Building a Web Browser Engine: Parsing & the DOM

A Real HTML Tokenizer, DOM Parser, CSS Engine & Style Tree — From Scratch

Topics covered:

A forgiving HTML tokenizer & stack-based DOM parser

A real CSS tokenizer, parser & selector matcher

Specificity, the cascade & inheritance

The style tree & a real user-agent stylesheet

Capstone: a real HTML+CSS document parsed end to end into a fully-resolved style tree

Exercises: 27 hands-on exercises with worked, verified solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Build a Browser Engine? HTML, CSS & the Rendering Pipeline

Building a Web Browser Engine: Parsing & the DOM

Chapter 1 · Why Build a Browser Engine? HTML, CSS & the Rendering Pipeline

Type a URL, and somewhere between pressing Enter and seeing a page, raw bytes of HTML and CSS text turn into an arrangement of colored rectangles and text on a screen. This two-course project builds the part of that journey that starts once the HTML and CSS text already exist — no networking, no JavaScript, nothing borrowed from a real browser engine's own source code. Just the actual pipeline, built by hand, in Python, verified at every step the same way every other course on this site has been: by running it and checking the answer, not by asserting it.

The Four-Stage Pipeline: Parse → Style → Layout → Paint

Every real browser engine — however large and however optimized — does the same four things to turn markup into pixels. This project builds each one, split across two courses.

STAGE	INPUT	OUTPUT	BUILT IN
Parse	Raw HTML text, raw CSS text	A DOM tree, a stylesheet	This course, Chapters 2-5
Style	The DOM tree + the stylesheet	A style tree — every DOM node paired with its own fully resolved computed style	This course, Chapters 6-9
Layout	The style tree	A layout tree — every box's own real width, height, and position	Course 2, Chapters 1-6
Paint	The layout tree	An actual pixel buffer	Course 2, Chapters 7-10

Each stage's own output is exactly the next stage's own input — a real pipeline, not four unrelated topics loosely grouped under one course title. By the end of Course 2's own capstone, a real static web page will go in as two text files and come out as an actual rendered image.

This Project's Own Honest Scope

Stated up front, the same way every other course in this site's own "ambitious learning projects" tier states its scope before writing a line of code: this is **build-to-understand, not build-to-ship**.

- **No JavaScript.** No `<script>` execution, no DOM mutation after the initial parse.
- **No networking.** HTML and CSS arrive as plain Python strings — no HTTP, no fetching linked stylesheets or images.
- **No images.** Layout and paint both work purely with boxes, text, and color.
- **No Flexbox, no Grid, no forms.** Block and inline layout only — the two layout modes CSS actually started with.
- **No real font rendering.** Course 2's own text layout uses a simplified, honest glyph-width model, not a real font-rasterization library.
- **A software-rasterized pixel buffer, not a real window.** The final output is a real image file, not anything drawn to actual screen hardware.

None of these are apologized-for gaps discovered too late — they're the same kind of deliberate, stated-in-advance boundary this site's own `compiler1` / `compiler2` pair drew around JavaScript-style dynamic typing extras, and the same kind every course in this tier draws to stay genuinely finishable while still being genuinely real.

Why Not Just Use Regex?

Before building a real parser, it's worth trying the shortcut everyone tries first — and seeing exactly where it breaks, concretely, rather than taking "you can't parse HTML with regex" on faith.

```
import re

def naive_extract(html, tag):
    pattern = f"<{tag}>(.*?)</{tag}>"
    return re.findall(pattern, html, re.DOTALL)
```

VERIFIED DIRECTLY — NESTING BREAKS IT, EXACTLY AS THE WELL-KNOWN WARNING SAYS, AND GETS WORSE THE DEEPER IT GOES

`naive_extract("<div><div>Inner</div></div>", "div")` returns `['<div>Inner']` — not the true outer content, `"<div>Inner</div>"`. The regex has no concept of nesting depth at all; it just scans forward from the first `<div>` and captures raw characters until it hits the *first literal* `</div>` it finds — completely oblivious that an inner `<div>` opened in between. Three levels of nesting makes it worse, not better: `"<div><div><div>Deep</div></div></div>"` returns `['<div><div>Deep']` — two stray, unclosed opening tags leaked straight into the "content."

THIS ISN'T A FIXABLE REGEX BUG — IT'S A REAL, STRUCTURAL LIMIT

Regular expressions match *regular* languages — patterns with no memory of how deep they've gone. HTML's own nesting is a *context-free* structure — correctly matching an outer tag's own closing tag requires remembering how many same-named tags have opened since, which a regex engine has no mechanism to track at all. Switching to a greedy `.*` instead of `.*?` doesn't fix this — it trades the nesting bug for the opposite failure, matching all the way from the first opening tag to the very *last* closing tag in the document, swallowing unrelated sibling elements in between.

What a Real Parser Needs Instead: Tracking Depth

The fix a real parser needs is small enough to preview here, one paragraph before Chapter 2 builds the real thing properly: track how many tags deep the scan currently is, and only treat a closing tag as the *match* once that count returns to zero.

```
def depth_aware_extract_first(html, tag):
    open_tag, close_tag = f"<{tag}>", f"</{tag}>"
    start = html.find(open_tag)
    pos = start + len(open_tag)
    depth = 1
    while depth > 0:
        next_open = html.find(open_tag, pos)
        next_close = html.find(close_tag, pos)
        if next_open != -1 and next_open < next_close:
            depth += 1; pos = next_open + len(open_tag)     # one level deeper
        else:
            depth -= 1; content_end = next_close; pos = next_close + len(close_tag)
    return html[start + len(open_tag): content_end]
```

VERIFIED DIRECTLY — TRACKING DEPTH GETS THE NESTED CASE EXACTLY RIGHT

On the identical nested input, `depth_aware_extract_first` returns `"<div>Inner</div>"` — the true, complete content of the outer `div`. Given a harder input mixing nesting *and* trailing text before the real close — `"<div><div>Inner</div>after inner</div><div>Second</div>"` — it correctly returns `"<div>Inner</div>after inner"`, stopping at the right closing tag even with a second, unrelated `div` sitting right after it in the source.

This is a genuine preview, not a toy — Chapter 2's own real tokenizer and parser are a more complete, more careful version of exactly this idea: never trust a closing tag in isolation, always track what's currently open.

A First Look at a Real DOM Tree

Before building a from-scratch tokenizer, it's worth seeing what the *target* actually looks like. Python's own standard library includes an HTML parser — used here purely to illustrate the shape of a DOM tree for this

one chapter, not reused anywhere else in this course. Chapter 2 builds a real tokenizer and parser completely from scratch.

```
source = "<div>\n  <p>Hello</p>\n</div>"
```

VERIFIED DIRECTLY — A REAL, IF ILLUSTRATIVE, DOM TREE FOR A TWO-ELEMENT DOCUMENT

```
Element('#document')
  Element('div')
    Element('p')
      Text('Hello')
```

`div` contains `p`, which contains the text `"Hello"` — a tree, not a flat list, with each element's own children nested directly inside it. A sibling structure, `"<ul><li>One</li><li>Two</li></ul>"`, produces two separate `li` elements as *direct children of the same parent* — `ul` — rather than nested inside one another, confirming the tree correctly distinguishes "nested inside" from "next to."

From DOM to Pixels — Sketching the Rest of the Pipeline

The same tiny `div`/`p`/"Hello" example, carried conceptually through every remaining stage this two-course project builds — not yet computed for real, since the cascade, the box model, and the rasterizer don't exist yet, but concrete enough to see where each future chapter's own work actually lands.

STAGE	WHAT IT ADDS TO THIS EXAMPLE
DOM tree (above)	`div` → `p` → `"Hello"`, with no notion of color, size, or position at all
Style tree (Ch.6-9)	Every node paired with its own resolved values — e.g. `div` might resolve to `padding: 10px`, `p` to `color: blue`, inherited down from whatever the stylesheet and the browser's own defaults specify
Layout tree (Course 2, Ch.1-6)	Real numbers — `div`'s own box might resolve to position `(0, 0)`, size `300×40`; `p`'s own line box sits inside it, inset by that `10px` padding
Pixels (Course 2, Ch.7-10)	An actual image — the `div`'s own background color filled into its box's own rectangle, "Hello" rendered as real glyphs at the `p`'s own resolved position

NOTHING HERE WAS COMPUTED — THAT'S DELIBERATE

This table is a map, not a result. Every specific number above (`10px`, `(0, 0)`, `300×40`) is illustrative, chosen to be plausible, not derived from any real cascade or layout algorithm — those don't exist in this course yet. The point is orientation: knowing what a "style tree" or a "layout tree" *is*, concretely, before spending several chapters building the machinery that actually produces one correctly.

Where Each Future Chapter Fits

CHAPTER	BUILDS
2	A real, hand-written HTML tokenizer — the depth-tracking idea above, generalized properly
3	A real HTML parser, building an actual DOM tree from the token stream
4	A CSS tokenizer and parser, building a real `Stylesheet` structure
5	Selectors and selector matching — deciding whether one CSS rule applies to one DOM node
6	Specificity and the cascade — deciding which rule wins when more than one matches
7	Inheritance and computed values
8	The style tree itself — the DOM tree, the cascade, and inheritance combined into one structure
9	A real default stylesheet — why `div` and `span` behave differently even with no CSS written at all
10	Capstone — a real document parsed all the way to a verified style tree

Hands-On Exercises

EXERCISE 1

Using this chapter's own `naive_extract`, test a four-level-deep nested input — `"<div><div><div><div>Deepest</div></div></div></div>"` — for the tag `"div"`. Determine exactly what gets captured, and explain the pattern connecting nesting depth to how many stray opening tags leak into the broken result.

View solution

EXERCISE 2

Using this chapter's own `depth_aware_extract_first`, extract the `"p"` content from `"<p>Hello <b>bold</b> world</p>"` — a tag containing a genuinely different nested tag, not a repeat of itself. Verify the result, and explain why this case doesn't even trip up the naive regex version — what's specifically different about same-tag nesting versus different-tag nesting?

View solution

EXERCISE 3

Using this chapter's own illustrative `TinyDOMBuilder`, build and print the DOM tree for `"<div><p>One</p><p>Two</p></div>"` — two sibling paragraphs inside one div. Sketch the expected tree shape by hand first, then verify your sketch against the actual printed output.

📄 View solution

CHAPTER 1 QUICK REFERENCE

- **The pipeline:** parse (text → DOM + stylesheet) → style (+ cascade → style tree) → layout (→ boxes with real positions) → paint (→ pixels)
- **Scope:** static HTML+CSS only — no JS, no networking, no images, no Flexbox/Grid, no real font rendering, a software-rasterized pixel buffer
- **Verified:** a naive regex tag-extractor breaks on nested elements exactly as the well-known warning predicts, and gets worse with each additional level of nesting
- **Verified:** a small depth-aware scan — tracking how many tags are currently open — gets the same nested case exactly right, previewing Chapter 2's own real approach
- **Verified:** a real, if illustrative, DOM tree correctly distinguishes nested elements from sibling elements
- **Next chapter:** Tokenizing HTML: A Real, Forgiving Lexer — building the actual, from-scratch tokenizer this chapter only previewed

CHAPTER 2 OF 10

Tokenizing HTML: A Real, Forgiving Lexer

Building a Web Browser Engine: Parsing & the DOM

Chapter 2 · Tokenizing HTML: A Real, Forgiving Lexer

Chapter 1's depth-aware scan was a preview, built to answer one narrow question — how does an outer tag's own closing tag get found correctly. A real tokenizer has to answer a lot more: which tags never get a closing tag at all, what an attribute actually looks like, how a comment stays safe from the very characters that normally start and end a tag. This chapter builds that tokenizer for real — the thing every later chapter in this course actually consumes.

Five Kinds of Token

```
class Token:
    def __init__(self, kind, name=None, attrs=None, text=None):
        self.kind = kind    # 'opentag' | 'closetag' | 'voidtag' | 'text' | 'comment'
        self.name = name
        self.attrs = attrs
        self.text = text
```

`voidtag` is its own kind, deliberately separate from `opentag` — a real, necessary distinction the rest of this chapter is mostly about.

Void Elements: Tags That Never Get a Closing Tag

A fixed, small set of HTML elements are defined to never have content and never get a closing tag at all — `<br>`, `<img>`, `<input>`, `<hr>`, and a dozen others. A tokenizer that doesn't know this list will wait forever for a `</br>` that a real HTML document is never going to contain.

```
VOID_ELEMENTS = frozenset({
    'area', 'base', 'br', 'col', 'embed', 'hr', 'img', 'input',
    'link', 'meta', 'param', 'source', 'track', 'wbr',
})
```

VERIFIED DIRECTLY — A TRAILING SLASH MAKES ZERO DIFFERENCE FOR A REAL VOID ELEMENT

`tokenize("<br>")` and `tokenize("<br/>")` produce **byte-for-byte identical** token lists — a single `voidtag('br', {})` either way. Whether or not the source bothered to write the XML-style trailing slash, `br` was always going to be treated as content-free.

The Surprising Part: a Trailing Slash on an Ordinary Element Does Nothing

`<div/>` looks like it should close itself, the way it would in XML or JSX. Real HTML doesn't work that way — per the actual HTML5 spec, a trailing `/` on anything *other* than a void element (or an SVG/MathML element, out of this course's own scope) is simply **ignored**. The element opens normally and stays open.

VERIFIED DIRECTLY — THREE "SELF-CLOSING" DIVS IN A ROW NEST THREE LEVELS DEEP INSTEAD

`tokenize("<div/><div/><div/>Hi</div></div></div>")` produces **three** `opentag('div', {})` **tokens**, then `text('Hi')`, then **three** `closetag('div')` **tokens** — never a single `voidtag`. Every one of those three `/` characters was silently thrown away. A real browser renders this exact markup with `"Hi"` genuinely nested three `div`s deep — not as three empty, self-closed boxes sitting side by side, which is what the JSX-trained eye expects on sight.

WHY THIS MATTERS FOR THE TOKENIZER SPECIFICALLY, NOT JUST TRIVIA

Getting this wrong here would silently corrupt every later chapter — if the tokenizer emitted a `voidtag` for `<div/>`, Chapter 3's own parser would never expect a matching `</div>`, and a real, valid document using this pattern would parse into a subtly wrong tree with no error raised anywhere. The fix lives entirely in one place: only treat the trailing slash as meaningful when the tag name is already in `VOID_ELEMENTS`.

Attributes: Three Shapes, One Parser

A single attribute string can hold double-quoted values, single-quoted values, unquoted values, and bare boolean attributes with no value at all — often all in the same tag.

VERIFIED DIRECTLY — ALL THREE QUOTING STYLES, PLUS A BOOLEAN ATTRIBUTE, PARSED CORRECTLY IN ONE TAG

`tokenize('<div class="hello" id=\'world\' disabled>')` produces attributes `{'class': 'hello', 'id': 'world', 'disabled': None}` — `None` specifically marking a boolean attribute, distinguishable from an attribute that was genuinely set to an empty string. Unquoted values work too: `<input type=text value=42>` parses to `{'type': 'text', 'value': '42'}`, scanning up to the next whitespace instead of a matching quote character.

Comments: Safe From the Characters That Normally Matter

The main scan loop treats `<` as the start of something structural and `>` as the end of a tag — but neither should mean anything *inside* a comment. Comments get their own dedicated scan, hunting specifically for the literal three-character sequence `-->`, ignoring everything else in between.

VERIFIED DIRECTLY — A COMMENT SURVIVES BOTH A STRAY '>' AND A STRAY '<' INSIDE IT

`tokenize("<!-- a > b --><p>after</p>")` produces one `comment(' a > b ')` token, correctly followed by a real `p` element — the internal `>` never terminated the comment early. The identical result holds for a stray `<`: `"<!-- a < b -->"` produces `comment(' a < b ')`, confirming the comment scanner is genuinely delimiter-aware, not merely "looking for the next `>`" the way the tag scanner is.

Case-Insensitivity

VERIFIED DIRECTLY — TAG NAMES NORMALIZE TO LOWERCASE REGARDLESS OF SOURCE CASING

`tokenize("<DIV>Hi</DIV>")` produces `opentag('div', {})` and `closetag('div')` — both lowercased, matching real HTML's own case-insensitive tag names. Every later chapter — selector matching, the cascade, the default stylesheet — gets to assume a tag name is always already lowercase, because the tokenizer normalized it once, here, rather than every consumer needing to remember to do it themselves.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A dedicated `voidtag` token, distinct from `opentag`	Chapter 3's own parser — a void element never gets pushed onto the open-element stack at all, sidestepping the entire "when does this close" question for the 14 tags that never need it answered
A trailing `/` is ignored unless the tag is void	Chapter 3's own stack-based parser, which will nest `<div/><p>...</p></div>` exactly as deeply as this chapter's own token stream implies — the parser doesn't re-decide this; the tokenizer already did
Attribute values default to `None` for boolean attributes	A forward reference to real CSS attribute selectors and form-control defaults — out of this course's own stated scope, but the distinction (present-with-no-value vs. genuinely absent) is preserved here regardless, for free

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Tag names normalized to lowercase during tokenizing	Chapter 5's own selector matching — a CSS selector like `div` can be compared directly against a DOM node's own tag name with a plain string equality check, no case-folding needed at match time

Hands-On Exercises

EXERCISE 1

Tokenize `"<div/><div/><div/>Hi</div></div></div>"` — three "self-closing" divs in a row, matched by three real closing tags — using this chapter's own `tokenize`. Confirm the exact sequence of token kinds produced, and explain what a real browser would visually render for this markup (in terms of nesting, not exact pixels).

View solution

EXERCISE 2

Tokenize `"<input type=text value=42>"` using this chapter's own `tokenize`. Confirm the resulting token's own `kind` and `attrs`, and explain why `input` produces a single token with no separate closing tag expected anywhere, even though this particular tag has real attributes on it.

View solution

EXERCISE 3

Tokenize `"<!-- a < b --><p>after</p>"` — a comment containing a literal `<` rather than this chapter's own `>` example. Verify the comment's own text is captured correctly and that the following `p` element still tokenizes normally, then explain specifically which part of `tokenize`'s own comment-handling branch is responsible for the `<` inside the comment never being treated as the start of a new tag.

View solution

CHAPTER 2 QUICK REFERENCE

- **Five token kinds:** `opentag`, `closetag`, `voidtag`, `text`, `comment`
- **Void elements:** a fixed 14-name set (`br`, `img`, `input`, ...) that never get a closing tag, tokenized as their own dedicated kind
- **Verified — the real, surprising quirk:** a trailing `/` is ignored for any element that isn't void; `<div/>` opens normally and stays open, confirmed with three "self-closing" divs nesting three levels deep instead of closing
- **Attributes:** double-quoted, single-quoted, unquoted, and boolean (value `None`) all handled by one parser — verified on all four shapes at once

- **Comments:** scanned for the literal `-->` sequence specifically — verified safe from both a stray `>` and a stray `<` inside
- **Verified:** tag names normalize to lowercase during tokenizing, so no later chapter needs to case-fold them again
- **Next chapter:** Parsing HTML into a DOM Tree — turning this token stream into the real, stack-based tree Chapter 1 only previewed

CHAPTER 3 OF 10

Parsing HTML into a DOM Tree

Building a Web Browser Engine: Parsing & the DOM

Chapter 3 · Parsing HTML into a DOM Tree

Chapter 2's tokenizer produces a flat list — every open tag, close tag, void tag, run of text, and comment, one after another, with no notion of nesting at all. This chapter turns that list into an actual tree: a real `Element` / `Text` / `Comment` hierarchy, built by a stack-based parser that tracks which elements are currently open, the same way Chapter 1's own preview scan tracked depth for one tag at a time — generalized here to the whole document at once.

Three Real Node Types

```
class DOMNode:
    def __init__(self, kind):
        self.kind = kind
        self.children = []

class Element(DOMNode):
    def __init__(self, tag, attrs):
        super().__init__('element')
        self.tag = tag
        self.attrs = attrs

class TextNode(DOMNode):
    def __init__(self, text):
        super().__init__('text')
        self.text = text

class CommentNode(DOMNode):
    def __init__(self, text):
        super().__init__('comment')
        self.text = text
```

Unlike Chapter 1's own illustrative tree, these are the real node types this course keeps using for the rest of both courses — including `CommentNode`, which the tree stores faithfully even though it will simply never be visible once painting exists.

The Parser: a Stack of Currently-Open Elements

```
def parse(tokens):
    root = Element('#document', {})
    stack = [root]
    for tok in tokens:
        if tok.kind == 'opentag':
            node = Element(tok.name, tok.attrs)
            stack[-1].children.append(node)
            stack.append(node)
        elif tok.kind == 'voidtag':
            node = Element(tok.name, tok.attrs)
            stack[-1].children.append(node)     # never pushed -- no close expected
        elif tok.kind == 'closetag':
            for i in range(len(stack) - 1, 0, -1):
                if getattr(stack[i], 'tag', None) == tok.name:
                    del stack[i:]; break
        elif tok.kind == 'text':
            stack[-1].children.append(TextNode(tok.text))
        elif tok.kind == 'comment':
            stack[-1].children.append(CommentNode(tok.text))
    return root
```

`voidtag`'s own special treatment carries straight through from Chapter 2: it becomes a child of whatever's currently open, but is never itself pushed onto the stack — there's nothing to later pop, because nothing will ever try to close it. The `closetag` branch searches from the top of the stack downward rather than assuming the top is always the right match — a real, deliberate defense against malformed markup, not just the tidy case.

VERIFIED DIRECTLY — A STRAY, MISMATCHED CLOSING TAG DOESN'T CRASH THE PARSER

`parse_html("<p>Hello</div>")` — a closing `</div>` that was never opened — produces a perfectly valid tree with `p` still open and containing `"Hello"`. The search for a matching `div` on the stack simply finds nothing and the token is silently ignored, exactly the way real browsers treat an orphaned closing tag.

The Real Quirk: `<p>` Closes Itself When It Has To

Real HTML lets a huge number of closing tags be omitted entirely — the parser is expected to work out where an element really ends from context. The single most common case: an open `<p>` is implicitly closed the moment certain other elements begin, even with no `</p>` anywhere in the source.

```
# tags that implicitly close an open <p>, per the real HTML5 spec
P_CLOSING_TAGS = frozenset({
    'address', 'article', 'aside', 'blockquote', 'details', 'div', 'dl',
    'fieldset', 'figcaption', 'figure', 'footer', 'form',
    'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'header', 'hr',
    'main', 'menu', 'nav', 'ol', 'p', 'pre', 'section', 'table', 'ul',
})

# inside the opentag branch, checked BEFORE pushing the new element:
if (tok.name in P_CLOSING_TAGS and
        getattr(stack[-1], 'tag', None) == 'p'):
    stack.pop()      # the open <p> is done, whether the source said so or not
```

VERIFIED DIRECTLY — TWO CONSECUTIVE <P> TAGS WITH ONLY ONE CLOSING TAG PRODUCE TWO REAL SIBLINGS

`parse_html("<p>Hello<p>World</p>")` produces **two separate sibling `p` elements** — one containing `"Hello"`, one containing `"World"` — not one `p` nested inside another. The first `<p>` is popped off the stack the instant the second one begins, purely because `'p'` is itself in `P_CLOSING_TAGS`. The exact same thing happens with no closing tag written anywhere at all: `"<p>Hello<div>World</div>"` also produces two top-level siblings, `p` and `div` — the `div` is never nested inside the `p`, even though the source text never once wrote `</p>`.

NOT EVERY ELEMENT TRIGGERS THIS — ONLY THE ONES ON THE LIST

`parse_html("<p>Hello<span>World</span></p>")` keeps `span` correctly **nested inside** `p`, because `span` was never a block-level element and was never added to `P_CLOSING_TAGS`. The rule isn't "any new tag closes an open `p`" — it's a specific, named list, and getting the list wrong in either direction (too broad or too narrow) produces a genuinely wrong tree.

A Real, Honest Limitation — Found by Testing It

`<li>` has an almost identical real-world rule: a new `<li>` is supposed to implicitly close a previous open `<li>`, the exact same shape as the `p` rule above. This chapter's own parser doesn't implement it.

VERIFIED DIRECTLY — A SECOND <LI> NESTS INSIDE THE FIRST INSTEAD OF CLOSING IT

`parse_html("<ul><li>One<li>Two</li></ul>")` — real, valid, extremely common HTML — produces the second `li` genuinely **nested inside** the first, not as a sibling. A real browser renders this as two separate list items; this chapter's own parser, as written, would not. `P_CLOSING_TAGS` only ever checks whether the currently-open element is a `p` — it has no equivalent check for an open `li` at all.

This isn't a bug hidden until an exercise reveals it — it's a genuine, stated scope boundary: this chapter builds the single most common implied-closing rule (`p`) to establish the technique, not an exhaustive table of every

element's own omission rules HTML actually defines. A real browser engine's own parser has dozens of rules like this one; this course builds one, correctly and completely, as a real, working example of the pattern.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A stack of currently-open elements, generalized from Chapter 1's own single-tag depth counter	Chapter 1's own `depth_aware_extract_first` — the same underlying idea, now tracking an entire document's own nesting at once instead of one tag pair
`voidtag` tokens never get pushed onto the stack	Chapter 2's own dedicated `voidtag` token kind — this chapter is where that distinction actually pays off, sidestepping any need to guess whether a `br` or `img` is "closed"
The real DOM tree this chapter produces	Chapter 8's own style tree, which pairs every one of these exact `Element` nodes with a fully resolved computed style — nothing about this tree's own shape changes between now and then
The `li` gap, found and left honestly unfixed	A direct parallel to this site's own established practice (Course 1 and Course 2 of the compiler project both did this) of naming a real, verified limitation rather than quietly working around it or pretending it doesn't exist

Hands-On Exercises

EXERCISE 1

Parse `"<p>Hello<ul><li>x</li></ul>"` and `"<p>Hello<h1>Title</h1>"` using this chapter's own `parse_html`. Confirm both produce two top-level siblings rather than nested elements, and explain why `ul` and `h1` both trigger the same implied-closing behavior `div` does.

View solution

EXERCISE 2

Parse `"<p>A<p>B<p>C</p>"` — three consecutive `<p>` tags, only the last one explicitly closed. Determine the exact number of top-level elements produced and confirm each one's own text content, then trace through the parser's own stack contents at each of the three `opentag` events to explain why chaining this rule works correctly without any special handling beyond the single check already shown in this chapter.

View solution

EXERCISE 3

Add an analogous rule fixing this chapter's own `<li>` limitation — a new `<li>` should implicitly close a previously open `<li>`, the same shape as the existing `p` check. Verify `parse_html("<ul><li>One<li>Two</li></ul>")` now

produces two sibling `li` elements instead of one nested inside the other, and confirm the existing `p`-closing tests from this chapter still pass unchanged.

View solution

CHAPTER 3 QUICK REFERENCE

- **Three real node types:** `Element`, `TextNode`, `CommentNode` — the actual tree the rest of this course keeps using
- **The parser:** a stack of currently-open elements; `opentag` pushes, `closetag` pops (searching down the stack, not just checking the top), `voidtag` is never pushed at all
- **Verified:** a stray, mismatched closing tag is silently ignored rather than crashing the parser
- **Verified — the real quirk:** an open `<p>` is implicitly closed by a following block-level element (another `p`, a `div`, a heading, a `ul`) even with no `</p>` anywhere in the source
- **Verified — the contrast:** non-block elements like `span` never trigger this rule; nesting stays intact
- **Verified — an honest limitation:** `<li>` needs its own analogous rule this chapter's own parser doesn't implement; a second `<li>` nests incorrectly instead of closing the first
- **Next chapter:** Tokenizing and Parsing CSS — building the stylesheet half of the pipeline this course's own style tree will need

CHAPTER 4 OF 10

Tokenizing and Parsing CSS

Building a Web Browser Engine: Parsing & the DOM

Chapter 4 · Tokenizing and Parsing CSS

CSS's own grammar is a lot simpler than HTML's — no implicit closing rules, no fourteen special-cased element types, no ambiguity about what's a tag versus what's content. One chapter is enough to take CSS source text all the way to a real, structured `Stylesheet` — the second half of what Chapter 8's own style tree will need, alongside the DOM tree Chapter 3 already built.

Three Small Structures

```
class Declaration:
    def __init__(self, property, value):
        self.property = property
        self.value = value

class Rule:
    def __init__(self, selectors, declarations):
        self.selectors = selectors          # a LIST -- one rule can match several selectors
        self.declarations = declarations

class Stylesheet:
    def __init__(self, rules):
        self.rules = rules
```

`selectors` stays a flat list of raw strings in this chapter — `['h1', 'h2']` for `h1, h2 { ... }` — not yet parsed into anything structured. Chapter 5 is where a selector string actually becomes something that can be matched against a DOM node.

Comments First, Before Anything Else

CSS comments (`/* ... */`) can legally appear almost anywhere — between rules, inside a declaration block, in the middle of a selector list. Stripping them out in one pass, before any real parsing starts, means nothing downstream ever has to think about them again.

```
def strip_comments(css):
    result = []
    i = 0
    while i < len(css):
        if css[i:i+2] == '/*':
            end = css.find('*/', i + 2)
            i = end + 2 if end != -1 else len(css)
        else:
            result.append(css[i]); i += 1
    return ''.join(result)
```

The Real Quirk: CSS Comments Don't Nest

Writing what looks like a comment inside a comment — to temporarily disable a block that already has its own comment in it, say — doesn't do what it looks like it should.

VERIFIED DIRECTLY — THE FIRST "*/" ENDS THE COMMENT, NO MATTER HOW MANY "/*" CAME BEFORE IT

`strip_comments("/* outer /* inner */ still outer */ p { color: red; }")` returns `" still outer */ p { color: red; }"` — not an empty string. The scanner has no concept of comment depth; it opens on the first `/*` and closes on the very next `*/` it finds, full stop. Everything after that first closing marker — including the second, now-orphaned `*/` — is treated as ordinary CSS text.

VERIFIED DIRECTLY — FOLLOWED ALL THE WAY THROUGH THE PARSER, THIS SILENTLY CORRUPTS A REAL SELECTOR, WITH NO ERROR ANYWHERE

Feeding that exact string through the full `parse_css` produces one rule whose own `selectors` is `['still outer */ p']` — not the clean `['p']` a developer obviously intended. The leftover comment fragment doesn't vanish; it gets swept straight into the next rule's own selector text as ordinary characters. Once Chapter 5 builds real selector matching, a selector string like that will simply never match anything in any real document — no crash, no warning, just CSS that silently does nothing.

TWO SEPARATE COMMENTS IN A ROW IS A COMPLETELY DIFFERENT, PERFECTLY SAFE THING

`parse_css("/* one */ /* two */ p { color: red; }")` parses cleanly to a single rule with `selectors == ['p']` — exactly as expected. Two *consecutive* comments, each properly opened and closed on its own, is nothing like one comment written to *look* nested. The failure above is specifically about a `/*` appearing *before* the matching `*/` of an already-open comment — a structural mistake, not "too many comments."

The Rest of the Grammar

```
def parse_css(css):
    css = strip_comments(css)
    rules = []
    i = 0
    while i < len(css):
        open_brace = css.find('{', i)
        if open_brace == -1: break
        selector_text = css[i:open_brace].strip()
        close_brace = css.find('}', open_brace)
        if close_brace == -1: break
        decl_text = css[open_brace+1:close_brace]
        if selector_text:
            selectors = [s.strip() for s in selector_text.split(',') if s.strip()]
            rules.append(Rule(selectors, parse_declarations(decl_text)))
        i = close_brace + 1
    return Stylesheet(rules)
```

VERIFIED DIRECTLY — A TRAILING SEMICOLON ON THE LAST DECLARATION IS GENUINELY OPTIONAL

`parse_css("p { color: red; font-size: 16px; }")` and `parse_css("p { color: red; font-size: 16px }")` — the second missing its own final semicolon — produce **identical** declaration lists. Splitting on `;` naturally produces a trailing empty chunk when the semicolon *is* present, which the blank-chunk check simply skips; when it's absent, there's no empty chunk to skip in the first place. The same code handles both shapes without a special case for either.

VERIFIED DIRECTLY — A MULTI-TOKEN VALUE LIKE "10PX 20PX" SURVIVES INTACT AS ONE VALUE, NOT TWO

`parse_css("div { margin: 10px 20px; }")` produces exactly one `Declaration('margin', '10px 20px')`. Splitting only ever happens on `;` (between declarations) and the *first* `:` within each chunk (between property and value) — nothing inside `parse_declarations` ever splits on whitespace, so a value that's legitimately several space-separated tokens stays exactly that, one string.

VERIFIED DIRECTLY — ARBITRARY INDENTATION AND LINE BREAKS CHANGE NOTHING ABOUT THE RESULT

A messily formatted, multi-line version of the same two rules — extra blank lines, inconsistent indentation, spaces around the colon — produces byte-for-byte identical `Rule` objects to a tightly packed one-liner. Every split point in this chapter's own parser strips whitespace immediately afterward, so formatting is never load-bearing.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A flat list of raw selector strings on each `Rule`	Chapter 5's own selector matching — this chapter deliberately stops at "here is the text," leaving "does this text match this DOM node" as a separate, dedicated problem

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
CSS comments stripped once, up front, before any rule parsing begins	Mirrors Chapter 3's own `closetag` handling being defensive by construction — both chapters choose to make one part of the pipeline robust early, so nothing downstream has to re-solve the same problem
A corrupted selector from a non-nested comment produces no error anywhere	A direct parallel to this course's own Chapter 1 finding about regex and HTML — a parser that silently produces a plausible-looking wrong answer is a harder failure mode to catch than one that crashes loudly
Property names lowercased during parsing	Chapter 3's own tag-name lowercasing — the same "normalize once, at the source, so every later consumer can assume it's already done" discipline, applied to CSS instead of HTML

Hands-On Exercises

EXERCISE 1

Parse `"h1, /* comment */ h2 { color: red; }"` using this chapter's own `parse_css` — a comment sitting between two comma-separated selectors, not inside a declaration block. Verify the resulting rule's own `selectors` list, and explain why placing `strip_comments` *before* selector-splitting is what makes this work correctly.

View solution

EXERCISE 2

Parse `"p {}"` — a rule with a genuinely empty declaration block — using this chapter's own `parse_css`. Confirm this doesn't raise an error and determine exactly what the resulting `Rule`'s own `declarations` list contains, then explain which specific check inside `parse_declarations` is responsible for handling an empty block correctly.

View solution

EXERCISE 3

Parse `"/* one */ /* two */ p { color: red; }"` — two separate, properly closed comments in a row — and confirm the resulting selector is the clean `['p']`, not a corrupted string the way this chapter's own nested-looking example produced. Explain precisely, in terms of how `strip_comments`'s own scan position advances, why two consecutive comments never trigger the same failure a nested-looking one does.

View solution

CHAPTER 4 QUICK REFERENCE

- **Three structures:** `Declaration` (property/value), `Rule` (a list of selector strings + a list of declarations), `Stylesheet` (a list of rules)
- **Verified — the real quirk:** CSS comments don't nest; the first `*/` always ends the comment, no matter how many `/*` came before it
- **Verified:** a non-nested-looking comment silently corrupts the following selector, with no error raised anywhere in the pipeline
- **Verified:** a trailing semicolon on the last declaration is genuinely optional — the same code handles both shapes with no special case
- **Verified:** a multi-token value like `"10px 20px"` survives intact as one value, since splitting only ever happens on `;` and the first `:`
- **Verified:** arbitrary whitespace and indentation never change the parsed result
- **Next chapter:** Selectors & Selector Matching — deciding whether one of this chapter's own raw selector strings actually applies to a given DOM node

CHAPTER 5 OF 10

Selectors & Selector Matching

Building a Web Browser Engine: Parsing & the DOM

Chapter 5 · Selectors & Selector Matching

Chapter 4 left every `Rule` 's own selectors as raw strings — `"div"` , `".highlight"` , `"div p"` — real text, but not yet anything that can be checked against a DOM node. This chapter builds both halves: a real selector parser, and a matcher that decides, for one selector and one `Element` , whether they apply to each other at all.

Simple Selectors and the Selector They Combine Into

```
class SimpleSelector:
    def __init__(self, tag=None, id=None, classes=None):
        self.tag = tag              # None means "no tag constraint"
        self.id = id
        self.classes = classes or set()

class Selector:
    def __init__(self, parts):
        self.parts = parts          # ancestors left-to-right, TARGET last -- e.g. "div p" -> [div, p
```

A single space-separated piece of a selector — `div` , `.card` , `p#main.highlight` — is one `SimpleSelector` , scanned for a leading tag name followed by any number of `.class` and `#id` markers, in whatever order they appear.

VERIFIED DIRECTLY — TAG, CLASS, AND ID ALL MATCH CORRECTLY ON THEIR OWN

A `div` with `class="card"` matches the selector `div` and fails to match `p` . An element with `class="foo highlight bar"` matches `.highlight` — the class attribute is space-separated, and `.highlight` only needs to be *one* of the classes present, not the whole string. An element with `id="main"` matches `#main` and nothing else with a different id.

VERIFIED DIRECTLY — A COMPOUND SELECTOR REQUIRES EVERY PART TO HOLD AT ONCE

`div.card` matches a `div` that also carries `class="card"`, but a plain `div` with no class at all does **not** match. A full three-way compound, `p#main.highlight`, correctly matches a `<p id="main" class="highlight">` and correctly *rejects* a `<span id="main" class="highlight">` — identical id and class, wrong tag, no match.

A Real Gap, Found the Moment Descendant Selectors Are Needed

`div p` means "a `p` with a `div` *somewhere above it*" — answering that requires walking *upward* from a node. Chapter 3's own `Element` / `TextNode` / `CommentNode` classes only ever store `children`. Nothing before this chapter ever needed to go the other direction.

VERIFIED DIRECTLY — A NODE BUILT BY CHAPTER 3'S OWN PARSER HAS NO .PARENT ATTRIBUTE AT ALL

Accessing `.parent` on a freshly-parsed `Element` — one that never had anything extra done to it — raises `AttributeError: 'Element' object has no attribute 'parent'`. This isn't a bug in Chapter 3; a tree built purely for top-down parsing genuinely never needed an upward pointer until a selector like `div p` came along and asked for one.

```
def assign_parents(node, parent=None):
    node.parent = parent
    for child in node.children:
        assign_parents(child, node)
```

One walk over the tree, once, before any matching starts, and every node gains a real `.parent` reference — the same shape a real browser's own `Node.parentNode` takes.

Matching a Descendant Selector: Search Upward, Part by Part

```
def matches_selector(element, selector):
    if not matches_simple(element, selector.parts[-1]):
        return False                        # the TARGET itself has to match first
    current = element.parent
    for part in reversed(selector.parts[:-1]):
        found = False
        node = current
        while node is not None:
            if node.kind == 'element' and matches_simple(node, part):
                found = True; current = node.parent; break
            node = node.parent
        if not found:
            return False
    return True
```

VERIFIED DIRECTLY — A MATCHING ANCESTOR AND A GENUINELY ABSENT ONE GIVE THE CORRECT OPPOSITE ANSWERS

`div p` matches a `p` nested inside a `div` (through any number of levels), and correctly fails to match a `p` sitting at the top level with no `div` anywhere above it.

VERIFIED DIRECTLY — THE SEARCH IS GAP-TOLERANT, MATCHING REAL CSS BEHAVIOR EXACTLY

`div section p` matches a `p` nested as `div > section > article > p` — with a completely unrelated `article` sitting between `section` and `p`, an element the selector never mentions at all. Each part of the selector only has to find *some* matching ancestor, not the immediate parent — exactly how real CSS descendant combinators work. The same selector correctly fails to match a `p` sitting directly inside a `div` with no `section` anywhere in between.

An Honest Limitation: Nothing Here Enforces id Uniqueness

VERIFIED DIRECTLY — TWO ELEMENTS SHARING THE SAME ID BOTH "MATCH" IT

Two sibling elements — a `div` and a `span` — both carrying `id="dup"` both correctly match `#dup`, individually. Real HTML documents are supposed to never repeat an id, but nothing in this chapter's own parser or matcher checks for or enforces that — `matches_simple` only ever asks "does *this* element's own id attribute equal the one in the selector," with no awareness of any other element in the document at all. A genuinely malformed document with a duplicate id won't raise an error here; it'll just mean an id selector quietly matches more than one thing, which is exactly what a real browser does too when handed the same invalid markup.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Chapter 3's tree had no parent pointer until a descendant selector needed one	A direct parallel to Chapter 9's own `origin` / `base` finding in the compiler project — a data structure built correctly for its original purpose still needing an honest, later revision once a new consumer's own requirements arrive
Gap-tolerant ancestor search for multi-part descendant selectors	Chapter 6's own specificity and cascade — every rule whose selector matches at all becomes a real candidate for a given element, regardless of how deeply nested the match happened to be
`SimpleSelector` / `Selector` built here, from Chapter 4's own raw strings	Chapter 8's own style tree, which will need to ask "does this rule apply to this element" for every rule in the stylesheet, against every element in the DOM — the exact question this chapter's own `matches_selector` answers
id uniqueness left honestly unenforced	The same kind of stated scope boundary Chapter 3 drew around `<li>`'s own missing implied-closing rule — a real limitation, named directly rather than silently assumed away

Hands-On Exercises

EXERCISE 1

Build a tree shaped `div > section > article > p` (four levels, each a direct child of the one before it) and match it against the selector `"div section p"` using this chapter's own `matches_selector`. Confirm it matches despite `article` never appearing in the selector, then build a second tree — `div > p` directly, no `section` anywhere — and confirm the same selector correctly fails to match.

View solution

EXERCISE 2

Build two sibling elements — a `div` and a `span` — both with `id="dup"`, and match both against the selector `"#dup"` using this chapter's own `matches_selector`. Confirm both report a match, and explain specifically why `matches_simple` has no way to detect or reject this, even in principle, without being handed information about the rest of the document it currently never receives.

View solution

EXERCISE 3

Build a `<p id="main" class="highlight">` and match it against the compound selector `"p#main.highlight"` using this chapter's own `matches_selector`. Then build a `<span id="main" class="highlight">` — identical id and class, different tag — and match it against the same selector. Confirm the two results differ, and trace through `matches_simple` to identify exactly which check is responsible for the rejection.

View solution

CHAPTER 5 QUICK REFERENCE

- **Two structures:** `SimpleSelector` (tag/id/classes, one space-separated piece) and `Selector` (an ordered list of them — ancestors, target last)
- **Verified:** tag, class (against a multi-class attribute), id, and compound (tag+id+class together) selectors all match correctly on their own
- **Real gap found and fixed:** Chapter 3's own DOM tree had no `.parent` reference at all — `assign_parents()` adds it, once, before any matching begins
- **Verified:** descendant matching is gap-tolerant — an unrelated ancestor sitting between two selector parts doesn't break the match, exactly like real CSS
- **Verified — an honest limitation:** id uniqueness is never enforced; two elements sharing the same id both "match" an id selector
- **Next chapter:** Specificity & the Cascade — deciding which rule wins when more than one selector matches the same element

CHAPTER 6 OF 10

Specificity & the Cascade

Building a Web Browser Engine: Parsing & the DOM

Chapter 6 · Specificity & the Cascade

Chapter 5 answered "does this one selector match this one element" in isolation. A real stylesheet has dozens of rules, and a single element routinely matches several of them at once — sometimes agreeing, sometimes flatly contradicting each other. The **cascade** is the algorithm that decides, for every property an element might have styled, which one of the competing values actually wins.

Specificity: a Tuple, Not a Single Number

```
def specificity(selector):
    """(id_count, class_count, tag_count), summed across every SimpleSelector
    in the WHOLE selector -- every combinator part counts, not just the target."""
    id_count = 0
    class_count = 0
    tag_count = 0
    for part in selector.parts:
        if part.id is not None:
            id_count += 1
        class_count += len(part.classes)
        if part.tag is not None:
            tag_count += 1
    return (id_count, class_count, tag_count)
```

The tuple is compared *lexicographically* — Python's own default tuple comparison — meaning the first component that differs decides the whole comparison. Nothing here ever collapses the tuple into one combined number.

VERIFIED DIRECTLY — A SINGLE ID OUTRANKS ANY NUMBER OF CLASSES

`specificity('#main')` is `(1, 0, 0)`. `specificity('.a.b.c.d.e.f.g.h.i.j')` — ten classes on one element — is `(0, 10, 0)`. Compared as tuples, `(1, 0, 0) > (0, 10, 0)` is **True**: the id wins outright, no matter how many classes are stacked against it. If specificity were computed as a single summed number instead — the mistake newcomers to CSS make constantly — ten classes would very plausibly outscore one id.

VERIFIED DIRECTLY — TWO CLASSES BEAT ONE, AND COMPOUND SELECTORS SUM ACROSS THEIR WHOLE SELECTOR

`specificity('.a.b')` is `(0, 2, 0)`, genuinely higher than `specificity('.a')`'s `(0, 1, 0)`. And a descendant selector like `div.card p` sums across *every* part, not just the target — `specificity('div.card p')` comes out to `(0, 1, 2)`: one class from `.card`, two tags from `div` and `p` combined.

The Cascade: Merging Per Property, Not Per Rule

```
def cascade(element, stylesheet):
    """Returns a dict of property -> value: the final computed declarations
    for this one element, with conflicts resolved PER PROPERTY."""
    matches = []
    for order, rule in enumerate(stylesheet.rules):
        for sel_text in rule.selectors:
            sel = parse_selector(sel_text)
            if matches_selector(element, sel):
                matches.append((specificity(sel), order, rule.declarations))
                break

    # weakest first, so later (winning) values overwrite in the dict
    matches.sort(key=lambda m: (m[0], m[1]))

    result = {}
    for spec, order, decls in matches:
        for decl in decls:
            result[decl.name] = decl.value
    return result
```

Sorting weakest-to-strongest and then letting each matching rule's declarations overwrite the running `result` dict, one property at a time, is what makes this per-property rather than per-rule. A rule that loses the cascade on one property can still be the only rule that sets some *other* property at all — and that other value has to survive.

VERIFIED DIRECTLY — SPECIFICITY BEATS SOURCE ORDER, AND LOSING ON ONE PROPERTY DOESN'T ERASE THE REST

A class rule `.highlight { color: blue; font-size: 12px; }` is written *first*. An id rule `#main { color: red; }` is written *second*, on the same element. The id rule's higher specificity wins the `color` conflict — final `color` is `red` — but `font-size` was never contested at all, and survives from the class rule untouched: `{'color': 'red', 'font-size': '12px'}`.

VERIFIED DIRECTLY — A NAIVE "WINNING RULE REPLACES EVERYTHING" APPROACH SILENTLY LOSES DATA

Taking only the single highest-specificity rule's own declarations wholesale — a plausible-looking shortcut — produces `{'color': 'red'}` for the exact same input, with `font-size` gone entirely. The correct, per-property merge keeps it. This is exactly the kind of bug that would only show up on a real page where two rules divide responsibility for different properties on the same element — extremely common in practice.

VERIFIED DIRECTLY — EQUAL SPECIFICITY FALLS BACK TO SOURCE ORDER, AND IT'S GENUINELY ORDER-SENSITIVE

Two bare `p` rules — identical specificity, `(0,0,1)` each — setting `color: green` and `color: purple` respectively. With `green` first and `purple` second in the stylesheet, the final color is `purple`. Swapping which one appears later flips the result to `green`. Source order is only ever consulted once specificity is *exactly* tied — it never overrides a genuine specificity difference.

A Real Bug: Comma-Separated Selectors in One Rule

Chapter 4's own `Rule.selectors` is a list — one rule like `h1, .warning { color: blue; }` produces a single `Rule` whose `selectors` field is `['h1', '.warning']`, sharing one set of declarations. The `cascade()` above loops over that list and `break`s the moment *any* one of them matches — which quietly picks whichever selector happens to be listed first, not necessarily the one with the highest specificity.

VERIFIED DIRECTLY — THE BUG REALLY DOES FLIP THE WINNER

An `<h1 class="warning">`. One rule, `['h1', '.warning'] { color: blue; }`, written *first*. A second rule, plain `h1 { color: red; }`, written *after* it. Because `'h1'` is listed first in the comma rule and matches immediately, `cascade()` records its specificity as `(0,0,1)` — the tag-only figure — even though the same rule also matches via `.warning`, specificity `(0,1,0)`, genuinely higher. That tuple tie against the second rule's own `(0,0,1)` gets broken by source order, and the *later* rule wins: **red**. Real CSS treats a comma list as though each selector were its own separate rule sharing the same declarations — the first rule should win outright on its `.warning` specificity alone, **blue**, regardless of source order.

The fix: for each rule, check *every* one of its own selectors that matches, and keep the highest specificity among them — not just the first one found.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Specificity as a compared tuple, never a summed number	Chapter 5's own compound-selector matching — the same "every part has to hold together" discipline, applied here to scoring rather than matching
Per-property merging across every matching rule	Chapter 8's own style tree, which needs a complete, final property set for *every* DOM node — exactly what `cascade()` produces for one element at a time

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A comma-list rule needing its own highest-matching specificity	Chapter 5's own honest id-uniqueness gap — another case where a real, unglamorous edge case in how CSS selectors combine only surfaces once you actually test it, not when you reason about the "normal" case alone

Hands-On Exercises

EXERCISE 1

Build one element that matches three rules at once through a tag selector, a class selector, and an id selector — all setting the same property. Run them through `cascade()` once with the id rule written last in the stylesheet, and again with the id rule written *first*. Confirm the id rule wins both times, and explain why source order never gets a chance to matter here.

View solution

EXERCISE 2

Compute `specificity()` for the descendant selector `"div.card p"` and for the plain id selector `"#x"`. Compare the two tuples directly and confirm which one wins, even though the first selector combines a tag, a class, *and* a second tag across two separate parts of the selector.

View solution

EXERCISE 3

Reproduce this chapter's own comma-selector bug directly: an `<h1 class="warning">`, a rule `['h1', '.warning']` declaring blue written first, and a plain `h1` rule declaring red written second. Confirm the buggy `cascade()` above really does return red. Then write a fixed version that computes, for each rule, the *highest* specificity among every one of its own selectors that matches — and confirm it now correctly returns blue.

View solution

CHAPTER 6 QUICK REFERENCE

- **Specificity:** an `(id_count, class_count, tag_count)` tuple, compared lexicographically — never summed into a single score
- **Verified:** one id always outranks any number of classes; two classes always outrank one, regardless of source order
- **Cascade:** merges declarations **per property** across every matching rule — a rule that loses on one property can still be the only source of another
- **Verified:** a naive "winning rule replaces everything" shortcut silently drops properties a losing rule was the only source for

- **Tie-break:** source order only ever matters once specificity is genuinely equal — later rule wins a true tie, but never overrides a real specificity difference
- **Real bug found and fixed:** a comma-separated selector list inside one rule needs its *highest*-matching specificity, not just whichever selector happens to be listed first
- **Next chapter:** Inheritance & Computed Values — what happens to a property no rule ever set at all

CHAPTER 7 OF 10

Inheritance & Computed Values

Building a Web Browser Engine: Parsing & the DOM

Chapter 7 · Inheritance & Computed Values

Chapter 6's `cascade()` only ever returns properties that some rule actually set for that *specific* element. Most properties on most elements were never set by anything at all — and yet a real browser still shows a definite, specific value for every one of them. This chapter closes that gap: for every property, on every element, there has to be one final, unambiguous **computed value**, whether a rule mentioned it or not.

Two Fates for an Unset Property

CSS splits its properties into two groups. A fixed set of mostly text-related properties **inherit** — if nothing sets them directly, they take whatever value the parent element ended up with. Everything else falls back to a real **initial value** instead, completely independent of any ancestor.

```
INHERITED_PROPERTIES = frozenset({
    'color', 'font-family', 'font-size', 'font-weight', 'font-style',
    'line-height', 'text-align', 'visibility', 'white-space', 'letter-spacing',
})

INITIAL_VALUES = {
    'color': 'black',           # inheritable -- this is only the FALLBACK if there's no ancestor eit
    'font-family': 'serif',
    'font-size': '16px',
    'display': 'inline',       # NOT inheritable
    'margin': '0',             # NOT inheritable
    'padding': '0',            # NOT inheritable
    'background-color': 'transparent',
    # ... and more, in the real implementation
}
```

This isn't arbitrary — it's why setting `color` once on `<body>` reliably colors every piece of text on a whole page, while setting `margin` on a container never leaks that same margin onto everything nested inside it. Box-model properties staying put on the element that set them is exactly what makes layout predictable at all.

A Naive First Attempt — and a Real Bug It Has

```
def compute_style_naive(element, stylesheet):
    own = cascade(element, stylesheet)
    result = dict(INITIAL_VALUES)
    if element.parent is not None and element.parent.kind == 'element':
        parent_own = cascade(element.parent, stylesheet)   # <- only the parent's OWN rules
        for prop in INHERITED_PROPERTIES:
            if prop in parent_own:
                result[prop] = parent_own[prop]
    result.update(own)
    return result
```

Reasonable-looking: borrow inheritable properties straight from the parent's own `cascade()` result. It even passes an obvious first test — a direct parent/child pair where the parent has a color rule.

VERIFIED DIRECTLY — IT SILENTLY BREAKS TWO GENERATIONS UP

A grandparent with `id="g"` setting `color: red`. A parent with *no rule of its own*. A child with no rule either. `compute_style_naive(child, sheet)` returns `black` — the initial default — not red. The parent's own direct `cascade()` result genuinely has no `color` entry at all (no rule matches the parent directly), so nothing gets borrowed from it, and the child never even looks past its immediate parent to find where the real value actually lives. A real browser shows this exact structure in red.

The Fix: Inherit From the Parent's Already-Computed Style

```
def compute_style(element, stylesheet, parent_computed=None):
    own = cascade(element, stylesheet)
    base = dict(INITIAL_VALUES)
    if parent_computed is not None:
        for prop in INHERITED_PROPERTIES:
            if prop in parent_computed:
                base[prop] = parent_computed[prop]
    base.update(own)
    return base

def compute_style_tree(root, stylesheet, parent_computed=None):
    """Walks the DOM TOP-DOWN -- each node's computed style becomes the
    parent_computed passed into every one of ITS OWN children."""
    if root.kind != 'element':
        return
    root.computed_style = compute_style(root, stylesheet, parent_computed)
    for child in root.children:
        compute_style_tree(child, stylesheet, root.computed_style)
```

The critical difference: `parent_computed` is the parent's own *fully resolved* style — which already includes whatever *it* inherited from further up — not the parent's raw, direct `cascade()` result. That's only obtainable by walking the tree top-down and carrying each computed style downward as the recursion goes.

VERIFIED DIRECTLY — THE FIX RESOLVES THE EXACT GRANDPARENT CASE THAT BROKE

Same three-generation tree. `compute_style_tree(grandparent, sheet)` gives grandparent `color: red` (its own rule), parent `color: red` (inherited from grandparent's computed style), and child `color: red` (inherited from parent's *own* computed style, which already includes what parent itself inherited). The value propagates transitively, exactly as far down the tree as nothing overrides it.

VERIFIED DIRECTLY — NON-INHERITED PROPERTIES GENUINELY STAY PUT

A parent with `margin: 20px; color: green;`. The child inherits `color: green`, but its `margin` comes out as the initial `0`, completely unaffected by the parent's own 20px. Two properties, set together on the same rule, propagating in two entirely different ways.

VERIFIED DIRECTLY — AN ELEMENT'S OWN RULE STILL OVERRIDES WHATEVER IT WOULD HAVE INHERITED

A parent set to `color: red`, a child with its own separate rule setting `color: blue`. The child's computed color is `blue` — `base.update(own)` runs *after* the inherited value is filled in, so a real matching rule on the element itself always has the final say.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Top-down tree walk carrying each parent's computed style downward	Chapter 5's own `assign_parents()` — the same DOM tree, now walked in the opposite direction its structure was originally built to support, exactly as the descendant-selector matcher already needed to
A property's own final computed value being unambiguous for every element	Chapter 8's own style tree, which pairs every DOM node with exactly this — its own complete, resolved set of computed values, the direct input the layout engine in Course 2 will consume
Inherit-from-direct-cascade being a genuine, easy-to-miss bug	Chapter 6's own comma-selector bug — another case where a single, non-recursive parent lookup looks completely correct until tested against a tree more than one level deep

Hands-On Exercises

EXERCISE 1

Build a single element with no parent at all and an empty stylesheet (no rules whatsoever). Run it through `compute_style_tree()` and confirm its resulting `computed_style` dict is exactly equal to `INITIAL_VALUES` — nothing more, nothing less.

View solution

EXERCISE 2

Build a three-generation chain (grandparent → mid → leaf). Give the grandparent a rule setting `color`, and give the *middle* element a separate rule setting a different inheritable property, `font-weight` — leave the leaf with no rule of its own. Confirm the leaf's computed style correctly picks up *both* inherited properties, each one traced back to a different ancestor.

View solution

EXERCISE 3

Reproduce this chapter's own `compute_style_naive()` bug directly: a grandparent with a color rule, a parent with no rule, and a child with no rule. Confirm the naive version really does return the initial default instead of the inherited color. Then run the same three-generation tree through the fixed `compute_style_tree()` and confirm it returns the correct, inherited value instead.

View solution

CHAPTER 7 QUICK REFERENCE

- **Two fates:** a fixed set of text-related properties (`color`, `font-*`, `line-height`, ...) inherit from the parent; everything else falls back to a real initial value
- **Real bug found and fixed:** inheriting from the parent's own direct `cascade()` result breaks two generations up — the fix walks the tree top-down, inheriting from the parent's own already-fully-resolved computed style instead
- **Verified:** inheritance is transitive — a value can propagate through any number of ancestors that never set it themselves, as long as one of them inherited it in turn
- **Verified:** non-inherited properties (like `margin`) never leak from parent to child, even when set on the same rule as an inheritable property that does
- **Verified:** an element's own matching rule always overrides whatever it would otherwise have inherited
- **Next chapter:** Building the Style Tree — combining the DOM, the cascade, and inheritance into one single annotated tree

CHAPTER 8 OF 10

Building the Style Tree

Building a Web Browser Engine: Parsing & the DOM

Chapter 8 · Building the Style Tree

Every piece this course has built so far answers one question in isolation: does a selector match (Chapter 5), which rule wins (Chapter 6), what's the final value for one property on one element (Chapter 7). None of them, on their own, hand Course 2's layout engine what it actually needs — one single tree, where every node already carries its own complete, resolved style. That's the **style tree**, and building it is this chapter's only job.

The StyledNode: a DOM Node Paired With Its Own Computed Style

```
class StyledNode:
    def __init__(self, node, style, children):
        self.node = node          # the underlying DOM node -- Element or TextNode
        self.style = style        # its own final, resolved computed-style dict
        self.children = children  # a list of StyledNode -- NOT raw DOM children
```

A second tree, shaped roughly like the DOM tree underneath it, but genuinely distinct — some DOM nodes won't have a matching `StyledNode` at all, as this chapter's own testing is about to show.

A Naive First Attempt — and the Bug Testing It Reveals

```
def build_style_tree_naive(dom_node, stylesheet, parent_computed=None):
    if dom_node.kind != 'element':
        return None
    style = compute_style(dom_node, stylesheet, parent_computed)
    children = []
    for child in dom_node.children:
        if child.kind == 'element':
            styled_child = build_style_tree_naive(child, stylesheet, style)
            if styled_child is not None:
                children.append(styled_child)
    return StyledNode(dom_node, style, children)
```

Reasonable-looking: only elements get a computed style at all, so only recurse into element children. It even builds a tree that *looks* structurally fine at a glance.

VERIFIED DIRECTLY — IT SILENTLY DROPS EVERY PIECE OF ACTUAL TEXT

`<p id="greet">Hello <!-- a note --> <b>world</b></p>`, run through `build_style_tree_naive`. Collecting all text out of the resulting tree gives `[]` — completely empty. `<b>` survives (it's an element), but the text `"Hello "` sitting directly in `<p>` is gone, and so is `"world"` inside `<b>`, since the recursive call on `<b>`'s own children applies the exact same element-only filter. A style tree with no text in it at all can't render a single visible word — for most real pages, text is the overwhelming majority of what actually gets painted to the screen.

The Fix: Three Cases, Handled Explicitly

```
def build_style_tree(dom_node, stylesheet, parent_computed=None):
    if dom_node.kind == 'comment':
        return None                    # comments are never rendered -- excluded outright

    if dom_node.kind == 'text':
        # a text node has no selectors of ITS OWN that could ever match it --
        # selectors only ever match elements (Chapter 5) -- so it simply takes
        # on whatever its containing element already resolved to
        style = dict(parent_computed) if parent_computed is not None else dict(INITIAL_VALUES)
        return StyledNode(dom_node, style, [])

    style = compute_style(dom_node, stylesheet, parent_computed)
    if style.get('display') == 'none':
        return None                    # removes this element AND every descendant, in one step

    children = []
    for child in dom_node.children:
        styled_child = build_style_tree(child, stylesheet, style)
        if styled_child is not None:
            children.append(styled_child)
    return StyledNode(dom_node, style, children)
```

VERIFIED DIRECTLY — TEXT IS PRESERVED, CORRECTLY NESTED, WITH THE RIGHT INHERITED STYLE

The same `<p id="greet">Hello <!-- a note --> <b>world</b></p>`, this time through the fixed `build_style_tree`. Collected text comes back as `['Hello ', 'world']` — both real pieces survive, in the right order, correctly nested inside `<b>`'s own `StyledNode`. The comment is gone from the tree entirely, not just visually hidden. And crucially: both text nodes' own `style` dict shows `color: blue` — inherited from `#greet`'s own rule — confirming a text node genuinely takes on its *containing element's* resolved style rather than trying (and failing) to match any selector of its own.

VERIFIED DIRECTLY — DISPLAY:NONE REMOVES AN ELEMENT AND ITS ENTIRE SUBTREE, NOT JUST ITSELF

A `<div>` containing a visible `<span>` and a hidden `<div id="hid">` that itself contains a nested `<span>`. With `#hid { display: none; }`, the container's own style tree ends up with exactly **one** child — only the visible sibling. The hidden `<div>` and its own nested `<span>` are both gone in a single step, because `build_style_tree` returns `None` for `#hid` before ever recursing into its own children at all — a completely different outcome from merely styling `#hid` itself invisible while still building boxes for whatever's nested inside it.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
One combined tree, every node carrying its own final computed style	The exact structure Course 2's own layout engine consumes first — a `StyledNode` tree is what gets converted into a layout tree, not the raw DOM tree and not a separate cascade result per element
Recursing only into element children silently drops all text	Chapter 3's own `TextNode`/`CommentNode` classes existing as genuinely distinct `kind`s in the first place — this chapter is the first to actually need that distinction to matter for something beyond just "don't confuse a comment for a tag"
display:none removing a whole subtree in one step	Course 2's own upcoming distinction between "not rendered at all" (this chapter's `display: none`) and "rendered but invisible" (a future `visibility: hidden`-style property, which still occupies layout space) — a real, separate concept this course deliberately doesn't build

Hands-On Exercises

EXERCISE 1

Build a `<div>` whose only child is a single `CommentNode` — no text, no other elements. Run it through `build_style_tree()` and confirm the resulting `StyledNode`'s own `children` list is empty, rather than the function crashing or producing a placeholder node standing in for the comment.

View solution

EXERCISE 2

Call `build_style_tree()` directly on an element whose own matching rule sets `display: none` — not as someone else's child, but as the root node passed in. Confirm the function returns `None` outright, and explain why this has to be the same result as when that element is reached deeper inside a larger tree, rather than a special case only handled at the point where a parent decides whether to keep a child.

View solution

EXERCISE 3

Reproduce this chapter's own naive-vs-fixed comparison directly: build `<p id="greet">Hello <!-- a note --> <b>world</b></p>`, run it through both `build_style_tree_naive()` and `build_style_tree()`, and collect all rendered text out of each resulting tree. Confirm the naive version returns an empty list while the fixed version returns both real text pieces in order, and identify the exact line in `build_style_tree_naive` responsible for the difference.

View solution

CHAPTER 8 QUICK REFERENCE

- **StyledNode:** a DOM node (Element or TextNode) paired with its own final computed style and a list of StyledNode children
- **Real bug found and fixed:** recursing only into element children silently drops every text node — since text is most of what a real page actually renders, this is a severe, not cosmetic, bug
- **Three cases, handled explicitly:** comments are excluded entirely; text nodes are included with their containing element's own inherited style, never their own rule match; elements get a real computed style via Chapter 6/7's own machinery
- **Verified:** `display: none` removes an element and its entire subtree from the style tree in one step — not just that one element while still rendering its children
- **Next chapter:** Default (User-Agent) Stylesheets — where values like `display: none` on `<head>` or block-level defaults for `<div>` actually come from, before any author stylesheet is even applied

CHAPTER 9 OF 10

Default (User-Agent) Stylesheets & Real-World Fidelity

Building a Web Browser Engine: Parsing & the DOM

Chapter 9 · Default (User-Agent) Stylesheets & Real-World Fidelity

Open a real, completely un-styled HTML page — no `<link>`, no `<style>`, nothing — and it still doesn't look like a flat wall of text. Headings are big and bold. Paragraphs have space between them. `<div>` stacks vertically; `<span>` doesn't. None of that is magic, and none of it is hardcoded tag-by-tag logic somewhere deep in the engine. It's CSS — a real stylesheet, built into the browser itself, applied silently before a single author rule is ever considered.

The Bug Hiding in Plain Sight Since Chapter 7

VERIFIED DIRECTLY — EVERY ELEMENT HAS BEEN DEFAULTING TO DISPLAY:INLINE THIS WHOLE COURSE

Chapter 7's own `INITIAL_VALUES` sets `'display': 'inline'`. Run a completely bare `<div>`, with no stylesheet at all, through `compute_style()`: the computed `display` comes back `inline`. Every single test in Chapters 6 through 8 that happened to care about layout-relevant properties simply never triggered this — but it's genuinely wrong. A real `<div>`, in every real browser, on a page with zero CSS anywhere, renders as a block.

CSS's own real initial value for `display` really is `inline` — that part of Chapter 7 was correct. What was missing is the second half of the picture: *before* any author stylesheet runs, the browser applies its own default stylesheet first, and that's where `<div>` actually picks up `display: block`.

A Real (If Simplified) User-Agent Stylesheet

```
UA_STYLESHEET = Stylesheet([
    Rule(['div', 'p', 'ul', 'li', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6'],
         [Declaration('display', 'block')]),
    Rule(['span', 'b', 'i', 'a'],
         [Declaration('display', 'inline')]),
    Rule(['h1'], [Declaration('font-size', '2em'), Declaration('font-weight', 'bold')]),
    Rule(['b'],  [Declaration('font-weight', 'bold')]),
    Rule(['p'],  [Declaration('margin', '16px 0')]),
])
```

It's built out of exactly the same `Rule` / `Stylesheet` shapes Chapter 4 defined — a UA stylesheet is a real stylesheet, not a special mechanism of its own. **This particular set of rules is deliberately illustrative**, not a verbatim copy of any real browser's own (much larger) default stylesheet — but the underlying idea it demonstrates is exactly how real browsers work.

VERIFIED DIRECTLY — APPLYING ONLY THE UA STYLESHEET ALREADY FIXES THE DISPLAY BUG

A raw `<div>`, run through `compute_style()` with *only* `UA_STYLESHEET` and no author rules at all, computes `display: block`. A raw `<span>`, same treatment, computes `display: inline`. Nothing else in this engine knows the difference between these two tags at all — the distinction lives entirely inside this one stylesheet.

A Real Bug: Specificity Alone Isn't Enough Once Two Origins Exist

The obvious next step — just concatenate `UA_STYLESHEET.rules` and the author's own rules into one combined `Stylesheet`, and run Chapter 6's own `cascade()` over it — looks reasonable. It even usually works, since most author rules naturally end up with equal or higher specificity than a simple UA default.

VERIFIED DIRECTLY — A HIGHER-SPECIFICITY UA RULE BEATS A LOWER-SPECIFICITY AUTHOR RULE THAT SHOULD HAVE WON

A UA rule `p.intro { margin: 40px; }` — specificity `(0,1,1)` — against an author's own reset, `p { margin: 0; }` — specificity `(0,0,1)`, genuinely lower. Concatenating both into one stylesheet and cascading by specificity alone: the UA rule wins, margin stays `40px`. Real CSS never allows this — an author's rule *always* beats a UA rule, regardless of which one happens to have higher specificity. Specificity is only ever compared *within* the same origin, never across origins.

```
def cascade_with_origin(element, ua_sheet, author_sheet):
    """Origin checked FIRST -- any matching author rule beats any matching UA
    rule, regardless of specificity. Specificity (then source order) only
    ever breaks a tie WITHIN the same origin."""
    matches = []   # (origin, specificity, order, declarations) -- 0=UA, 1=author
    order_counter = 0
    for origin, sheet in ((0, ua_sheet), (1, author_sheet)):
        for rule in sheet.rules:
            best_spec = None
            for sel_text in rule.selectors:
                sel = parse_selector(sel_text)
                if matches_selector(element, sel):
                    s = specificity(sel)
                    if best_spec is None or s > best_spec:
                        best_spec = s
            if best_spec is not None:
                matches.append((origin, best_spec, order_counter, rule.declarations))
            order_counter += 1

    matches.sort(key=lambda m: (m[0], m[1], m[2]))
    result = {}
    for origin, spec, order, decls in matches:
        for decl in decls:
            result[decl.name] = decl.value
    return result
```

VERIFIED DIRECTLY — THE FIX RESOLVES THE EXACT CASE THAT BROKE, AND HOLDS UP UNDER A GENUINE TIE TOO

Same two rules through `cascade_with_origin()`: final margin is `0` — the author's reset wins outright. And separately: an author rule with the exact *same* specificity as a competing UA rule (both plain `h1` selectors, `(0,0,1)` each) still resolves in the author's favor — origin is checked before specificity is ever consulted, so even a genuine specificity tie can't let a UA rule sneak through.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A real, built-in stylesheet supplying every tag's own default appearance	Chapter 8's own style tree — `display: none` was already demonstrated there via an author rule; this chapter shows `display: block` / `inline` normally comes from the UA sheet instead, for every element, on every page, with or without any author CSS at all

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Origin outranking specificity, checked as its own separate tier	Chapter 6's own cascade sort key, now needing a THIRD component ahead of the two it already had — `(origin, specificity, order)` instead of just `(specificity, order)` — the same "add one more tier to the same sorting idea" pattern that specificity itself was to source order
Chapter 7's INITIAL_VALUES being individually correct but incomplete without this chapter	A direct parallel to Chapter 5's own honest id-uniqueness gap and Chapter 3's `<li>` gap — a piece that was genuinely right for what it covered, revealed as incomplete only once a later chapter tests the case it never handled

Hands-On Exercises

EXERCISE 1

Run a raw `<b>` element through `cascade_with_origin()` using `UA_STYLESHEET` and a completely empty author stylesheet. Confirm the computed `font-weight` is still `bold`, and explain why origin-aware cascading produces the exact same result as plain cascading whenever only one origin has a matching rule at all.

View solution

EXERCISE 2

Build an `<h1>` element. Give the UA stylesheet its usual `h1 { font-weight: bold; }` rule, and give the author stylesheet its own `h1 { font-weight: normal; }` rule — the exact same specificity, `(0,0,1)`, on both sides. Run it through `cascade_with_origin()` and confirm the author's rule still wins despite the tie, then trace through the sort key to explain exactly which comparison decides it.

View solution

EXERCISE 3

Reproduce this chapter's own margin bug directly: a UA rule `p.intro { margin: 40px; }` against an author rule `p { margin: 0; }`, on a `<p class="intro">` element. Run it through both `cascade_naive_combined()` (the buggy concatenate-and-cascade approach) and `cascade_with_origin()`. Confirm the two functions disagree, and identify exactly which comparison inside `cascade_naive_combined()` is responsible for letting the UA rule win.

View solution

CHAPTER 9 QUICK REFERENCE

- **User-agent stylesheet:** a real, ordinary `Stylesheet` the browser applies before any author CSS — this is where `<div>`'s block-level default and `<span>`'s inline default actually come from, not a hardcoded tag check

- **Verified:** Chapter 7's `INITIAL_VALUES` alone gives every element `display: inline` — correct as CSS's own true initial value, but genuinely wrong for most real elements without a UA stylesheet supplying the rest
- **Real bug found and fixed:** concatenating UA and author rules into one stylesheet and cascading by specificity alone lets a higher-specificity UA rule beat a lower-specificity author rule — real CSS checks origin (author always outranks UA) before specificity at all
- **The fix:** a three-part sort key, `(origin, specificity, order)` — origin decides first; specificity and source order only ever break a tie within the same origin
- **Verified:** even a genuine specificity tie between a UA rule and an author rule resolves in the author's favor, since origin is checked before specificity is ever compared
- **Next chapter:** Capstone — parsing a real HTML+CSS document, with the UA stylesheet applied, all the way into a fully-resolved style tree

CHAPTER 10 OF 10

Capstone — Parsing a Real HTML+CSS Document into a Style Tree

Building a Web Browser Engine: Parsing & the DOM

Chapter 10 · Capstone — Parsing a Real HTML+CSS Document into a Style Tree

Every chapter so far has tested one piece in isolation, against hand-built objects standing in for whatever the previous chapter was supposed to hand it. This chapter does something none of them did: run a real HTML document and a real stylesheet through the *actual* tokenizer, parser, CSS parser, selector matcher, cascade, and style-tree builder — the same functions, wired together, end to end — and check the result against what CSS's own rules predict by hand.

A Real Bug, Found Only by Actually Integrating Everything

Wiring the real `parse_css()` output directly into the real `cascade()`-style code surfaced something no single chapter's own tests ever could.

VERIFIED DIRECTLY — A GENUINE ATTRIBUTEERROR, THE MOMENT THE TWO HALVES ACTUALLY MEET

Chapter 4's own `Declaration` class stores its field as `.property`: `Declaration(property, value)`. Every cascade-related chapter since Chapter 6 was written and tested against its own local stand-in for `Declaration` — and every one of those stand-ins, independently, used `.name` instead. Handing a real `Declaration` built by this chapter's own `parse_css()` into any of those earlier chapters' own final loops — `result[decl.name] = decl.value` — raises a genuine `AttributeError: 'Declaration' object has no attribute 'name'`. Nothing in Chapters 6 through 9 was ever wrong *in isolation* — every one of their own tests, built against their own consistent local stand-ins, passed cleanly. The mismatch was only ever between chapters, and only a real end-to-end run could have caught it.

The fix is a one-word correction, applied everywhere the cascade logic reads a declaration's own property name: `decl.property`, matching Chapter 4's real, original field — not `decl.name`.

The Real Document

```
<!-- html_source -->
<div id="page">
<h1>Welcome</h1>
<p class="intro">This is <b>bold</b> and normal text.</p>
<ul>
<li>One</li>
<li class="hidden">Two</li>
</ul>
</div>

/* css_source */
#page { color: navy; }
.intro { font-size: 20px; }
.hidden { display: none; }
p { margin: 24px 0; }
```

Run through `tokenize()` → `parse()` → `assign_parents()` → `parse_css()` → `build_style_tree_full()` (the Chapter 8 style-tree builder, upgraded to call Chapter 9's `cascade_with_origin()` against both this author stylesheet and the Chapter 9 `UA_STYLESHEET` together) — the real pipeline, nothing hand-constructed.

Verifying the Result, Node by Node

VERIFIED DIRECTLY — DISPLAY AND COLOR RESOLVE CORRECTLY AT THE VERY TOP OF THE TREE

`<div id="page">`: `display: block` (Chapter 9's UA rule — nothing in the author stylesheet even mentions `display` on this element) and `color: navy` (the author's own `#page` rule).

VERIFIED DIRECTLY — H1 COMBINES A UA DEFAULT AND AN INHERITED VALUE FROM TWO ENTIRELY DIFFERENT SOURCES

`<h1>`: `display: block`, `font-weight: bold`, `font-size: 2em` — all three from the UA stylesheet, since the author stylesheet has no `h1` rule at all — plus `color: navy`, inherited straight from `#page` (Chapter 7's own mechanism), despite no rule anywhere directly targeting `h1`'s own color.

VERIFIED DIRECTLY — THE AUTHOR'S MARGIN RULE WINS OVER THE UA'S, AT EQUAL SPECIFICITY, PURELY ON ORIGIN

`<p class="intro">`: `margin: 24px 0` — the author's own `p` rule — not the UA stylesheet's `16px 0`. Both selectors are the identical bare tag `p`, specificity `(0,0,1)` on both sides — a genuine tie. Chapter 9's origin tier decides it: author always outranks UA, even without needing higher specificity. `font-size: 20px` comes from the author's own `.intro` rule (nothing in the UA sheet touches `font-size` on `p` at all), and `color: navy` is inherited from `#page`, two levels up.

VERIFIED DIRECTLY — INHERITANCE CONTINUES CORRECTLY THROUGH A THIRD LEVEL, INTO AN ELEMENT WITH NO RULE OF ITS OWN

`<b>`, nested inside `p.intro`: `display: inline` and `font-weight: bold`, both from the UA stylesheet's own `b` rule — plus `font-size: 20px` and `color: navy`, *neither* set by any rule matching `<b>` directly, both inherited transitively from `p.intro`, which itself only has `font-size` from its own `.intro` rule and `color` inherited yet again from `#page`.

VERIFIED DIRECTLY — EVERY TEXT NODE CARRIES ITS CONTAINING ELEMENT'S OWN RESOLVED STYLE, EXACTLY AS CHAPTER 8 ESTABLISHED

Collecting all text under `p.intro` gives `['This is ', 'bold', ' and normal text.']` — all three pieces present, correctly ordered, correctly nested. The `"bold"` text node's own style shows `color: navy`, `font-size: 20px`, *and* `font-weight: bold` — inheriting the first two from `<b>`'s own inherited values, and the third from `<b>`'s own UA-set value — a text node never distinguishes between a value its container inherited versus one its container set directly; it just takes the whole finished computed style as one unit.

VERIFIED DIRECTLY — DISPLAY:NONE REMOVES LI.HIDDEN AND ITS OWN TEXT ENTIRELY, THREE LEVELS DOWN

`<ul>`'s style-tree children include real whitespace-only text nodes from the source's own line breaks between tags (a genuine, if invisible, style-tree entry — Course 2's own inline-layout chapters are where whitespace like this actually gets collapsed or preserved for real). Filtering to *element* children specifically: exactly **one** survives — the first `<li>`, containing `"One"`. The second `<li class="hidden">`, matched by the author's own `.hidden { display: none; }` rule, is gone completely — itself and its own `"Two"` text child, removed together as one unit, exactly as Chapter 8 verified. The surviving `<li>`'s own `color` is `navy` — inherited through a real three-level chain, `#page → ul → li`, with neither intermediate element setting `color` itself.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A real integration bug, invisible to every individual chapter's own tests	The exact reason a capstone chapter exists at all in this course's own format — Chapters 2 through 9 were each genuinely correct in isolation; only running them together for the first time could ever have surfaced this
A fully-resolved style tree, built from real source text start to finish	Course 2, Building a Web Browser Engine: Layout & Rendering — this exact `StyledNode` tree, produced by this exact pipeline, is the direct input its own first chapter starts from
Whitespace-only text nodes surviving as real style-tree entries	Course 2's own inline-layout chapters, which are where a real browser engine actually decides what to do with whitespace between elements — collapsing it, preserving it, or treating it as insignificant depending on context

What This Course Doesn't Cover

Restating Chapter 1's own honest scope, now that every piece of it has actually been built: no JavaScript, no networking beyond an in-memory string, no images, no Flexbox or Grid, no forms, no real font rendering, and no real GPU/OS rendering surface. This course produces a fully-resolved style tree — the DOM, the cascade, inheritance, and a real (if simplified) user-agent stylesheet, all genuinely working together. It does not yet know how big anything is, where anything sits on a page, or what a single pixel looks like.

Where This Connects: On to Course 2

Course 2, **Building a Web Browser Engine: Layout & Rendering**, picks up exactly here — turning this `StyledNode` tree into a real layout tree with actual box dimensions (the CSS box model), resolving block and inline layout, measuring real text, and finally rasterizing the whole thing into an actual pixel buffer. Every chapter in that course assumes this one's own output as its starting input.

COURSE 1 COMPLETE — BUILDING A WEB BROWSER ENGINE: PARSING & THE DOM

- **Chapters 1-2:** why a browser engine matters, and a real, forgiving HTML tokenizer (void elements, the ignored-trailing-slash quirk, quoted/unquoted/boolean attributes, comment safety)
- **Chapters 3-4:** a real stack-based DOM parser (with the implied `<p>`-closing rule) and a real CSS tokenizer/parser (with the non-nesting-comment quirk)
- **Chapters 5-6:** real selector parsing and matching (with the added DOM parent-pointer fix), and a real cascade (specificity as a compared tuple, per-property merging, and the comma-selector specificity bug found and fixed)
- **Chapters 7-8:** real inheritance (with the parent's-raw-cascade bug found and fixed) and the real `StyledNode` style tree (with the dropped-text-nodes bug found and fixed)
- **Chapter 9:** a real user-agent stylesheet and origin-aware cascading (with the specificity-vs-origin bug found and fixed)
- **Chapter 10 (this chapter):** every piece wired together for the first time on a real document — surfacing and fixing one final, genuine integration bug (`Declaration.property` vs. `.name`) that no single chapter's own isolated tests could ever have caught
- **Next:** Building a Web Browser Engine: Layout & Rendering — turning this course's own finished style tree into real, measured, rasterized pixels