>`);
const q = sql`SELECT * FROM comments WHERE author = ${name} AND id = ${Number(id)}`;
console.log(`${q.text} -- ${JSON.stringify(q.values)}`);
const parts = (strings, ...values) => `strings=${JSON.stringify(strings)} raw=${JSON
console.log(parts`a\n${1}b${2}`);
console.log(`raw=${String.raw`C:\new\table`} interpolated=${`C:\new`.length}`);
```

QUIZ

1. `"a,b,".split(",")` is…

A. `["a", "b"]`

B. `["a", "b", ""]` — a trailing separator yields an empty last element

C. `["a", "b", null]`

D. An error

2. A tag function receives…

A. The finished string

B. The array of literal parts (with a `.raw` view) and the interpolated values separately, before any joining

C. A regex

D. Only the values

3. `"x".replace("a", "b")` replaces…

A. Every `a`

B. Only the first occurrence — use `replaceAll` or a `g` regex for all

C. Nothing

D. Every character

4. Formatting `1234567.891` for a German reader should use…

A. `toFixed(3)` and manual dots

B. `new Intl.NumberFormat("de-DE").format(n)` → `1.234.567,891`

C. `toLocaleString()` with no locale

D. String replacement of `.` with `,`

5. `"abc".substring(2, 0)` is…

A. `""`

B. `"ab"` — `substring` swaps reversed arguments; `slice(2, 0)` would give `""`

C. `"c"`

D. An error

ANSWER KEY · THE STRING TOOLBOX, TEMPLATE LITERALS, TAGGED TEMPLATES AND INTL FORMATTING

1. B `"".split(",")` is `[""]`, not `[]`.

2. B So it can escape, parameterise or transform — `html`, `sql`, `css`.

3. B And a string replacement interprets `$` sequences — use a function when the replacement is data.

4. B Always pass a locale; create the formatter once and reuse it.

5. B Prefer `slice`; it also supports negative indices.

LESSON 3 · 14 MIN

Regular expressions — syntax, flags, groups and the five methods

A regular expression is a small program for matching text, and JavaScript's engine is a good one: full backtracking, lookaround, named groups, Unicode. The syntax is dense but finite; this lesson covers all of it that matters — literals and the constructor, the seven flags, character classes and quantifiers, greedy versus lazy, anchors and boundaries, capturing/non-capturing/named groups, alternation and backreferences — then the five methods you call (`test`, `exec`, `match`, `matchAll`, `replace`) and the one stateful trap (`lastIndex` with the `g` flag) that has cost everyone an afternoon.

Creating a regex

```
const re1 = /\d+/g;                          // literal — compiled once when t
const re2 = new RegExp("\\d+", "g");         // constructor — for patterns bui
const re3 = new RegExp(`^${escapeRegExp(userText)}$`, "i");   // user input MU
```

A regex object holds the pattern, the flags and, for `g`/`y`, a `lastIndex`. Literals inside a function are recreated per evaluation cheaply (engines cache compiled patterns); hoist them only for clarity.

Flags

Flag	Name	Effect
`g`	global	find all matches; makes `test` / `exec` stateful via `lastIndex`; required by `matchAll` and `replaceAll`
`i`	ignore case	case-insensitive (Unicode-aware with `u`)
`m`	multiline	`^` / `$` match at line starts/ends, not only string ends
`s`	dotAll	`.` matches newlines too
`u`	unicode	pattern works in code points: `.` matches an emoji, `\u{1F600}` and `\p{…}` work, case folding is correct — **use it on any Unicode text**
`y`	sticky	match only at exactly `lastIndex` — tokenizers
`d`	indices	`exec` results carry `.indices` with start/end of each group (Node 16+)

Atoms: what a position can match

```
.            any char except line terminators (all with s)
\d \w \s     digit [0-9]; word [A-Za-z0-9_]; whitespace     \D \W \S  their co
[abc] [a-z] [^0-9]     a set, a range, a negated set; inside [] most metachars
\p{L} \p{Lu} \p{Script=Greek} \P{N}    Unicode properties — letters, uppercase
\b \B        word boundary / non-boundary (between \w and \W); ASCII-based eve
^ $          start/end of input (of line with m)
\t \n \r é \u{1F600} \x41 \0      escapes; \. \* \( etc. escape metacharacters
```

`\w` and `\b` are ASCII: `\w` does not match `é`. For letters in any language use `\p{L}` with `u`.

Quantifiers

```
a*  a+  a?  a{3}  a{2,5}  a{2,}      zero+, one+, optional, exactly, range, at
a*?  a+?  a??  a{2,5}?               LAZY — take as little as possible, extend
```

Greedy `<.+>` on `<a><b>` matches the whole thing; lazy `<.+?>` matches `<a>`. Neither is "correct" — the negated class `<[^>]+>` is usually what you meant, and it never backtracks

across a `>`. Prefer a negated class or a specific class over `.*?` when you can name what must not appear.

Groups

```
(abc)          capturing: numbered left to right by opening parenthesis; $1, r
(?:abc)        non-capturing: grouping only — use for alternation and quantifi
(?<year>\d{4}) named: m.groups.year, $<year> in replacements, \k<year> to back
(a|b|cd)       alternation — tries left to right; order matters when alternati
\1  \k<name>   backreference: match the same text again — (\w)\1 finds doubled
```

A group inside a quantifier captures the **last** iteration only (`(\d,)+` on `1,2,3,` captures `3,`). Use `matchAll` for all of them.

The five methods (and two more)

```
const re = /(?<key>\w+)=(?<val>\d+)/g;
const text = "a=1 b=22 c=333";

re.test(text);                 // true — with g, advances lastIndex! see below
re.exec(text);                 // one match: ["a=1", "a", "1", index: 0, input
text.match(re);                // with g: ["a=1", "b=22", "c=333"] — full matc
[...text.matchAll(re)];        // every match WITH groups and indices; require
text.replace(re, "$<key>:$<val>");            // "a:1 b:22 c:333" — $1 $<name>
text.replace(re, (m, key, val, offset, whole, groups) ⇒ `${key}(${val.length}
text.replaceAll(re, …);                       // same with g required; string
text.split(/\s+/);  text.search(/\d/);        // split on a pattern; index of
```

Prefer `matchAll` for extraction, `replace` with a function for transformation, `test` for yes/no. `exec` in a `while` loop is the pre-2020 form of `matchAll`.

The `lastIndex` trap

A regex with `g` or `y` remembers where the last match ended in `lastIndex`, and `test` / `exec` start from there:

```
const re = /a/g;
re.test("a");    // true  — lastIndex is now 1
re.test("a");    // false — starts at 1, finds nothing, resets lastIndex to 0
```

Reusing one global regex for `test` across different strings gives alternating results — the classic "works every other time" bug. Fixes: do not put `g` on a regex you only `test`; or create the regex inside the function; or reset `re.lastIndex = 0` before each use. `match`, `matchAll`, `replace` and `split` handle `lastIndex` themselves.

Reading a pattern aloud

`^(?<user>[\w.+-]+)@(?<host>[\w-]+(?:\.[\w-]+)+)$` — "start; a name of word chars, dots, plus or minus; an at sign; a host of one or more dot-separated labels; end." Naming the groups and writing the regex on several lines (in a comment, or via `new RegExp` with a template literal and stripped whitespace) turns a wall of punctuation into something a reviewer can check.

Common mistakes

- Forgetting to escape `.` (`\d+.\d+` matches `1x2`), `+`, `?`, `(`, `[`, `\` (doubled in strings).
- `.*` greedy across the whole line when a negated class was meant; `.` not matching newlines without `s`.
- `\w` / `\b` on non-ASCII text; no `u` flag on Unicode text.
- A global regex reused with `test`; `match` with `g` when you needed groups (use `matchAll`).
- Capturing when you only needed grouping — `(?:…)` — and then miscounting `$n`.
- Alternation order: `/a|ab/` on `ab` matches just `a`.

Interview angle

- *"Greedy versus lazy?"* Greedy takes the maximum and backtracks; lazy takes the minimum and extends; a negated class is often better than either.
- *"What does the `g` flag change?"* All matches for `match` / `replace`; `matchAll` / `replaceAll` require it; `test` / `exec` become stateful through `lastIndex`.

"Named groups?" `(?<name>…)`, read as `m.groups.name`, replaced as `$<name>`, backreferenced as `\k<name>`.

"How do you get all matches with their groups?" `for (const m of str.matchAll(re))` (regex needs `g`).

"Why use the `u` flag?" Code-point semantics: `.` and classes match astral characters, `\p{…}` works, case-insensitivity is correct.

Key takeaways

Literal for fixed patterns, `new RegExp` (escaped) for dynamic ones; flags `g i m s u y d`; `u` on Unicode text.

Classes, `\p{L}`, anchors, greedy/lazy quantifiers — prefer negated classes to `.*?`.

Groups: capturing, `(?:…)`, `(?<name>…)`; alternation tries left to right; backreferences `\1` / `\k<name>`.

`test` (yes/no), `matchAll` (extract), `replace` with `$<name>` or a function (transform), `split`, `search`.

`g` / `y` make `test` / `exec` stateful via `lastIndex` — never reuse a global regex for `test` across strings.

EXERCISE 1

One regex, every method

Line 1 is a pattern, line 2 its flags (`-` for none), the rest are subjects. Build the regex and print `regex=<String(re)> global=<re.global> unicode=<re.unicode>`. For each subject, using a **fresh** regex object each time, print `<JSON subject>: test=<bool> match=<JSON of the match array or null>` plus `groups=<JSON>` when the match has named groups. Finally call `test` on the **same** regex object three times against the first subject and print `sameRegexThrice=<results> lastIndex=<lastIndex after each>`.

Example: `(?<user>\w+)@(?<host>[\w.]+)` with flags `g`, subjects `ada@example.com and bob@x.io` and `nobody here` →

```
regex=/(?<user>\w+)@(?<host>[\w.]+)/g global=true unicode=false
"ada@example.com and bob@x.io": test=true match=["ada@example.com","bob@x.io"]
"nobody here": test=false match=null
sameRegexThrice=true,true,false lastIndex=15,28,0
```

With `g`, `match` returns whole matches only (no groups), and `test` on a reused regex walks through the string and then fails.

INPUT 1

```
(?<user>\w+)@(?<host>[\w.]+)
g
ada@example.com and bob@x.io
nobody here
```

OUTPUT 1

```
regex=/(?<user>\w+)@(?<host>[\w.]+)/g
global=true unicode=false
"ada@example.com and bob@x.io":
test=true match=
["ada@example.com","bob@x.io"]
"nobody here": test=false match=null
sameRegexThrice=true,true,false
lastIndex=15,28,0
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const [pattern, flagsLine, ...subjects] = input.split("\n").filter((l) ⇒ l ≠= "");
const flags = flagsLine.trim() ≡= "-" ? "" : flagsLine.trim();
const re = new RegExp(pattern, flags);
console.log(`regex=${re} global=${re.global} unicode=${re.unicode}`);
for (const subject of subjects) {
  const fresh = new RegExp(pattern, flags);        // a fresh object so lastIndex ne
  const m = subject.match(fresh);
  console.log(`${JSON.stringify(subject)}: test=${new RegExp(pattern, flags).test(su
}
if (subjects.length) {
  const first = subjects[0];
  const results = [], indices = [];
  for (let k = 0; k < 3; k++) { results.push(re.test(first)); indices.push(re.lastIr
  console.log(`sameRegexThrice=${results.join(",")} lastIndex=${indices.join(",")}`)
}
```

EXERCISE 2

Dates with named groups and matchAll

With `DATE = /(?<y>\d{4})-(?<m>\d{2})-(?<d>\d{2})/g`, print: `dates=<count>: <match@index …>` from `matchAll`; `british=<text with every date rewritten as $<d>/$<m>/$<y>>`; `years=<comma list> latest=<max date string or none>`; `numbers=<JSON of every \d+ run as numbers>`; `parts=<JSON of text.split(/\s*[,;]\s*/)>`; and `matchWithG=<JSON of text.match(DATE)> firstExec=<JSON of the groups from exec with a non-global copy, or null>`.

Example: `release 2024-03-15, patch 2024-04-02; eol 2025-01-31 (build 77)` →

```
dates=3: 2024-03-15@8 2024-04-02@26 2025-01-31@42
british=release 15/03/2024, patch 02/04/2024; eol 31/01/2025 (build 77)
years=2024,2024,2025 latest=2025-01-31
numbers=[2024,3,15,2024,4,2,2025,1,31,77]
parts=["release 2024-03-15","patch 2024-04-02","eol 2025-01-31 (build 77)"]
matchWithG=["2024-03-15","2024-04-02","2025-01-31"] firstExec={"y":"2024","m":"03","
```

INPUT 1

```
release 2024-03-15, patch 2024-04-02;
eol 2025-01-31 (build 77)
```

OUTPUT 1

```
dates=3: 2024-03-15@8 2024-04-02@26
2025-01-31@42
british=release 15/03/2024, patch
02/04/2024; eol 31/01/2025 (build 77)
years=2024,2024,2025 latest=2025-01-31
numbers=
[2024,3,15,2024,4,2,2025,1,31,77]
parts=["release 2024-03-15","patch 2024-
04-02","eol 2025-01-31 (build 77)"]
matchWithG=["2024-03-15","2024-04-
02","2025-01-31"] firstExec=
{"y":"2024","m":"03","d":"15"}
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const text = input.trim();
const DATE = /(?<y>\d{4})-(?<m>\d{2})-(?<d>\d{2})/g;
const found = [...text.matchAll(DATE)];
console.log(`dates=${found.length}: ${found.map((m) => `${m[0]}@${m.index}`).join("
console.log(`british=${text.replace(DATE, "$<d>/$<m>/$<y>")}`);
console.log(`years=${found.map((m) => m.groups.y).join(",")} latest=${found.map((m)
console.log(`numbers=${JSON.stringify((text.match(/\d+/g) ?? []).map(Number))}`);
console.log(`parts=${JSON.stringify(text.split(/\s*[,;]\s*/))}`);
console.log(`matchWithG=${JSON.stringify(text.match(DATE))} firstExec=${JSON.stringi
```

QUIZ

1. `/a/g.test("a")` twice on the same regex object gives…

A. `true, true`

B. `true, false` — the `g` flag makes `test` resume from `lastIndex`

C. `false, true`

D. An error

2. `"<a><b>".match(/<.+>/)[0]` is…

A. `"<a>"`

B. `"<a><b>"` — greedy `.+` takes as much as possible; `<[^>]+>` (or lazy `.+?`) gives `"<a>"`

C. `"<b>"`

D. `null`

3. `str.match(/(\d+)-(\d+)/g)` returns…

A. Every match with its groups

B. Only the whole matches — with `g`, `match` drops groups; use `matchAll` for groups

C. The first match with groups

D. `null` always

4. `\w` and `\b` in a regex with the `u` flag match…

A. Letters in every language

B. ASCII word characters only — use `\p{L}` (with `u`) for international letters

C. Emoji

D. Whitespace

5. `(?<year>\d{4})` can be referenced in a replacement as…

A. `\year`

B. `$<year>` in a replacement string, `m.groups.year` in a match, `\k<year>` inside the pattern

C. `%year`

D. `{year}`

ANSWER KEY · REGULAR EXPRESSIONS — SYNTAX, FLAGS, GROUPS AND THE FIVE METHODS

1. B Never reuse a global regex for `test`; use a fresh one or reset `lastIndex`.

2. B A negated class is usually better than a lazy quantifier.

3. B `matchAll` requires the `g` flag and yields full match objects.

4. B `u` fixes code-point handling and `\p{…}`, not the ASCII classes.

5. B Named groups make patterns readable and replacements robust to reordering.

LESSON 4 · 13 MIN

Regex in practice — lookarounds, Unicode properties, escaping input, ReDoS, and knowing when to stop

The syntax lesson makes regexes *possible*; this one makes them *safe and useful*. Lookarounds let you assert context without consuming it (thousands separators, password rules, "a word not followed by…"); Unicode property escapes replace `[A-Za-z]` with something that works for every language; a function replacer turns `replace` into a small compiler. Then the two hazards every production regex must be checked for: user input pasted into a pattern (escape it), and **catastrophic backtracking** — patterns that take seconds or hours on a crafted input and can knock a server over. The lesson ends with the honest list of things a regex should not be asked to do.

Lookarounds

```
(?=…)   positive lookahead:  x(?=y) matches x only if y follows      — not cor
(?!…)   negative lookahead:  x(?!y) matches x only if y does NOT follow
(?⩽…)  positive lookbehind: (?⩽y)x matches x only if y precedes    — ES2018,
(?<!…)  negative lookbehind
```

```
"1234567".replace(/\B(?=(\d{3})+(?!\d))/g, ",");          // "1,234,567" — ir
/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/.test(pw);         // password: at lea
"price: $42, cost: $7".match(/(?⩽\$)\d+/g);               // ["42", "7"] — di
"foo.js foo.test.js".match(/\b\w+(?<!\.test)\.js\b/g);    // files ending .js
```

Lookarounds are zero-width: they check and do not advance. Multiple lookaheads at the start of a pattern express "all of these must hold somewhere" — the standard shape for validation rules. Lookbehind must have a bounded length in some engines; V8 allows variable length.

Unicode property escapes (with `u`)

```
/\p{L}+/gu                    // letters in any script — "café", "日本語", "Eλ
/\p{Lu}/u  /\p{Ll}/u          // uppercase / lowercase letters
/\p{N}/u   /\p{Nd}/u          // any numeric / decimal digits (incl. non-ASCI
/\p{Script=Han}/u             // a specific script
/\p{Emoji_Presentation}/u     // emoji shown as pictures (approximately — emo
/[\p{L}\p{M}]+/u              // letters plus combining marks — a "word" that
```

`\w`, `\d`, `\b` stay ASCII even with `u`. For "a word" in user text, `[\p{L}\p{M}\p{N}_]+` with `u` is the honest replacement; for splitting into words in any language, `Intl.Segmenter` (lesson 1) is better still.

`replace` with a function — a small compiler

```
const vars = { name: "Ada", n: 3 };
"Hi {{name}}, you have {{n}} items {{missing}}".replace(/\{\{(\w+)\}\}/g, (_,
// "Hi Ada, you have 3 items {{missing}}"

text.replace(/\b(\w)(\w*)\b/g, (_, first, rest) ⇒ first.toUpperCase() + rest)
csv.replace(/"((?:[^"]|"")*)"/g, (_, inner) ⇒ inner.replace(/""/g, '"'));
```

The callback receives `(match, ...groups, offset, input, namedGroups)` and returns the replacement text; `$` sequences are **not** interpreted in its return value, which is exactly why it is the safe form for replacement text that comes from data.

Escaping user input

```
const escapeRegExp = (s) ⇒ s.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
const re = new RegExp(`\\b${escapeRegExp(query)}\\b`, "iu");
```

Any string from outside placed into `new RegExp` must be escaped, or `.` matches anything, `(` throws a `SyntaxError`, and `(a+)+$` (below) takes the server down. `RegExp.escape` is a 2025 addition; on Node 16 the one-liner above is the standard.

Catastrophic backtracking (ReDoS)

```
/^(\w+\s?)*$/.test("a ".repeat(30) + "!");     // seconds; each extra characte
/(a+)+b/.test("a".repeat(40));                  // same disease
```

When a pattern has **nested quantifiers**, or **adjacent quantified parts that can match the same text** (`\w+\s?` repeated: is `"ab"` one `\w+` or two?), a failing match makes the engine try every way to split the input — exponential time. A single request with a 40-character string can pin a CPU for minutes: a denial-of-service with one HTTP call. Rules:

- No nested quantifiers over overlapping parts: `(\w+\s?)*` → `\w+(\s\w+)*` or `(\w+\s)*\w+` — make each repetition **consume something unambiguous**.
- Prefer negated classes to `.*`: `"[^"]*"` cannot backtrack across a quote.
- Anchor patterns (`^…$`) so failure is detected early rather than retried at every start position.
- Bound repetition where the format allows (`\d{1,5}` not `\d+`).
- Test suspicious patterns against long adversarial strings; tools (`safe-regex`, `recheck`) flag the classic shapes.
- Keep regexes that run on **untrusted input** simple; validate length first (`if (s.length > 200) reject`).

Common patterns, pragmatically

Need	Pattern	Note
Integer	`^-?\d+$`	`Number.isInteger(Number(s))` is often clearer
Decimal	`^-?\d+(\.\d+)?$`	
Hex colour	`^#(?:[0-9a-f]{3}){1,2}$` with `i`	
ISO date	`^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])$`	validates shape, not calendar — `Date` for that
IPv4	`^(?:(?:25[0-5]|2[0-4]\d|1?\d?\d)\.){3}(?:25[0-5]|2[0-4]\d|1?\d?\d)$`	
Email	`^[^\s@]+@[^\s@]+\.[^\s@]+$`	deliberately loose — the RFC grammar is not worth encoding; send a confirmation mail
Slug	`^[a-z0-9]+(?:-[a-z0-9]+)*$`	
Trim internal spaces	`replace(/\s+/g, " ")`	

Validation regexes answer "does this look right"; a parser or the real API (`new URL(s)`, `Date`) answers "is it right".

When a regex is the wrong tool

HTML/**XML**: nested, contextual — use a parser (browsers have `DOMParser`; Node has `cheerio` / `parse5`). `/<b>(.*?)<\/b>/` breaks on attributes, nesting, whitespace and comments.

JSON: `JSON.parse`. **URLs**: `new URL`. **Dates**: a date library or `Date` with a fixed format.

Programming languages / **templates with nesting**: a tokenizer plus a recursive-descent parser (next lesson).

Anything you cannot explain in one sentence: split it into two passes or write the loop.

The sticky flag makes regexes excellent **tokenizers** — one anchored alternation of token patterns advanced by `lastIndex` — and that is the right division of labour: regex for the tokens, code for the structure.

Common mistakes

Unescaped user input in `new RegExp`.

`(\w+\s?)*`-shaped patterns on request data.

`\w` for "letters" in international text; `[A-Za-z]` in a product used outside one language.

A 300-character email regex that rejects valid addresses and accepts invalid ones.

Parsing HTML with regex; extracting balanced parentheses with regex (impossible in general).

`$1` in a replacement string when the replacement comes from data (use a function).

Interview angle

"What is a lookahead?" A zero-width assertion that the following text matches (or, negative, does not) without consuming it.

"How do you insert thousands separators with a regex?" `\B(?=(\d{3})+(?!\d))` → `,`.

"What is catastrophic backtracking?" Exponential matching time from nested or overlapping quantifiers on a failing input — a DoS vector; fix by making repetitions unambiguous.

"How do you safely build a regex from user input?" Escape metacharacters first.

"Should you parse HTML with regex?" No — nesting and context need a parser; regex for tokens only.

Key takeaways

Lookarounds assert context without consuming: `(?=)` `(?!)` `(?<=)` `(?<!)`; stack lookaheads for multi-rule validation.

`\p{L}` / `\p{N}` with `u` for international text; `\w` / `\b` stay ASCII.

`replace` with a function is a mini compiler and the safe form for data-driven replacements; escape input before `new RegExp`.

ReDoS: no nested/overlapping quantifiers, negated classes over `.*`, anchors, bounded repeats, length checks on untrusted input.

Regex for tokens and shapes; parsers for structure (HTML, JSON, URLs, languages).

EXERCISE 1

Lookarounds, Unicode properties and a validation table

Line 1: numbers — print `grouped=<each with thousands separators via \B(?=(\d{3})+(?!\d))>`. Line 2: text — print `letters=<\p{L} count> asciiWordChars=<\w count> digits=<\p{Nd} count> words=<runs of [\p{L}\p{M}]+>` (all with the `u` flag). Line 3: candidate passwords — print `passwords=<pw:bool …>` for `/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/`. Remaining lines `<kind> <value>` validate against: `hex` `^#(?:[0-9a-f]{3}){1,2}$` (case-insensitive), `ipv4` (each octet 0–255), `date` (`YYYY-MM-DD` shape with month 01–12 and day 01–31), `slug` `^[a-z0-9]+(?:-[a-z0-9]+)*$`; print `<kind> <value>: <bool>`.

Example (excerpt):

```
grouped=1,234,567 42 1,000 999,999
letters=11 asciiWordChars=9 digits=2 words=3
passwords=abc12345:false Passw0rdX:true short1A:false
hex #12345g: false
ipv4 256.1.1.1: false
date 2024-02-30: true
```

The date regex validates shape, not the calendar — `2024-02-30` passes; a `Date` would reject it.

INPUT 1

```
1234567 42 1000 999999
café 日本 42 naïve
abc12345 Passw0rdX short1A
hex #fff
hex #12345g
ipv4 192.168.1.255
ipv4 256.1.1.1
date 2024-02-30
date 2024-13-01
slug hello-world-2
slug Hello_World
```

OUTPUT 1

```
grouped=1,234,567 42 1,000 999,999
letters=11 asciiWordChars=9 digits=2
words=3
passwords=abc12345:false Passw0rdX:true
short1A:false
hex #fff: true
hex #12345g: false
ipv4 192.168.1.255: true
ipv4 256.1.1.1: false
date 2024-02-30: true
date 2024-13-01: false
slug hello-world-2: true
slug Hello_World: false
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const [numbersLine, textLine, passwordLine, ...checks] = lines;
console.log(`grouped=${numbersLine.trim().split(/\s+/).map((n) => n.replace(/\B(?=(\
const letters = (textLine.match(/\p{L}/gu) ?? []).length, asciiWord = (textLine.matc
console.log(`letters=${letters} asciiWordChars=${asciiWord} digits=${digits} words=$
const strong = /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/;
console.log(`passwords=${passwordLine.trim().split(/\s+/).map((p) => `${p}:${strong.
const validators = {
  hex: /^#(?:[0-9a-f]{3}){1,2}$/i,
  ipv4: /^(?:(?:25[0-5]|2[0-4]\d|1?\d?\d)\.){3}(?:25[0-5]|2[0-4]\d|1?\d?\d)$/,
  date: /^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])$/,
  slug: /^[a-z0-9]+(?:-[a-z0-9]+)*$/,
};
for (const line of checks) {
  const [kind, value] = line.trim().split(/\s+/);
  console.log(`${kind} ${value}: ${validators[kind].test(value)}`);
}
```

EXERCISE 2

Escape the query, flag the dangerous pattern

Line 1 is a user's search query; the following lines are texts to search, except lines starting with `pattern` which are regex sources to inspect. Escape the query with `escapeRegExp` (in the starter), build a case-insensitive global regex, and print each text with matches wrapped in `[…]` followed by `(<count>)`. Then try the **unescaped** query as a regex and print `unescaped: matches <total across texts> (<source>)` or `unescaped: <error constructor name>` if it does not compile. Finally, for each `pattern`, print `pattern <p>: suspicious` when a quantified group contains a quantifier or two open-ended quantified pieces (`\w+`, `.*`, `\s+`…) sit next to each other, else `ok`.

Example: query `a.c`, texts `abc a.c aXc` and `A.C matches?`, patterns `(\w+\s?)*$`, `^[a-z]+$`, `(a+)+b`, `\d{1,5}-\d{1,5}` →

```
abc [a.c] aXc (1)
[A.C] matches? (1)
unescaped: matches 5 (a.c)
pattern (\w+\s?)*$: suspicious
pattern ^[a-z]+$: ok
pattern (a+)+b: suspicious
pattern \d{1,5}-\d{1,5}: ok
```

Unescaped, the dot matched `abc`, `aXc` and more — five hits instead of two.

INPUT 1

```
a.c
abc a.c aXc
A.C matches?
pattern (\w+\s?)*$
pattern ^[a-z]+$
pattern (a+)+b
pattern \d{1,5}-\d{1,5}
```

OUTPUT 1

```
abc [a.c] aXc (1)
[A.C] matches? (1)
unescaped: matches 5 (a.c)
pattern (\w+\s?)*$: suspicious
pattern ^[a-z]+$: ok
pattern (a+)+b: suspicious
pattern \d{1,5}-\d{1,5}: ok
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const escapeRegExp = (s) => s.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");
const [query, ...rest] = lines;
const texts = rest.filter((l) => !l.startsWith("pattern "));
const patterns = rest.filter((l) => l.startsWith("pattern ")).map((l) => l.slice(8).
const safe = new RegExp(escapeRegExp(query.trim()), "giu");
for (const t of texts) {
  const hits = t.match(safe) ?? [];
  console.log(`${t.replace(safe, (m) => `[${m}]`)} (${hits.length})`);
}
let raw;
try { const re = new RegExp(query.trim(), "giu"); raw = `matches ${texts.reduce((n,
catch (err) { raw = `${err.constructor.name}`; }
console.log(`unescaped: ${raw}`);
// A heuristic, not a proof: a quantified group whose body is itself quantified, or
const suspicious = (p) => /\([^()]*[+*][^()]*\)[+*{]/.test(p) || /[+*]\)?\s*\(?[^()\
for (const p of patterns) console.log(`pattern ${p}: ${suspicious(p) ? "suspicious"
```

QUIZ

1. `x(?=y)` matches…

A. `xy`

B. `x` only when followed by `y`, without consuming `y` — a zero-width lookahead

C. `y`

D. `x` or `y`

2. `/^(\w+\s?)*$/` on a long non-matching string…

A. Fails instantly

B. Can take exponential time — overlapping quantifiers make the engine try every split (catastrophic backtracking / ReDoS)

C. Matches anyway

D. Throws

3. Before `new RegExp(userInput)` you must…

A. Lower-case it

B. Escape regex metacharacters — otherwise `.` matches anything and `(` throws

C. Trim it

D. Nothing

4. A `replace` callback's return value…

A. Interprets `$1`

B. Is inserted literally — `$` sequences are **not** interpreted, which makes it the safe form for data-driven replacements

C. Must be a regex

D. Is ignored

5. Parsing HTML with a regex…

A. Is fine for small pages

B. Breaks on nesting, attributes, comments and whitespace — use a parser; regex is for tokens

C. Is faster and correct

D. Is required by browsers

ANSWER KEY · REGEX IN PRACTICE — LOOKAROUNDS, UNICODE PROPERTIES, ESCAPING INPUT, REDOS, AND KNOWING WHEN TO STOP

1. B Stacked lookaheads at the start express "all of these must hold".

2. B Make each repetition consume something unambiguous: `\w+(\s\w+)*`.

3. B `s.replace(/[.*+?^${}()|[\]\\]/g, "\\$&")`.

4. B The callback receives `(match, ...groups, offset, input, namedGroups)`.

5. B Same for JSON (`JSON.parse`), URLs (`new URL`) and balanced parentheses.

LESSON 5 · 14 MIN

Parsing text by hand — tokenizers, recursive descent, CSV and error positions

Regexes stop where structure starts: nesting, quoting rules, grammars. For those you write a **parser**, and the good news is that the two techniques covering almost every case fit in a page each. A **tokenizer** turns characters into a list of typed tokens using one sticky regex; a **recursive-descent parser** turns tokens into a tree (or a value) with one function per grammar rule. Add position tracking and you get error messages that point at the problem. This lesson builds both for arithmetic expressions, walks through the quoting rules of CSV — the format everyone thinks is trivial and nobody parses right the first time — and ends with a small key/value grammar and the design habits that make hand-written parsers maintainable.

Tokenizing with a sticky regex

```
const TOKEN = /\s*(?:(?<num>\d+(?:\.\d+)?)|(?<id>[A-Za-z_]\w*)|(?<op>[-+*/()=,
function tokenize(src) {
  const tokens = [];
  TOKEN.lastIndex = 0;
  while (TOKEN.lastIndex < src.length) {
    const start = TOKEN.lastIndex;
    const m = TOKEN.exec(src);
    if (!m) {
      if (/^\s*$/.test(src.slice(start))) break;                    // only tra
      throw new SyntaxError(`unexpected character '${src[start + (src.slice(st
    }
    const type = Object.keys(m.groups).find((k) => m.groups[k] !== undefined);
    tokens.push({ type, value: m.groups[type], pos: m.index + m[0].length - m.
  }
  tokens.push({ type: "eof", value: "", pos: src.length });
  return tokens;
}
```

The `y` flag makes `exec` match **only at `lastIndex`** and advance it — the pattern is an alternation of every token type, longest/most-specific alternatives first, with leading whitespace absorbed. A failed match at a position means an illegal character, and you know exactly where. Named groups tell you which alternative won. The `eof` sentinel saves every consumer a bounds check.

Recursive descent

Write the grammar with precedence built in — lower precedence at the top, one function per rule, each returning a value (or an AST node):

```
expr   := term (("+" | "-") term)*
term   := factor (("*" | "/") factor)*
factor := "-" factor | number | "(" expr ")"
```

```
function parse(tokens) {
  let i = 0;
  const peek = () => tokens[i];
  const next = () => tokens[i++];
  const expect = (value) => {
    const t = next();
    if (t.value !== value) throw new SyntaxError(`expected '${value}' but fou
    return t;
  };
  function expr() {
    let left = term();
    while (peek().value === "+" || peek().value === "-") { const op = next().
    return left;
  }
  function term() {
    let left = factor();
    while (peek().value === "*" || peek().value === "/") {
      const op = next(); const right = factor();
      if (op.value === "/" && right === 0) throw new RangeError(`division by z
      left = op.value === "*" ? left * right : left / right;
    }
    return left;
  }
  function factor() {
    const t = peek();
    if (t.value === "-") { next(); return -factor(); }
    if (t.type === "num") { next(); return Number(t.value); }
    if (t.value === "(") { next(); const v = expr(); expect(")"); return v; }
    throw new SyntaxError(`unexpected '${t.value || "end of input"}' at ${t.po
  }
  const value = expr();
  if (peek().type !== "eof") throw new SyntaxError(`unexpected '${peek().value
  return value;
}
```

Each function consumes exactly the tokens of its rule and returns. Precedence falls out of the nesting: `term` binds tighter than `expr` because `expr` calls `term` for its operands. Left-associativity comes from the `while` loop accumulating into `left`; right-associativity (exponent) would recurse instead. Adding a rule — comparison operators, function calls, variables — is adding one function and one line in the caller. This is how real language front-ends are written; the technique scales from a calculator to a JSON parser to a programming language.

CSV — the trivial format that is not

RFC 4180: fields separated by commas; a field may be quoted with `"`; inside quotes, commas and newlines are literal and `"` is written `""`. A `split(",")` parser fails on the second rule. A character-by-character state machine handles all of them:

```
function parseCsvLine(line) {
  const fields = [];
  let field = "", inQuotes = false;
  for (let i = 0; i < line.length; i++) {
    const ch = line[i];
    if (inQuotes) {
      if (ch === '"') {
        if (line[i + 1] === '"') { field += '"'; i++; }      // escaped quote
        else inQuotes = false;                               // closing quote
      } else field += ch;
    } else if (ch === '"') inQuotes = true;
    else if (ch === ",") { fields.push(field); field = ""; }
    else field += ch;
  }
  if (inQuotes) throw new SyntaxError("unterminated quoted field");
  fields.push(field);
  return fields;
}
```

Fields spanning lines need the state machine to run over the whole text rather than line by line — one more reason real projects use a library (`csv-parse`, `papaparse`) and the hand version is for understanding. The same state-machine shape parses INI files, `.env` files, query strings and log formats.

Errors that point somewhere

Every `throw` above carries a **position**. Turn it into line and column for multi-line input (`text.slice(0, pos).split("\n")` → line count and last segment length) and print the offending line with a caret underneath; that is what compilers do and what saves a user twenty minutes. Fail on the **first** error for a language (the rest is noise after a missing bracket); collect errors for a data file where each record is independent (module 7's rule).

Design habits

Separate lexing from parsing. The parser reads tokens, never characters; whitespace and comments vanish in the tokenizer.

One function per rule, named after the rule, with the grammar written in a comment above the parser.

`peek` / `next` / `expect` as the only ways to touch the token stream.

Return values or AST nodes, not both; an AST (`{ type: "binary", op, left, right }`) lets you evaluate, pretty-print and analyse with separate walkers.

Sentinel `eof`; positions on every token and every error.

Know when to stop: a real grammar with many precedence levels wants a Pratt parser or a generator (PEG.js, Chevrotain, ANTLR), and a real data format wants its library.

Common mistakes

Tokenizing with `split` and losing positions and quoted content.

Precedence by hand-ordering `if`s instead of by grammar structure; associativity backwards.

Forgetting the `eof` check, so `1 2` parses as `1` with garbage ignored.

CSV by `split(",")`; not handling `""` or unterminated quotes.

Errors without positions ("invalid input").

Reaching for a hand parser when `JSON.parse`, `new URL` or a library exists.

Interview angle

"How would you evaluate `2 + 3 * (4 - 1)` from a string?" Tokenize with a sticky regex, then recursive descent with `expr` / `term` / `factor` so precedence is structural.

"Why can't `split(',')` parse CSV?" Quoted fields contain commas and doubled quotes; you need a state machine.

"What does the sticky flag do?" Matches only at `lastIndex` and advances it — the tokenizer primitive.

"How do you produce good parse errors?" Carry positions on tokens; report the unexpected token and where; convert to line/column.

"When do you stop writing parsers by hand?" Many precedence levels or a standard format: use a Pratt parser, a generator, or the library.

Key takeaways

Tokenizer: one sticky alternation with named groups, positions on tokens, an `eof` sentinel, an error for illegal characters.

Recursive descent: one function per grammar rule; precedence from nesting, associativity from loops versus recursion; `peek` / `next` / `expect`.

CSV and similar formats are state machines over characters with quoting rules — not `split`.

Every error carries a position; first-error for languages, all-errors for independent records.

Regex for tokens, code for structure, libraries for standard formats.

EXERCISE 1

A CSV line parser

Implement `parseCsvLine(line)` following RFC 4180: fields separated by commas; a field may be wrapped in double quotes, inside which commas are literal and a quote is written as two quotes; an unterminated quoted field throws `SyntaxError("unterminated quoted field")`. Print each line's fields as JSON, or `error: <message>`.

Example:

```
ada,36,London
"Smith, John",41,"He said ""hi"""
,,
"unterminated,1
plain
```

→

```
["ada","36","London"]
["Smith, John","41","He said \"hi\""]
["","",""]
error: unterminated quoted field
["plain"]
```

INPUT 1

```
ada,36,London
"Smith, John",41,"He said ""hi"""
,,
"unterminated,1
plain
```

OUTPUT 1

```
["ada","36","London"]
["Smith, John","41","He said \"hi\""]
["","",""]
error: unterminated quoted field
["plain"]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
function parseCsvLine(line) {
  const fields = [];
  let field = "", inQuotes = false;
  for (let i = 0; i < line.length; i++) {
    const ch = line[i];
    if (inQuotes) {
      if (ch === '"') {
        if (line[i + 1] === '"') { field += '"'; i++; }   // an escaped quote
        else inQuotes = false;
      } else field += ch;
    } else if (ch === '"') inQuotes = true;
    else if (ch === ",") { fields.push(field); field = ""; }
    else field += ch;
  }
  if (inQuotes) throw new SyntaxError("unterminated quoted field");
  fields.push(field);
  return fields;
}
for (const line of lines) {
  try { console.log(JSON.stringify(parseCsvLine(line.replace(/\r$/, "")))); }
  catch (err) { console.log(`error: ${err.message}`); }
}
```

EXERCISE 2

Tokenize and evaluate arithmetic

Write `tokenize(src)` with the starter's sticky regex (numbers and the operators `+ - * / ( )`), recording each token's position and appending an `eof` token; an unexpected character throws `SyntaxError("unexpected character '<c>' at <pos>")`. Then a recursive-descent `evaluate` over `expr := term (('+'|'-') term)*`, `term := factor (('*'|'/') factor)*`, `factor := '-' factor | number | '(' expr ')'`, throwing `RangeError("division by zero at <pos of />")`, `SyntaxError("expected ')' but found <'tok' or end of input> at <pos>")` and `SyntaxError("unexpected <'tok' or end of input> at <pos>")` (also for leftover tokens). Print `<expression> = <value>` or `<Name>: <message>` per line.

Example:

```
2 + 3 * (4 - 1)
-(2 + 3) * 2
1 / 0
2 +
(1 + 2
3 $ 4
1 2
```

→

```
2 + 3 * (4 - 1) = 11
-(2 + 3) * 2 = -10
RangeError: division by zero at 2
SyntaxError: unexpected end of input at 3
SyntaxError: expected ')' but found end of input at 6
SyntaxError: unexpected character '$' at 2
SyntaxError: unexpected '2' at 2
```

INPUT 1

```
2 + 3 * (4 - 1)
-(2 + 3) * 2
10 / 4
1 / 0
2 +
(1 + 2
3 $ 4
1 2
7
```

OUTPUT 1

```
2 + 3 * (4 - 1) = 11
-(2 + 3) * 2 = -10
10 / 4 = 2.5
RangeError: division by zero at 2
SyntaxError: unexpected end of input at
3
SyntaxError: expected ')' but found end
of input at 6
SyntaxError: unexpected character '$' at
2
SyntaxError: unexpected '2' at 2
7 = 7
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const TOKEN = /\s*(?:(?<num>\d+(?:\.\d+)?)|(?<op>[-+*/()]))/y;
function tokenize(src) {
  const tokens = [];
  TOKEN.lastIndex = 0;
  while (TOKEN.lastIndex < src.length) {
    const start = TOKEN.lastIndex;
    const m = TOKEN.exec(src);
    if (!m) {
      const rest = src.slice(start);
      if (/^\s*$/.test(rest)) break;
      const at = start + rest.match(/^\s*/)[0].length;
      throw new SyntaxError(`unexpected character '${src[at]}' at ${at}`);
    }
    const type = m.groups.num !== undefined ? "num" : "op";
    tokens.push({ type, value: m.groups[type], pos: m.index + m[0].length - m.groups
  }
  tokens.push({ type: "eof", value: "", pos: src.length });
  return tokens;
}
function evaluate(src) {
  const tokens = tokenize(src);
  let i = 0;
  const peek = () => tokens[i];
  const next = () => tokens[i++];
  const describe = (t) => (t.type === "eof" ? "end of input" : `'${t.value}'`);
  function expr() {
    let left = term();
    while (peek().value === "+" || peek().value === "-") { const op = next().value;
    return left;
  }
  function term() {
    let left = factor();
    while (peek().value === "*" || peek().value === "/") {
      const op = next(); const right = factor();
      if (op.value === "/" && right === 0) throw new RangeError(`division by zero at
      left = op.value === "*" ? left * right : left / right;
    }
    return left;
  }
  function factor() {
    const t = peek();
    if (t.value === "-") { next(); return -factor(); }
    if (t.type === "num") { next(); return Number(t.value); }
    if (t.value === "(") {
      next();
      const v = expr();
      if (peek().value !== ")") throw new SyntaxError(`expected ')' but found ${desc
```

```
      next();
      return v;
    }
    throw new SyntaxError(`unexpected ${describe(t)} at ${t.pos}`);
  }
  const value = expr();
  if (peek().type !== "eof") throw new SyntaxError(`unexpected ${describe(peek())} a
  return value;
}
for (const line of lines) {
  try { console.log(`${line.trim()} = ${evaluate(line)}`); }
  catch (err) { console.log(`${err.name}: ${err.message}`); }
}
```

QUIZ

1. The sticky flag `y` makes `exec` …

A. Match anywhere after `lastIndex`

B. Match only at exactly `lastIndex` and advance it — the tokenizer primitive

C. Match all occurrences

D. Ignore case

2. In a recursive-descent parser, `*` binds tighter than `+` because…

A. Of an operator table

B. `expr` calls `term` for its operands and `term` calls `factor` — precedence is the nesting of the grammar functions

C. Of left-to-right scanning

D. It is hard-coded in `Math`

3. `split(",")` cannot parse CSV because…

A. It is too slow

B. Quoted fields may contain commas and doubled quotes — you need a state machine over characters

C. CSV uses semicolons

D. It drops empty fields

4. Why append an `eof` token?

A. Performance

B. So the parser can check nothing is left over and report `unexpected '2'` instead of silently ignoring trailing input

C. It is required by regex

D. To mark line breaks

5. Good parse errors…

A. Say "invalid input"

B. Name the unexpected token and its position — converted to line/column for multi-line input

C. List the grammar

D. Are thrown only at the end

ANSWER KEY · PARSING TEXT BY HAND — TOKENIZERS, RECURSIVE DESCENT, CSV AND ERROR POSITIONS

1. B A failed match at a position means an illegal character, and you know where.

2. B Left-associativity comes from the `while` loop; right-associativity from recursing on the right.

3. B Multi-line quoted fields need the machine to run over the whole text, not per line.

4. B `1 2` should be an error, not `1`.

5. B First error for a language; all errors for independent records.

CHECKPOINT · 25 MIN

Checkpoint — Strings, Unicode, regular expressions and parsing

This checkpoint covers UTF-16 code units versus code points versus grapheme clusters, normalisation, case mapping and locale-aware comparison; the string toolbox, template literals, tagged templates and `Intl` formatting; regular-expression syntax, flags, groups, the five methods and `lastIndex`; lookarounds, Unicode property escapes, escaping input and catastrophic backtracking; and hand-written parsing with sticky tokenizers, recursive descent and CSV quoting.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- Why `"😀".length` is 2; how to count code points and graphemes; what `normalize("NFC")` fixes; how `toUpperCase` can change length.
- `slice` versus `substring`; what `"".split(",")` returns; what a tag function receives; why `Intl` formatters are created once.
- Which flag makes `test` stateful; greedy versus lazy versus a negated class; how to get every match with groups.
- What a lookahead is; why `(\w+\s?)*` is dangerous; how to escape user input for `new RegExp`.
- Why the sticky flag suits tokenizers; where precedence lives in a recursive-descent parser; why `split(",")` cannot parse CSV.

The programs are a Unicode-aware word-statistics tool, a template renderer with filters and conditionals built on `replace` with a function, and an expression evaluator with variables, a sticky tokenizer and positioned errors.

EXERCISE 1

Unicode-aware word statistics

Read the whole input. Extract words with `Intl.Segmenter("en", { granularity: "word" })` keeping segments where `isWordLike` is true, normalise each to NFC and lower-case. Print `words=<count> unique=<distinct>`; `top=<three most frequent as word=n, ties by localeCompare>`;

`longest=<the distinct word with the most graphemes, ties by localeCompare> graphemes=<n> codeUnits=<length>`; `letters=<\p{L} count> digits=<\p{Nd} count> emoji=<\p{Extended_Pictographic} count>` over the raw input; and `sorted=<distinct words sorted with an Intl.Collator("en")>`.

Example: `The café was the CAFÉ we loved; the 日本語 menu had 3 items 😀` →

```
words=13 unique=10
top=the=3 café=2 3=1
longest=items graphemes=5 codeUnits=5
letters=42 digits=1 emoji=1
sorted=3 café had items loved menu the was we 日本語
```

The two spellings of café (precomposed and decomposed) became one word after normalisation.

INPUT 1

```
The café was the CAFÉ we loved; the 日本語 menu had 3 items 😀
```

OUTPUT 1

```
words=13 unique=10
top=the=3 café=2 3=1
longest=items graphemes=5 codeUnits=5
letters=42 digits=1 emoji=1
sorted=3 café had items loved menu the was we 日本語
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const segmenter = new Intl.Segmenter("en", { granularity: "word" });
const graphemes = (s) => [...new Intl.Segmenter("en", { granularity: "grapheme" }).s
const words = [...segmenter.segment(input)].filter((s) => s.isWordLike).map((s) => s
const freq = new Map();
for (const w of words) freq.set(w, (freq.get(w) ?? 0) + 1);
const top = [...freq].sort(([a, x], [b, y]) => y - x || a.localeCompare(b)).slice(0,
console.log(`words=${words.length} unique=${freq.size}`);
console.log(`top=${top.map(([w, n]) => `${w}=${n}`).join(" ")}`);
const longest = [...freq.keys()].sort((a, b) => graphemes(b) - graphemes(a) || a.loc
console.log(`longest=${longest} graphemes=${longest === "-" ? 0 : graphemes(longest)
console.log(`letters=${(input.match(/\p{L}/gu) ?? []).length} digits=${(input.match(
const collator = new Intl.Collator("en");
console.log(`sorted=${[...freq.keys()].sort(collator.compare).join(" ")}`);
```

EXERCISE 2

A template renderer built on replace

Line 1 is a JSON context; the rest is a template. Implement `render(tpl, ctx)` with three passes of `replace` with functions: `{{#each key}}…{{/each}}` repeats the body for each array element with `this` and `index` added to the context (rendering the body recursively); `{{#if key}}…{{/if}}` keeps the body when the value is truthy; `{{ path | filter | filter:arg }}` looks up a dotted path and applies filters `upper`, `lower`, `money` (USD via Intl) and `default:<text>` (used when the value is undefined, null or empty); an unknown filter throws. Missing values render as empty. Print the result.

Example context `{"name":"Ada","items":["pen","ink"],"total":12.5,"vip":true,"nick":""}` and template

```
Hello {{ name | upper }}!
{{#if vip}}Welcome back.{{/if}}
{{#each items}}- {{ this | upper }} ({{ index }})
{{/each}}Total: {{ total | money }} for {{ nick | default:friend }}{{#if missing}} r
```

→

```
Hello ADA!
Welcome back.
- PEN (0)
- INK (1)
Total: $12.50 for friend
```

INPUT 1

```
{"name":"Ada","items":
["pen","ink"],"total":12.5,"vip":true,"nick":""}
Hello {{ name | upper }}!
{{#if vip}}Welcome back.{{/if}}
{{#each items}}- {{ this | upper }} ({{ index
}})
{{/each}}Total: {{ total | money }} for {{ nick
| default:friend }}{{#if missing}} never{{/if}}
```

OUTPUT 1

```
Hello ADA!
Welcome back.
- PEN (0)
- INK (1)
Total: $12.50 for friend
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const [contextLine, ...templateLines] = input.split("\n");
const context = JSON.parse(contextLine);
const template = templateLines.join("\n").trimEnd();
const filters = {
  upper: (v) => String(v).toUpperCase(),
  lower: (v) => String(v).toLowerCase(),
  money: (v) => new Intl.NumberFormat("en-US", { style: "currency", currency: "USD"
  default: (v, arg) => (v === undefined || v === null || v === "" ? arg : v),
};
const lookup = (ctx, path) => path.split(".").reduce((o, k) => (o === undefined || o
function render(tpl, ctx) {
  let out = tpl.replace(/\{\{#each (\w+)\}\}([\s\S]*?)\{\{\/each\}\}/g, (_, key, boc
    const list = lookup(ctx, key);
    return Array.isArray(list) ? list.map((item, i) => render(body, { ...ctx, this:
  });
  out = out.replace(/\{\{#if (\w+)\}\}([\s\S]*?)\{\{\/if\}\}/g, (_, key, body) => (l
  out = out.replace(/\{\{\s*([\w.]+)((?:\s*\|\s*\w+(?::[^|}]+)?)*)\s*\}\}/g, (_, pat
    let value = lookup(ctx, path);
    for (const step of pipes.split("|").map((s) => s.trim()).filter(Boolean)) {
      const [name, arg] = step.split(":");
      if (!filters[name]) throw new SyntaxError(`unknown filter '${name}'`);
      value = filters[name](value, arg);
    }
    return value === undefined || value === null ? "" : String(value);
  });
  return out;
}
console.log(render(template, context));
```

EXERCISE 3

A calculator with variables and exponents

Extend the expression evaluator: tokens are numbers, identifiers and `+ - * / ^ ( ) =`. A line is either `name = expr` (store the value in a `Map`, print `name = value`) or an expression (print its value). Grammar: `term := unary (('*'|'/') unary)*`, `unary := '-' unary | power`, `power := primary ('^' unary)?` — so `^` is right-associative, binds tighter than unary minus (`-2 ^ 2` is −4) and allows a negative exponent. An unknown variable throws `ReferenceError("unknown variable 'z' at <pos>")`; other errors as in the previous exercise. Variables persist across lines.

Example:

```
x = 3 + 4
y = x * 2
y - x
2 ^ 3 ^ 2
-2 ^ 2
z + 1
```

→

```
x = 7
y = 14
7
512
-4
ReferenceError: unknown variable 'z' at 0
```

INPUT 1

```
x = 3 + 4
y = x * 2
y - x
2 ^ 3 ^ 2
-2 ^ 2
(x + 1) / 0
z + 1
x =
4 4
x @ 2
```

OUTPUT 1

```
x = 7
y = 14
7
512
-4
RangeError: division by zero at 8
ReferenceError: unknown variable 'z' at
0
SyntaxError: unexpected end of input at
4
SyntaxError: unexpected '4' at 2
SyntaxError: unexpected character '@' at
2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const TOKEN = /\s*(?:(?<num>\d+(?:\.\d+)?)|(?<id>[A-Za-z_]\w*)|(?<op>[-+*/^()=]))/y;
const vars = new Map();
function tokenize(src) {
  const tokens = [];
  TOKEN.lastIndex = 0;
  while (TOKEN.lastIndex < src.length) {
    const start = TOKEN.lastIndex;
    const m = TOKEN.exec(src);
    if (!m) {
      const rest = src.slice(start);
      if (/^\s*$/.test(rest)) break;
      const at = start + rest.match(/^\s*/)[0].length;
      throw new SyntaxError(`unexpected character '${src[at]}' at ${at}`);
    }
    const type = Object.keys(m.groups).find((k) => m.groups[k] !== undefined);
    tokens.push({ type, value: m.groups[type], pos: m.index + m[0].length - m.groups
  }
  tokens.push({ type: "eof", value: "", pos: src.length });
  return tokens;
}
function run(src) {
  const tokens = tokenize(src);
  let i = 0;
  const peek = () => tokens[i];
  const next = () => tokens[i++];
  const describe = (t) => (t.type === "eof" ? "end of input" : `'${t.value}'`);
  function statement() {
    if (tokens[0].type === "id" && tokens[1].value === "=") {
      const name = next().value; next();
      const value = expr();
      vars.set(name, value);
      return `${name} = ${value}`;
    }
    return String(expr());
  }
  function expr() {
    let left = term();
    while (peek().value === "+" || peek().value === "-") { const op = next().value;
    return left;
  }
  function term() {
    let left = unary();
    while (peek().value === "*" || peek().value === "/") {
      const op = next(); const right = unary();
      if (op.value === "/" && right === 0) throw new RangeError(`division by zero at
      left = op.value === "*" ? left * right : left / right;
    }
```

```
    return left;
  }
  function unary() {
    if (peek().value === "-") { next(); return -unary(); }       // binds looser tha
    return power();
  }
  function power() {
    const base = primary();
    if (peek().value === "^") { next(); return base ** unary(); }   // right-associa
    return base;
  }
  function primary() {
    const t = peek();
    if (t.type === "num") { next(); return Number(t.value); }
    if (t.type === "id") {
      next();
      if (!vars.has(t.value)) throw new ReferenceError(`unknown variable '${t.value}
      return vars.get(t.value);
    }
    if (t.value === "(") {
      next();
      const v = expr();
      if (peek().value !== ")") throw new SyntaxError(`expected ')' but found ${desc
      next();
      return v;
    }
    throw new SyntaxError(`unexpected ${describe(t)} at ${t.pos}`);
  }
  const result = statement();
  if (peek().type !== "eof") throw new SyntaxError(`unexpected ${describe(peek())} a
  return result;
}
for (const line of lines) {
  try { console.log(run(line)); }
  catch (err) { console.log(`${err.name}: ${err.message}`); }
}
```

CHECKPOINT QUESTIONS

1. `[..."👩‍👩‍👧"].length` versus its grapheme count:

A. Both 1

B. 5 code points (three people and two joiners) but 1 grapheme

C. 8 and 1

D. 1 and 5

2. `"e\u0301".normalize("NFC")` is…

A. Unchanged

B. `"\u00e9"` — NFC composes base + combining mark into the precomposed character

C. `"e"`

D. An error

3. A reused `Intl.NumberFormat` matters because…

A. Formatting differs per call

B. Construction loads locale data and is slow; `format` is cheap — build once, call many times

C. It caches results

D. Locales change

4. `String.raw\` \n` is…

A. A newline

B. The two characters backslash and `n` — the raw literal text

C. An empty string

D. `"\\\\n"`

5. `/a|ab/.exec("ab")[0]` is…

A. `"ab"`

B. `"a"` — alternation tries alternatives left to right and takes the first that matches

C. `null`

D. `"b"`

6. A capturing group inside a quantifier, `(\d,)+` on `1,2,3,`, captures…

A. All three

B. Only the last iteration, `3,` — use `matchAll` to collect every repetition

C. The first

D. Nothing

7. `(?<=\$)\d+`…

A. Matches `$` and digits

B. Matches digits preceded by `$` without including the `$` — a lookbehind

C. Is invalid

D. Matches digits followed by `$`

8. To make `(\w+\s?)*` safe you would…

A. Add the `u` flag

B. Rewrite it so each repetition is unambiguous, e.g. `\w+(\s\w+)*`, and anchor it

C. Add `i`

D. Use `.*` instead

9. The `d` flag adds…

A. Digit matching

B. `.indices` on match results — start/end offsets for each group

C. Dot-all

D. Debug output

10. Right-associativity of `^` in a recursive-descent parser is achieved by…

A. A `while` loop

B. Recursing on the right operand (`base ** power()`) instead of looping

C. Reversing the tokens

D. A lookup table

11. Separating the tokenizer from the parser means…

A. Two files

B. The parser sees typed tokens with positions and never raw characters; whitespace and comments vanish in the tokenizer

C. Slower parsing

D. No error messages

12. `"".split(",")` returns…

A. `[]`

B. `[""]` — one empty field

C. `null`

D. `[","]`

ANSWER KEY · CHECKPOINT — STRINGS, UNICODE, REGULAR EXPRESSIONS AND PARSING

1. B `.length` would say 8 code units.

2. B NFD does the reverse; NFKC additionally folds compatibility characters.

3. B Also pass an explicit locale (and `timeZone` for dates).

4. B Every tag receives `strings.raw` alongside the cooked parts.

5. B Put longer alternatives first when they share a prefix.

6. B Groups hold one value each.

7. B Lookbehind arrived in ES2018 and V8 supports variable length.

8. B Negated classes, anchors and bounded repeats are the other tools.

9. B Handy for highlighting and for error positions.

10. B Loops give left-associativity — right for `+` and `*`.

11. B `peek` / `next` / `expect` are then the only ways the parser touches input.

12. B A recurring surprise when parsing empty lines.

CodeKairo · JavaScript study plan · Module 10: Strings, Unicode, regular expressions and parsing. Solve the exercises online to track your progress and earn XP.