

Idiosyncrasies of the HTML parser

Table of Contents

[Preface](#)

[Chapter 1. Introduction](#)

[Chapter 2. The HTML syntax](#)

[Chapter 3. The HTML parser](#)

[Chapter 4. Scripting complications](#)

[Chapter 5. Serializing](#)

[Chapter 6. Security implications](#)

[Appendix A. Implementations](#)

[Appendix B. Conformance checkers](#)

[Appendix C. Microsyntaxes](#)

[Appendix D. Bibliography](#)

Preface



Intended audience

This is not an HTML beginner's book. The intended audience is web developers who want to gain a deeper understanding of how the HTML parser works, or the history and

rationale behind certain behaviors. Some prior knowledge of HTML and the DOM is assumed. If you are going to implement your own HTML parser (awesome!), then this book will hopefully be helpful, but please implement from the [HTML standard](#). If you contribute to a browser engine or to web standards (awesome!), then this book will hopefully be helpful. If nothing else, I hope it will at least be an interesting read.

Definition

Dictionary.com offers the following [definition](#) of *parse* in the context of computers:

to analyze (a string of characters) in order to associate groups of characters with the syntactic units of the underlying grammar.

The Wikipedia page for [Parsing](#) offers the following:

Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin *pars* (orationis), meaning part (of speech).

In the context of HTML, the *HTML parser* is responsible for the process of converting a stream of characters (the HTML markup) to a tree representation known as the Document Object Model (the DOM).

Scope

This book covers the history of HTML parsers, how to write syntactically correct HTML, how an HTML parser works, including error handling, what can be done with the parsed DOM representation, and how to serialize it back to a string. It also covers parsing of some HTML *microsyntaxes* (parsing of some attribute values), which are strictly speaking not part of the HTML parser, but a layer above. It further discusses implementations and conformance checkers.

Parsing of other languages, such as XML, JavaScript, JSX (React's HTML-like syntax), or CSS, is not covered in this book.

Practical application

Knowing exactly how the HTML parser works is not necessary to be a successful web developer. However, some things can be good to know, and having a deeper understanding makes it easier to reason about its behavior. It can also be good to know that you should usually pull in an HTML parser instead of writing a [regular expression to "parse" HTML](#).

The following is a non-exhaustive list of things that would be good for most web developers to understand about the HTML parser.

- **How `</script>` works.** "`</script>`" in a script block does not always close the script. This is discussed in the [Script states](#) section in [Chapter 3. The HTML parser](#).
- **Implied tags/omitted tags.** Some tags are optional, and some tags are implied without being optional. This explains why, for example, it's not possible to nest an `<ul>` in `<p>`. This is discussed in the [Implied tags](#) section in [Chapter 3. The HTML parser](#).
- **`document.body` being null.** Before the `<body>` has been parsed, `document.body` is null. See [Chapter 4. Scripting complications](#).
- **Scripting and styling.** Knowing what the DOM will look like helps with working with the DOM with script or writing selectors in CSS. This has some overlap with implied tags. For example, `<tbody>` is implied in `<table>` even if that tag is not present.
- **Writing correct HTML.** Knowing how the parser works may give you more confidence in how to write HTML. For example, a relatively common error is to use `"/>"` syntax on a non-void HTML element (`br` is a void element, `div` is not void), although that is not supported (it will be treated as a regular start tag, ignoring the slash). See [Chapter 2. The HTML syntax](#).
- **Security.** For example, cross-site scripting (XSS) attacks sometimes target holes in sanitizers. Such attacks may be prevented by using an HTML parser-based sanitizer. See [Chapter 6. Security implications](#).
- **Web compatibility.** The HTML parser specification is known to be compatible with HTML as it is used on the web. When Opera implemented the specified HTML parser, it eliminated [20% of its web compatibility bugs](#) (of any kind).

About the author

Simon started contributing to the WHATWG in 2005, worked at Opera Software between 2007 and 2017, then at Bocoup between 2017 and 2022, and is currently working at Mozilla as a Web Standards Engineer. He contributed to the design of some aspects of the HTML parser specification, such as how SVG in HTML works and finding a web-compatible way to tokenize `script` elements. He edited the specification for the `picture` element from 2014 onwards and is currently an editor of the WHATWG [HTML standard](#) and the WHATWG [Quirks Mode standard](#). He is [@zcorpan](#) on X and [@zcorpan@mastodon.social](#) in the fediverse.

Acknowledgements

Thanks to Mathias Bynens for suggesting the platypus for the front cover (I asked on X "If the HTML parser were an animal, what would it be?").

The [platypus sketch](#) on the front cover is from Wikipedia, by Hmich176, with the following licenses:

GNU Free Documentation License

Creative Commons Attribution-ShareAlike 3.0

The chapter images in this book were generated using OpenAI's ChatGPT, as allowed by their [terms of use](#).

The font used on the front cover is [Archistico](#), by [Archistico](#), and has the following license:

You can use the font for commercial purposes, but not sell it! Every once in a comment on the page would be nice.

This book contains quotes from the [WHATWG HTML Standard](#) which has the following copyright and license:

Copyright © 2018 WHATWG (Apple, Google, Mozilla, Microsoft). This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

Thanks to Ian Hickson and Henri Sivonen for letting me quote their emails, blog posts, etc. in this book.

Thanks to Ingvar Stepanyan for letting me use some of his X quizzes in this book.

Thanks to Mike Smith for providing a raw log from a validator instance for the [Most common errors](#) section in [Appendix B. Conformance checkers](#).

Thanks to Marcos Caceres, Sam Sneddon, Taylor Hunt, Mike Smith, Anne van Kesteren, Marie Staver, Ian Hickson, Mathias Bynens, Henri Sivonen, and Philip Jägenstedt for reviewing this book.

Thanks to Jens Oliver Meiert for contributing fixes for this book.

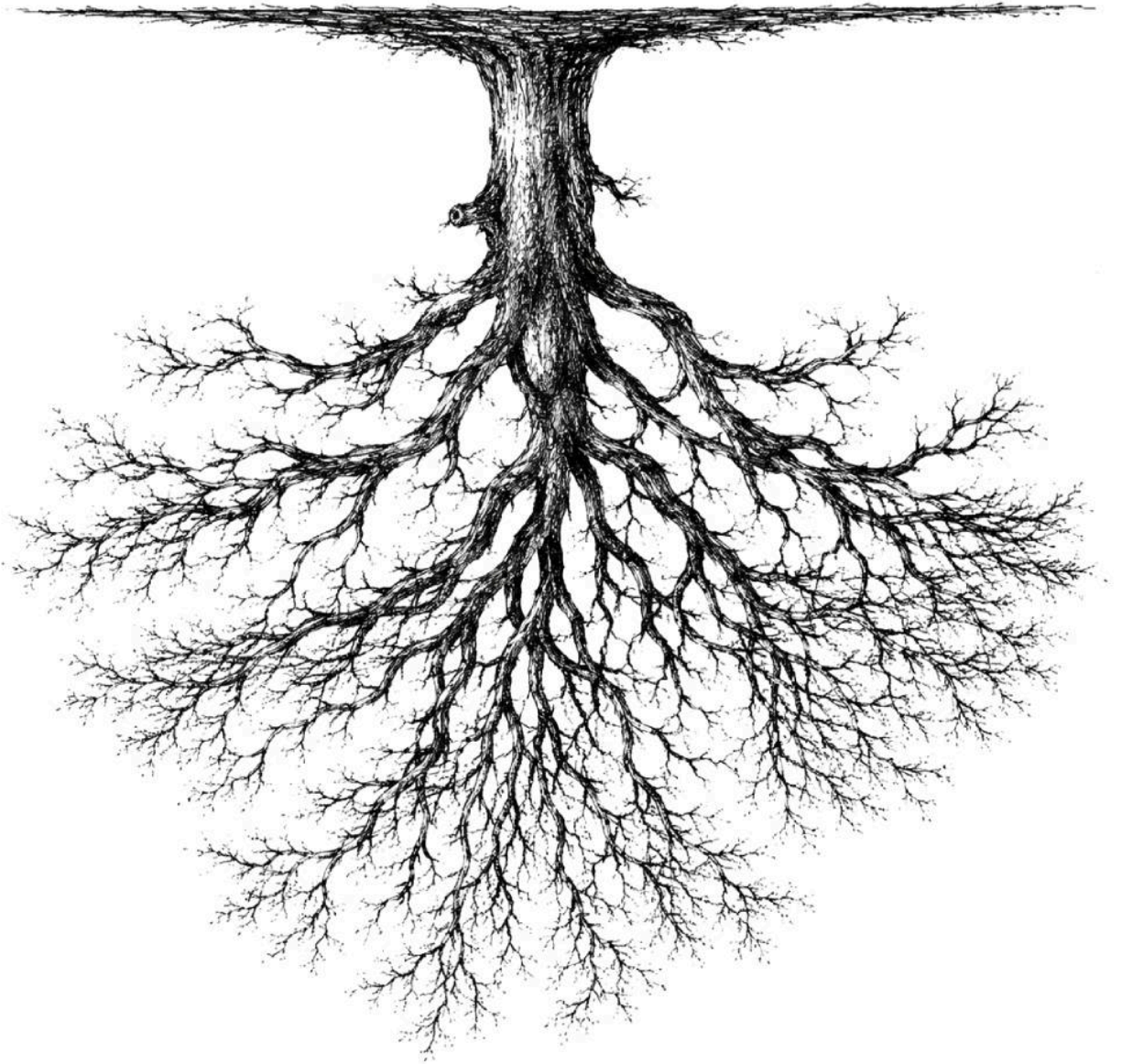
Contribute

The source code for this book is [available on GitHub](#). This book and the source code are [licensed under CC-BY-4.0](#). Feel free to report issues, submit pull requests, fork, etc.!

In the [web version of this book](#), there is a feedback link in the bottom-right corner. You can select some text and click the feedback link to create a new issue about the selected text in the GitHub repository. The link has `accesskey="1"` so it can be activated with the keyboard — how to activate it depends on the browser and OS, see [documentation on MDN about accesskey](#).

If you use X, you can follow [@htmlparserbook](#) if you want to be notified about new commits.

Chapter 1. Introduction



The DOM, parsing, and serialization

The Document Object Model (DOM) is a representation of a document as a tree of nodes. Some kinds of nodes can have child nodes (thus forming a tree).

These are the different kinds of nodes that the HTML parser can produce, and which nodes they are allowed to have as children, if any:

Document

The root node. Allowed children: **Comment**, **DocumentType**, **Element**

DocumentType

The doctype (e.g., `<!doctype html>`). No children.

Element

An element (e.g., `<p>Hello</p>`). Allowed children: **Element**, **Text**, **Comment**.

DocumentFragment

Used when parsing [templates](#). Allowed children: **Element**, **Text**, **Comment**.

Text

A text node (e.g., `Hello`). No children.

Comment

A comment (e.g., `<!-- hello -->`). No children.

Nodes can also have certain properties; for example:

- **Element** nodes have a **namespaceURI** and **localName** which together represent the element type (e.g., "an HTML **p** element"), and a list of attributes (e.g., `<html lang="en">` has one attribute).
- **Text** and **Comment** nodes have **data** which holds the node's text contents.

The DOM also includes APIs to traverse and mutate the tree with script. For example, the `divElement.remove()` method removes a node from its parent, `footerElement.append(divElement)` inserts `divElement` into `footerElement` as the last child. This is discussed in [Chapter 4. Scripting complications](#).

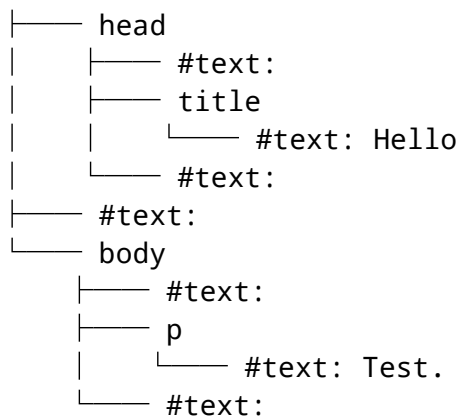
Parsing HTML means to turn a string of characters (the markup) into a DOM tree.

For example, the following document:

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Test.</p>
  </body>
</html>
```

...is parsed into the following DOM tree:

```
#document
├── DOCTYPE: html
└── html lang="en"
```



How this works is discussed in [Chapter 3. The HTML parser](#).

Serializing HTML means to do the opposite of parsing, i.e., start with a DOM representation of a document, and turning it to a string. This is discussed in [Chapter 5. Serializing](#).

A tool that is handy for quickly trying what DOM tree is produced for a piece of HTML markup is the [Live DOM Viewer](#), which Ian Hickson created when he was writing the HTML parser specification. Give it a try!

History of HTML parsers

SGML and early HTML

The earliest documentation on HTML, as far as I know, is [HyperText Mark-up Language](#), from CERN, 1992 ([also hosted on w3.org](#)). The first paragraph reads:

The WWW system uses marked-up text to represent a hypertext document for transmission over the network. The hypertext mark-up language is an SGML format. WWW parsers should ignore tags which they do not understand, and ignore attributes which they do not understand of tags which they do understand.

Already here, it is established that HTML is an SGML format, but that parsers should ignore tags and attributes they don't understand.

The next few drafts are at IETF:

- [Hypertext Markup Language \(HTML\)](#), also known as "draft-ietf-iiir-html-00", June 1993

- [HyperText Markup Language Specification - 2.0](#), November 1994

These maintain that HTML is an SGML document type, however draft-ietf-iiir-html-00 also says:

Conversely, to implement an HTML parser, one need only implement those parts of an SGML parser that are needed to parse an instance after parsing the HTML DTD.

Standard Generalized Markup Language (SGML) is a syntax framework for defining markup languages which predates HTML and the web, defined in 1986. HTML was originally inspired by SGML (in particular the [SGMLguid](#) language, an application of SGML), and later defined to be a proper application of SGML. However, web browsers have never used an actual SGML parser to parse HTML.

To parse a document, SGML required a Document Type Definition (DTD), which was specified in the doctype declaration. The DTD specifies which tags are optional, which attributes are allowed (and their values for enumerated attributes), how elements are allowed to be nested, and so forth. HTML user agents roughly integrated the DTD semantics directly into the parser without caring about how things were formally defined, and were able to parse HTML regardless of the doctype declaration.

SGML has some convenience markup features that browsers did not implement for HTML. For example, a feature called SHORTTAG allowed syntax like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<title/Misinterpreted/
<p/Little-known SGML markup features/
</html>
```

...which is, per SGML rules, equivalent to:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<title>Misinterpreted</title>
<p>Little-known SGML markup features</p>
</html>
```

But browsers parse it as a `title` start tag with a bunch of attributes, until they find a `>`:

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   │   └── title misinterpreted="" <p="" little-known="" sgm1=""
```

```
markup="" features="" <="" html=""  
  └── body
```

You may have come in contact with an SGML parser when validating your markup, for example at validator.w3.org. Up to and including HTML4, it used a DTD-based validator for HTML, which used an SGML parser. The example above would thus validate but not work in browsers. More recently, validator.w3.org started to emit warnings whenever the SHORTTAG feature was used.

As an interesting aside, when using the XML `</>` syntax in HTML, according to SGML rules it would trigger the SHORTTAG feature. When used on a void element, the slash just marks the end of the start tag, and the `>` is text content. Therefore, the following are equivalent:

```
<link rel="stylesheet" href="style.css" />
```

```
<link rel="stylesheet" href="style.css">>
```

Note the extra `>` at the end. This is equivalent to having the `>` escaped as a character reference:

```
<link rel="stylesheet" href="style.css">&gt;
```

Since the `>` (or `>`) is text, and text is not allowed in **head**, this implicitly opens the **body** element (the start and end tags of **head** and **body** are optional). However, note that web browsers never supported the SHORTTAG feature, and would instead basically ignore the slash, so it has not been any problem in practice to use `</>` on void elements (such as `link`) in HTML.

SGML is incompatible with HTML in other ways as well. For example, enumerated attributes can be shortened to only the *value* per SGML, but HTML user agents parse it as an *attribute name*.

```
<input checkbox>
```

...is per SGML rules equivalent to:

```
<input type="checkbox">
```

...but HTML parsers treat it as:

```
<input checkbox="">
```

SGML also did not specify any error handling behavior. Meanwhile, web content was overwhelmingly erroneous and relied on error handling that browsers employed.

The HTML standard has the following note about the relationship to SGML:

While the HTML syntax described in this specification bears a close resemblance to SGML and XML, it is a separate language with its own parsing rules.

Some earlier versions of HTML (in particular from HTML2 to HTML4) were based on SGML and used SGML parsing rules. However, few (if any) web browsers ever implemented true SGML parsing for HTML documents; the only user agents to strictly handle HTML as an SGML application have historically been validators. The resulting confusion — with validators claiming documents to have one representation while widely deployed Web browsers interoperably implemented a different representation — has wasted decades of productivity. This version of HTML thus returns to a non-SGML basis.

Authors interested in using SGML tools in their authoring pipeline are encouraged to use XML tools and the XML serialization of HTML.

In 2000, before Netscape 6 was released, [Gecko had a parser mode](#) called "Strict DTD" that enforced stricter rules for HTML for documents with certain doctypes. This was quickly found to be incompatible with existing web content, and was [removed](#) only two months after the parser mode was turned on in beta.

XML and XHTML

XML is, like SGML, a syntax framework for defining markup languages, and is a simplification of SGML. Unlike SGML, XML defined error handling – a syntax error must halt normal processing. It omitted many features of SGML, such as SHORTTAG and optional tags. This allowed for parsing documents without reading the DTD. DTDs were retained in XML to allow for validation, although better schema languages were developed later. In hindsight it would have been a good opportunity to drop DTD support from XML, as it complicates the parser quite a bit.

XHTML 1.0 is a reformulation of HTML 4.01 in XML. It has all the same features as HTML 4.01. Although it is technically XML, most XHTML web content was using the HTML MIME type `text/html`, which meant that browsers would use the HTML parser. The XHTML 1.0 specification has an appendix that specifies guidelines for how to write XHTML 1.0 documents while being compatible with HTML user agents. For example, section C.2. says:

Include a space before the trailing / and > of empty elements, e.g. `<br />`, `<hr />` and ``. Also, use the minimized tag syntax for empty elements, e.g. `<br />`, as the alternative syntax `<br></br>` allowed by XML gives uncertain results in many existing user agents.

Indeed, the HTML standard now specifies that `</br>` is to be parsed as `<br>`. The space before the slash was for compatibility with Netscape 4, which would parse `<br />` as an element `br/` which is not a known HTML element.

Internet Explorer, Firefox, Safari and Opera

When the HTML parser was first specified [in 2006](#), Internet Explorer was at version 6.

IE6 had an interesting HTML parser. It did not necessarily produce a tree; rather it would produce a graph, to more faithfully preserve author intent. Ill-formed markup, e.g., `<em><p></em></p>`, would result in an ill-formed DOM. This could cause scripts to go into infinite loops by just trying to iterate over the DOM.

In early 2006, Firefox was at version 1.5. Its HTML parser had its own interesting effects, but unlike IE it would always produce a strict DOM tree. Safari was similar to Mozilla, but had a different approach to handling misnested blocks in inlines. Opera also had its own approach, which involved styling nodes in ways that could not be explained by looking at the DOM tree alone. To understand what was going on, let's go back and read what Ian Hickson, then the editor of the HTML standard, [found when he was specifying the HTML parser](#).

Imagine the following (invalid) markup:

```
<!DOCTYPE html><em><p>XY</p></em>
```

What should the DOM look like? The general consensus is that the DOM should look like this:

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       ├── em
│       │   └── p
│       │       └── #text: XY
```

That is, the p element should be completely inside (that is, a child of) the em element.

No problem so far.

Now consider this markup:

```
<!DOCTYPE html><em><p>X</p>Y</em>
```

What should the DOM look like?

This is where things start getting hairy. I've covered [a similar case](#) before, so I'll just summarise the results:

Windows Internet Explorer

The DOM is not a tree. The text node for the "Y" is a child of both the p element and the body element. Violates the DOM Core specifications.

Opera

The DOM is a simple tree, the same as for the first case, but the "Y" is not emphasised. Violates the CSS specifications.

Mozilla and Safari

The DOM looks like this:

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       ├── em
│       └── p
│           ├── em
│           │   ├── #text: X
│           │   └── #text: Y
│           └── #text: Y
```

...which basically means that malformed invalid markup gets handled differently than well-formed invalid markup.

In the past, I would have stopped here, made some wry comment about the insanity that is the Web, and called it a day.

But I'm **trying to spec this**. Stopping is not an option.

What IE does is insane. What Opera does is also insane. Neither of those options is something that I can put in a specification with a straight face.

This leaves the Mozilla/Safari method.

It's weird, though. If you look at the two examples above, you'll notice that their respective markups start the same — both of them start with this markup:

```
<!DOCTYPE html><em><p>X
```

Yet the end result is quite different, with one of the elements (the p) having different parents in the two cases. So when do the browsers decide what to do? They can't be buffering content up and deciding what to do later, since that would break incremental rendering. So what exactly is going on?

Well, let's check. What do Mozilla and Safari do for that truncated piece of markup?

Mozilla

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       ├── em
│       └── p
│           └── em
│               └── #text: X
```

Safari

```
#document
├── html
│   └── body
│       └── em
│           └── p
│               └── #text: X
```

Hrm. They disagree. Mozilla is using the "malformed" version, and Safari is using the "well-formed" version. Why? How do they decide?

Let's look at Safari first, by running a script while the parser is running. First, the simple case:

```
<!DOCTYPE html>
<em>
  <p>
    XY
    <script>
      var p = document.getElementsByTagName('p')[0];
      p.title = p.parentNode.tagName;
    </script>
  </p>
</em>
```

Result:

```
#document
├── html
│   └── body
│       └── em
│           ├── #text:
│           ├── p title="EM"
│           │   ├── #text: XY
│           │   └── script
│           │       └── #text: var p =
document.getElementsByTagName('p')[0]; p.title =
p.parentNode.tagName;
│           ├── #text:
│           └── #text:
```

Exactly as we'd expect. The parentNode of the p element as shown in the DOM tree view is the same as shown in the title attribute value, namely, the **em** element.

Now let's try the bad markup case:

```
<!DOCTYPE html>
<em>
  <p>
    X
    <script>
      var p = document.getElementsByTagName('p')[0];
      p.title = p.parentNode.tagName;
    </script>
  </em>
  Y
</p>
```

Result:

```
#document
├── html
│   └── body
│       ├── em
│       │   ├── #text:
│       │   └── p title="EM"
│       │       ├── em
│       │       │   ├── #text: X
│       │       │   └── script
│       │       │       └── #text: var p =
document.getElementsByTagName('p')[0]; p.title =
p.parentNode.tagName;
│       │       ├── #text:
│       │       └── #text: Y
│       └──
```

Wait, what?

When the embedded script ran, the parent of the `p` was the `em`, but when the parser had finished, the DOM had changed, and the parent was no longer the `em` node!

If we look a little closer:

```
<!DOCTYPE html>
<em>
  <p>
    X
    <script>
      var p = document.getElementsByTagName('p')[0];
      p.setAttribute('a', p.parentNode.tagName);
    </script>
  </em>
  Y
  <script>
    var p = document.getElementsByTagName('p')[0];
    p.setAttribute('b', p.parentNode.tagName);
  </script>
</p>
```

...we find:

```
#document
├── html
│   └── body
│       ├── em
│       │   ├── #text:
│       │   └── p a="EM" b="BODY"
│       │       ├── em
│       │       │   ├── #text: X
│       │       │   └── script
│       │       │       └── #text: var p =
│       │       │           document.getElementsByTagName('p')[0]; p.setAttribute('a',
│       │       │           p.parentN...
│       │       └── #text:
│       │           ├── #text: Y
│       │           └── script
│       │               └── #text: var p =
│       │                   document.getElementsByTagName('p')[0]; p.setAttribute('b',
│       │                   p.parentN...
│       └── #text:
```

...which is to say, the parent changes half way through! (Compare the `a` and `b` attributes.)

What actually happens is that Safari notices that something bad has happened, and moves the element around in the DOM. After the fact. (If you remove the `p` element from the DOM in that first script block, then [Safari crashes](#).)

How about Mozilla? Let's try the same trick. The result:

```
#document
├── html
│   └── body
│       ├── em
│       │   └── #text:
│       │       p a="BODY" b="BODY"
│       │           ├── em
│       │           │   ├── #text: X
│       │           │   └── script
│       │           └── #text: var p =
│       │               document.getElementsByTagName('p')[0]; p.setAttribute('a',
│       │               p.parentN...
│       │                   ├── #text:
│       │                   ├── #text: Y
│       │                   └── script
│       │                       ├── #text: var p =
│       │                       document.getElementsByTagName('p')[0]; p.setAttribute('b',
│       │                       p.parentN...
│       │                           └── #text:
```

It doesn't re-parent the node. So what does Mozilla do?

It turns out that Mozilla does a pre-parse of the source, and if a part of it is well-formed, it creates a well-formed tree for it, but if the markup isn't well-formed, or if there are any script blocks, or, for that matter, if the TCP/IP packet boundary happens to fall in the wrong place, or if you write the document out in two `document.write()`s instead of one, then it'll make the more thorough nesting that handles ill-formed content.

Who would have thought that you would find Heisenberg-like quantum effects in an HTML parser. I mean, I knew they were obscure, but this is just taking the biscuit.

The problem is I now have to determine which of these four options to make the other three browsers implement (that is, which do I put in the spec). What do you think is the most likely to be accepted by the others? As a reminder, the options are incestual elements that can be their own uncles, elements who have secret lives in the rendering engine, elements that change their mind about who their parents are half-way through their childhood, and quantum elements whose parents change depending on whether you observe their birth or not.

The key requirements are probably:

- Coherence: scripts that rely on DOM invariants (like the fact that the DOM is a tree) shouldn't go off into infinite loops.

- Transparency: we shouldn't have to describe a whole extra section that explains how the CSS rendering engine applies to HTML DOMs; CSS should just work on the real DOM as you would see it from script.
- Predictability: it shouldn't depend on, e.g., the protocol or network conditions — every browser should get the same DOM for the same original markup in all situations.

The least worse [sic] option is probably the Safari-style on-the-fly reparenting, I think, but I'm not sure. It's the only one that fits those requirements. Is there a fifth option I'm missing?

Well, it appeared that there wasn't a fifth option, as the Safari approach was what was adopted. This is called the Adoption Agency Algorithm in the HTML standard.

The HTML parser is specified

A couple of years prior to the HTML parser being specified, in June 2004, the W3C decided to discontinue work on HTML at a workshop on [Web Applications and Compound Documents](#). In response, Opera, Mozilla, and Apple set up the Web Hypertext Application Technology Working Group (WHATWG), an initiative, open for anyone to contribute, to extend HTML in a backwards-compatible manner (in contrast with the W3C XForms and XHTML 2.0 specifications, which were [by design not backwards compatible](#)). One of the grounding principles of the WHATWG was well-defined error handling, which had not been addressed for HTML previously.

In February 2006, Ian Hickson [announced on the WHATWG mailing list](#) that "the first draft of the HTML5 Parsing spec is ready". He had done what had never been attempted before; define how to parse HTML.

So...

The first draft of the HTML5 Parsing spec is ready.

I plan to start implementing it at some point in the next few months, to see how well it fares.

It is, in theory, more compatible with IE than Safari, Mozilla, and Opera, but there are places where it makes intentional deviations (e.g. the comment parsing, and it doesn't allow `<object>` in the `<head>` -- browsers are inconsistent about this at the moment, and we're dropping `declare=""` in HTML5

anyway so it isn't needed anymore; I plan to look for data on how common this is in the Web at some point in the future to see if it's ok for us to do this).

It's not 100% complete. Some of the things that need work are:

- Interaction with document.open/write/close is undefined
- How to determine the character encoding
- Integration with quirks mode problems
- `<style>` parsing needs tweaking if we want to exactly match IE
- `<base>` parsing needs tweaking to handle multiple `<base>`s
- `<isindex>` needs some prose in the form submission section
- No-frames and no-script modes aren't yet defined
- Execution of `<script>` is not yet defined
- New HTML5 elements aren't yet defined
- There are various cases (marked) where EOF handling is undefined
- Interaction with the "load" event is undefined

However, none of the above are particularly critical to the parsing.

If you have any comments, please send them. This part of the spec should be relatively stable now, so now is a good time to review it if you want to. And if anyone wants to implement it to test it against the real live Web content out there, that's encouraged too. :-)

The more evidence we have that this parsing model is solid and works with the real Web, the more likely we are to be able to convince Apple/Safari/Mozilla to implement it. And if all the browsers implement the same parsing model, then HTML interoperability on the Web will take a huge leap forward. T'would be save [sic] everyone a lot of time.

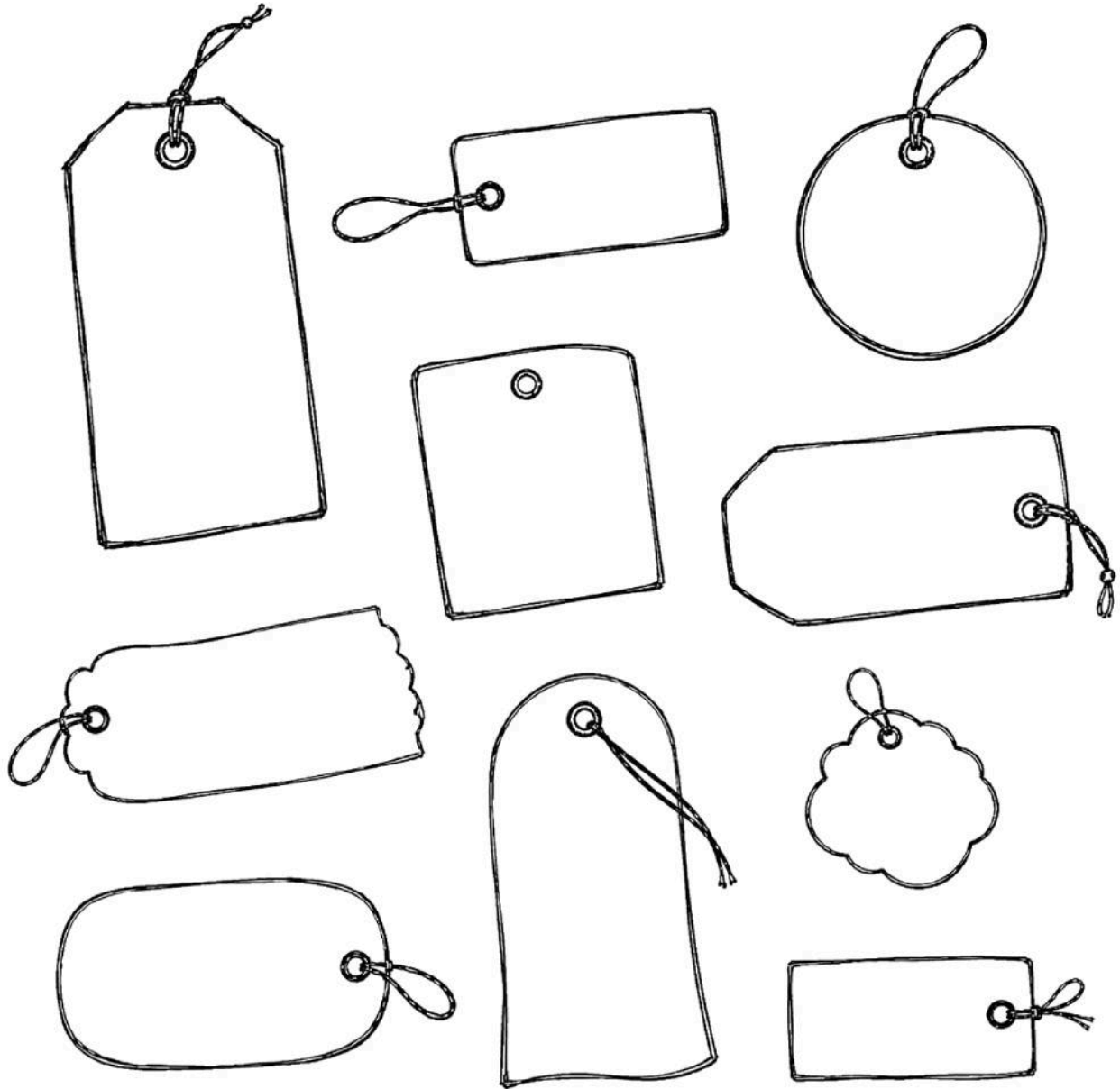
Wouldn't it, indeed.

The following table shows when each browser shipped with a new HTML parser implementation, conforming to the specification.

Browser	Version	Release date
Firefox	4	2011-03-22
Safari	5.1	2011-07-20

Browser	Version	Release date
Opera	12	2012-08-30
Internet Explorer	10	2012-09-04

Chapter 2. The HTML syntax



This chapter covers the syntax of HTML, i.e. how to write HTML. This chapter is similar to the [Writing HTML documents](#) section of the HTML standard.

The doctype

The doctype is required because without a doctype, browsers use quirks mode for the document, which changes some behavior, mainly in CSS. Quirks mode was introduced by IE5 for Mac, released in 2000, in an attempt to both be compatible with the contemporary legacy and with the CSS1 specification. This approach was then copied by all browsers and has now been specified. There are now three rendering modes for HTML:

- quirks mode,
- limited-quirks mode,
- no-quirks mode.

[Activating Browser Modes with Doctype](#) by Henri Sivonen has more information on doctype switching. The WHATWG [Quirks Mode standard](#) specifies some of the effects of the quirks mode.

The doctype can be either:

```
<!doctype html>
```

...case-insensitive, or:

```
<!doctype html system "about:legacy-compat">
```

Also case-insensitive, except for the "about:legacy-compat" part.

The purpose of the longer doctype is for compatibility with markup generators that are unable to produce the short doctype. If you don't find yourself in such a situation, just use the short doctype.

Prior versions of HTML had other doctypes that are now defined to trigger one of the different rendering modes. For example, this HTML 4.01 doctype triggers no-quirks mode:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">
```

One of my first [contributions](#) to the WHATWG, in June 2005, was to propose to change the doctype to `<!doctype html>`. Finally a doctype that can be remembered! (Though, for some reason, I still remember how to type `<!doctype html public "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">`. Sigh.)

Elements

HTML defines the following kinds of elements:

- **Void elements.** The list of conforming elements is: `area`, `base`, `br`, `col`, `embed`, `hr`, `img`, `input`, `link`, `meta`, `param`, `source`, `track`, `wbr`. Non-conforming elements that parse and serialize like void elements are: `basefont`, `bgsound`, `frame`, `keygen`.
- **The `template` element.** This element has a category of its own.
- **Raw text elements.** The list of conforming elements is: `script`, `style`. The `iframe` element parses and serializes like raw text elements, however, in conforming documents, the `iframe` element must be empty. The non-conforming elements that parse and serialize like raw text elements are: `xmp`, `noembed`, `noframes`, `noscript` (if scripting is enabled). The non-conforming `plaintext` element serializes like raw text element, but it has special parsing rules such that from a parsing perspective, it has a category of its own.
- **Escapable raw text elements.** `textarea`, `title`.
- **Foreign elements.** SVG and MathML elements.
- **Normal elements.** All other HTML elements.

All kinds of elements can have a start tag, although for some elements the start tag is optional.

Void elements consist of just a start tag.

```
<br>
```

The `template` element is special because its contents are parsed into a separate `DocumentFragment` instead of being children of the element itself. This is discussed in more detail in the [Templates](#) section in [Chapter 3. The HTML parser](#).

```
<template></template>
```

Raw text elements, escapable raw text elements and normal elements have a start tag, some contents, and an end tag (but some elements have optional end tags, or optional start and end tags). Raw text means that the contents are treated as text instead of as markup, except for the end tag, and except that script has pretty special parsing rules (see *Script states* of the *Tokenizer*). Escapable raw text is like raw text, except that character references work.

```
<title>SpiderMonkey &amp; the GC Jitters</title>
```

Normal elements can have text (except < and ambiguous ampersand), character references, other elements, and comments.

The `pre` and `textarea` elements have a special rule: they may begin with a newline that will be ignored by the HTML parser. To have content that actually starts with a newline, two newlines thus have to be used. (A newline in HTML is a line feed, a carriage return, or a CRLF pair.) For example, the following is equivalent to `<pre>Use the force</pre>` (without a newline):

```
<pre>
Use the force</pre>
```

Foreign elements are slightly closer to XML in their syntax: `</>` works (self-closing start tag), CDATA sections work (`<![CDATA[ ... ]]>`, the contents are like raw text). But note that other aspects still work like HTML; element names and attribute names are case-insensitive, and XML namespaces don't work (only some namespaced attributes work with a predefined prefix).

```
<p>Circling the drain.
<svg viewBox="-1 -1 2 2" width=16><circle r=1 /></svg>
</p>
```

Documents

An HTML document consists of a doctype followed by an `html` element, and there may be whitespace and comments before, between, and after. The following example is a complete and conforming HTML document:

```
<!-- a comment -->
<!doctype html>
<html lang=en>
  <head>
    <title>Key to success</title>
  </head>
  <body>
    <p>Such like these, unless combined, are inane.</p>
  </body>
</html>
```

Start tags

A start tag has this format:

<, the tag name (case-insensitive), whitespace (if there are attributes), any number of attributes separated by whitespace, optionally some whitespace, >. (In the HTML syntax, whitespace means [ASCII whitespace](#), i.e., tab, line feed, form feed, carriage return, or space.)

```
<p class="warning">
```

For void elements, the tag may end with either > or />, although the slash makes no difference.

```
<hr />
```

Foreign elements (SVG and MathML) support self-closing start tags, which end with /> and means there are no contents and no end tag. The element name for foreign elements is case-insensitive in the HTML syntax.

```
<CIRCLE r="1" />
```

End tags

An end tag has this format:

</, the tag name (case-insensitive), optionally whitespace, >.

```
</p>
```

Attributes are not allowed on end tags.

Attributes

Attributes come in a few different formats.

- **Empty attribute syntax.** This is just the attribute name. The value in the DOM will be the empty string. This syntax is often used for boolean attributes, but is allowed for any attribute (provided that the empty string is an allowed value). For example:

<video preload>

- **Unquoted attribute value syntax.** The attribute name, optionally whitespace, =, optionally whitespace, then the value, which can't be the empty string and is not allowed to contain whitespace or these characters: " ' = < > and `. If this is the last attribute and the start tag ends with /> (which is allowed on void elements and foreign elements), there has to be whitespace before the slash (otherwise the slash becomes part of the value). For example:

```
<source src=bbb_sunflower_2160p_60fps_normal.mp4 />
```

- **Single-quoted attribute value syntax.** The attribute name, optionally whitespace, =, optionally whitespace, ', the value not containing ', then '. For example:

```
<track src='big-buck-bunny.webvtt'>
```

- **Double-quoted attribute value syntax.** The attribute name, optionally whitespace, =, optionally whitespace, ", the value not containing ", then ". For example:

```
<a href="https://peach.blender.org/download/">Download Big Buck Bunny</a>
```

All attribute names are case-insensitive, including attributes on SVG and MathML elements.

All attribute values support character references. This can be particularly relevant for URLs in attributes, which sometimes contain & that should be escaped as **&**, lest it be interpreted as a character reference.

```
<a href="?title=Lone+Surrogates&amp;reg">
```

If the ampersand was unescaped in this example, like this:

```
<a href="?title=Lone+Surrogates&reg">
```

...then **&reg** would be interpreted as a named character reference, which expands to "®", i.e., it's equivalent to:

```
<a href="?title=Lone+Surrogates®">
```

Duplicate attributes, i.e., two attributes with the same name, are not allowed.

```
<p class="cool" class="uncool">
```

Foreign elements support the following namespaced attributes (with fixed prefixes):
xlink:actuate, xlink:arcrole, xlink:href, xlink:role, xlink:show, xlink:title, xlink:type, xml:lang, xml:space, xmlns (without prefix but is a namespaced attribute), xmlns:xlink.

```
<svg xmlns="http://www.w3.org/2000/svg">
```

Note that in the HTML syntax, it's optional to declare the namespace.

```
<svg>
```

Optional tags

Certain tags can be omitted if the resulting DOM doesn't change if they are so omitted, including "minor" changes such as where whitespace ends up or where a comment ends up. The rules for when they can be omitted are slightly convoluted, but they assume that the DOM is not allowed to change by omitting a tag. It is however conforming to intentionally move a tag such that omitting it no longer changes the DOM.

For example, consider this snippet:

```
<p>Can a paragraph be one word long?</p>
<p>Yes.</p>
```

Because there is a line feed between the paragraphs, there will be a **Text** node for it in the DOM. Omitting the end tags will cause the line feed to be part of the first paragraph instead:

```
<p>Can a paragraph be one word long?
<p>Yes.
```

However, in most cases this makes no difference at all. (It can make a difference if you style the paragraphs as **display: inline-block**, for example.)

For the exact rules on when tags can be omitted, please consult the HTML standard.

Here are the tags that may (sometimes) be omitted:

Element	Start tag	End tag
html	Omissible	Omissible
head	Omissible	Omissible
body	Omissible	Omissible
li		Omissible
dt		Omissible
dd		Omissible
p		Omissible
rt		Omissible

Element	Start tag	End tag
rp		Omissible
optgroup		Omissible
option		Omissible
colgroup	Omissible	Omissible
caption		Omissible
thead		Omissible
tbody	Omissible	Omissible
tfoot		Omissible
tr		Omissible
td		Omissible
th		Omissible

Character references

There are three kinds of character references:

- **Named character reference.** `&`, the name, `;`. There are over two thousand names to choose from. These are case-sensitive, although a few characters have character reference names in both all-lowercase and all-uppercase (e.g., `<` and `<`).

` `;

- **Decimal numeric character reference.** `&#`, a decimal number of a code point, `;`.

` `;

- **Hexadecimal numeric character reference.** `&#x` or `&#X`, a hexadecimal number of a code point, `;`. The hexadecimal number is case-insensitive.

` `;

HTML has a concept of an ambiguous ampersand, which is `&`, alphanumerics (a-zA-Z0-9), `;`, when this is not a known named character reference. Ambiguous ampersands are not allowed. The following is an example of an ambiguous ampersand:

`I've sent a support request to AT&T; no reply, yet.`

However, other unescaped ampersands are technically allowed:

`Ind. Unrealisk & Ind. Brunn`

CDATA sections

CDATA sections can only be used in foreign content, and have this format:

`<![CDATA[ (case-sensitive), text not containing ]]>`, then `]]>`.

```
<svg><title><![CDATA[ The <html>, <head>, & <title> elements ]]></title>
... </svg>
```

Comments

Comments have this format:

`<!--` followed by any text (with some restrictions, detailed below), then `-->`.

```
<!-- Hello -->
```

The text is not allowed to contain `-->` since that would end the comment.

A somewhat recent change to comment syntax is that `--` is now allowed in the text. This is not allowed in XML.

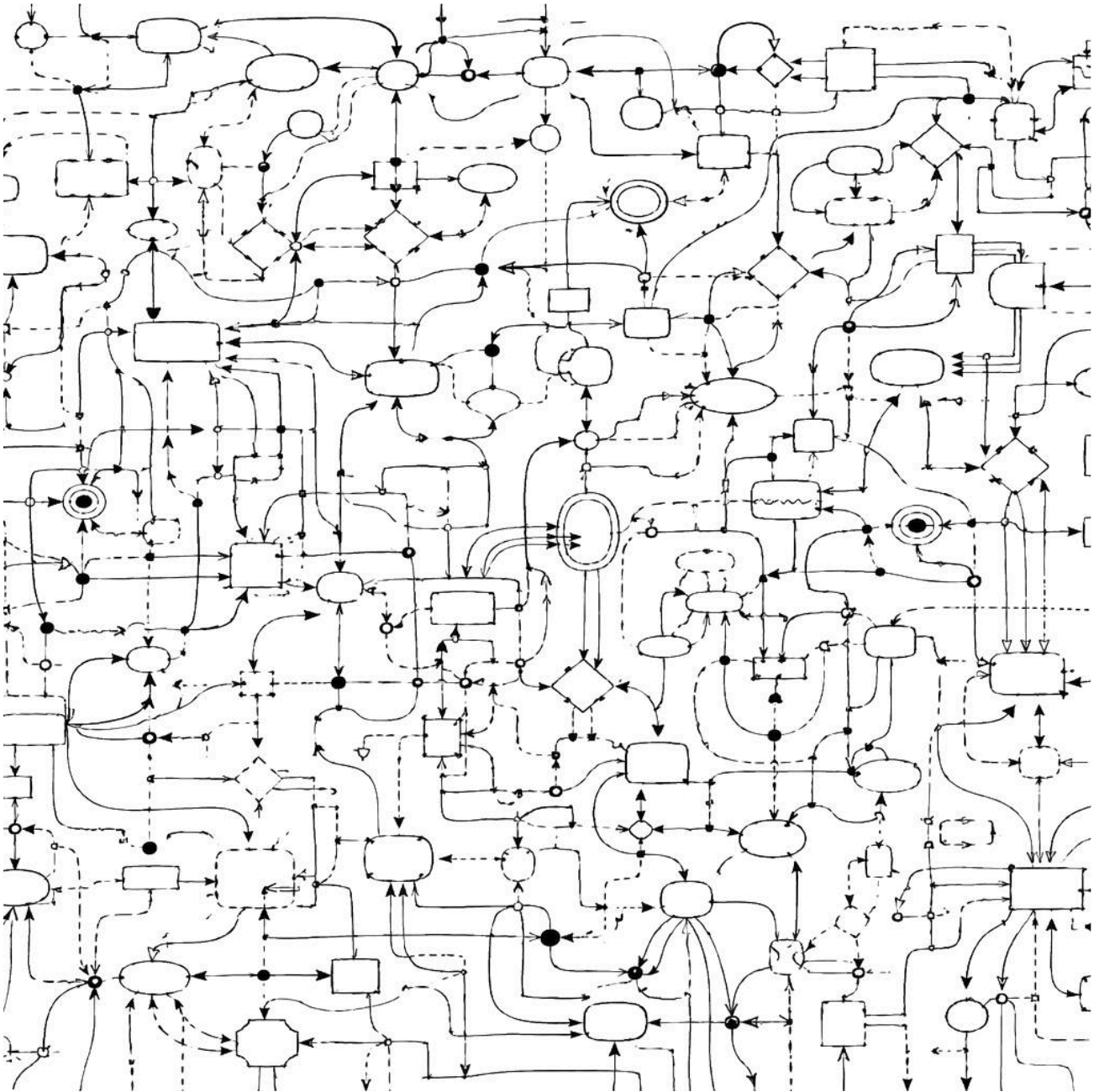
```
<!-- Hello -- there -->
```

The text is not allowed to contain `<!--` since that is an indicator of a nested comment, and nested comments don't work.

```
<!-- <!-- this is an error --> -->
```

The text is not allowed to start with `>` or `->` or contain `--!>` because the HTML parser will end the comment at that point.

Chapter 3. The HTML parser



Overview of the HTML parser

The HTML parser consists of two major components, the tokenizer and the tree builder, which are both state machines.

In the typical case, the input for the HTML parser comes from the network. However, it can also come from script with the `document.write()` API, which complicates the model. This is discussed in the [document.write\(\)](#) section of [Chapter 4. Scripting complications](#).

In the typical case, parsing a document goes through these stages:

Network ⇒ Byte Stream Decoder ⇒ Input Stream Preprocessor ⇒ Tokenizer ⇒ Tree builder ⇒ DOM

For example, consider the following document:

```
<!doctype html><p>Hello world.
```

Bytes go over the network and a decoder will produce a stream of code points (the details of how that works is a topic of another book). The tokenizer walks through the stream of code points, character by character, and emits tokens; in this case: a doctype token, a start tag token (p), and a series of character tokens (one token per character, although implementations can optimize by combining character tokens, if the end result is equivalent). The tree builder takes those tokens and builds the following DOM:

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       └── p
│           └── #text: Hello world.
```

Note that the tree builder created some elements (html, head, body) that did not have any corresponding tags in the source text. These elements have optional start and end tags, but implied tags can also happen in non-conforming cases, such as when a required end tag is omitted (more on this in the [Implied tags](#) section).

Error handling

The HTML parser specification specifies exactly what to do in case of an error. Technically, an implementation is allowed to abort processing upon an error, but no browser does that. Instead, they follow the specification to recover from the error in some particular way, which is carefully designed to be compatible with web content.

When the parser identifies something that is an error, it says that "it is a parse error". Some parse errors have an identifying code. For example, for the character reference `&#0;`:

If the number is 0x00, then this is a null-character-reference parse error. Set the character reference code to 0xFFFFD.

Detecting character encoding

The character encoding of the document must be specified (but not all documents do this). Not only must it be specified, but it must be UTF-8 (again not all documents do this).

Character encoding can be specified in a number of ways. First, it can be specified at the transport layer; e.g., HTTP **Content-Type** can have a **charset** parameter that gives the encoding of the document. If present, this wins over a **meta** element encoding declaration.

The document can also start with a byte order mark (BOM), for UTF-8 and UTF-16 encodings. If present, this wins over both HTTP and **meta** encoding declarations.

Otherwise, a **meta** element can be used. It comes in two forms.

```
<meta charset="utf-8">
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

In earlier versions of HTML, only the second variant was specified, but browsers already supported the first variant as well. The reason for this is that there were documents on the web that incorrectly omitted quote marks even though the value contains a space.

```
<meta http-equiv=Content-Type content=text/html; charset=utf-8>
```

Note that "charset=utf-8" appears as its own attribute.

The HTML standard took the opportunity to make the shorter syntax conforming (that is, **<meta charset="utf-8">**), which lowers the barrier to specify an encoding for the document.

Originally, **<meta http-equiv>** was a feature intended for web servers, not for clients. The idea was that servers could scan for the http-equiv in an HTML file, and set the corresponding HTTP headers when serving it. Servers didn't do that. Instead, web browsers picked it up.

Also note the absurdity of encoding the character encoding in the character encoding of the document that you're trying to decode, especially when it's not the very first thing in the file (like in, e.g., CSS and XML).

Before the HTML parser starts, a prescan of the byte stream can take place in an attempt to find a character encoding declaration. This prescan is essentially a simplified HTML parser. The prescan is usually done on the first 1024 bytes, and there is a conformance requirement for documents to include the encoding declaration within the first 1024 bytes.

The prescan supports skipping comments and bogus comments, but doesn't make any effort to skip, e.g., `style` elements. So something that looks like an encoding declaration inside a `style` element will be picked up as an encoding declaration by the prescan.

The encoding that is picked up by the prescan, if any, is only tentative. The actual HTML parser can find an encoding declaration with a different value (uncommon but possible), and if it does, the parser will change the encoding to the newly found value, either on the fly (if possible), or by reloading the document with the new encoding.

If the document is encoded in a UTF-16 encoding, then any inline encoding declaration is ignored, since UTF-16 will be detected as such by its bit pattern (usually by the initial byte order mark). The prescan assumes an ASCII compatible encoding, so it will not find an encoding declaration in a UTF-16 document (assuming the declaration is encoded in UTF-16). The HTML parser can find an encoding declaration in a UTF-16 document, but if it says anything but the same variant of UTF-16, it must be incorrect and therefore it is ignored.

However, if the document is not UTF-16 and an encoding declaration is found that claims UTF-16, it will be interpreted as saying UTF-8. The declaration is incorrect, but assuming UTF-8 is apparently better than ignoring it.

Another rewrite is the `x-user-defined` encoding label, which is changed to `windows-1252`, but only when found in a `meta` element, not anywhere else. The reason for this is that web sites used this encoding label together with a custom font to get visual rendering of their language's non-ASCII characters, instead of using Unicode and a proper font. Meanwhile, `x-user-defined` is [an actual encoding](#) that web pages use for [binary data using the XMLHttpRequest API](#). The solution that Chrome invented and that [Firefox copied](#) was to rewrite this label but just for `meta` in HTML.

If no encoding declaration is found, then the default will usually depend on the user's locale. The most common default is `windows-1252`, but there are 33 other locales with other defaults. For example, Arabic defaults to `windows-1256`, and Japanese defaults to `Shift_JIS`. Having locale-specific default encodings on a global information network is, of course, also absurd.

When the prescan has happened and potentially found a tentative encoding to use, we're ready to preprocess the input stream.

Preprocessing the input stream

At this stage, we are working with a stream of code points rather than a stream of bytes. This is responsible for normalizing newlines to line feed characters. This is defined as follows:

U+000D CARRIAGE RETURN (CR) characters and U+000A LINE FEED (LF) characters are treated specially. Any LF character that immediately follows a CR character must be ignored, and all CR characters must then be converted to LF characters. Thus, newlines in HTML DOMs are represented by LF characters, and there are never any CR characters in the input to the tokenization stage.

Note that scripts can insert text into the input stream using the `document.write()` API. Preprocessing the input stream thus happens for such text as well.

For implementations that support `document.write()`, it is important that this is implemented as specified, in particular how to handle CRLF. Consider the following document.

```
<!DOCTYPE html><pre>x<script>document.write('\r');</script>
y
```

Without running script, the input stream just has one line feed. Loading the document with scripting enabled, there will be a CRLF pair; the script writes a CR character, which appears in the input stream after the script end tag, and then the source markup has a LF as the next character.

However, the newline needs to be present in the DOM while the script runs, because the script is able to observe what the DOM tree looks like, or it might do another `document.write()`. So it can't wait for the following LF before inserting the LF in the DOM.

```
<!DOCTYPE html>
<pre id=pre>x<script>
document.write('\r');
alert(pre.innerText.length);
</script>
y
```

This will alert "2" (the "x" and the carriage return converted to line feed). The line feed after the `</script>` tag, which appears in the input stream after the script has run, is then ignored.

Tokenizer

The tokenizer processes each character in the input stream with a state machine. The output is a series of tokens that are used by the tree construction stage.

The possible tokens are: doctype, start tag, end tag, comment, character, and end-of-file. The tokens have various properties:

- Doctype tokens: name, public identifier, system identifier, force-quirks flag.
- Start and end tag tokens: tag name, self-closing flag, attributes.
- Comment and character tokens: data.
- End-of-file has no properties.

Tags and text

Let's walk through a simple example to see how the tokenizer works: how it switches states and what tokens are produced.

```
<p>Hello</p>
```

The tokenizer always starts in the *data state*, which is defined as follows:

Consume the next input character:

U+0026 AMPERSAND (&)

Set the return state to the data state. Switch to the character reference state.

U+003C LESS-THAN SIGN (<)

Switch to the tag open state.

U+0000 NULL

This is an unexpected-null-character parse error. Emit the current input character as a character token.

EOF

Emit an end-of-file token.

Anything else

Emit the current input character as a character token.

The next input character is the "<", which switches the tokenizer to the *tag open state*:

Consume the next input character:

U+0021 EXCLAMATION MARK (!)

Switch to the markup declaration open state.

U+002F SOLIDUS (/)

Switch to the end tag open state.

ASCII alpha

Create a new start tag token, set its tag name to the empty string.

Reconsume in the tag name state.

U+003F QUESTION MARK (?)

This is an unexpected-question-mark-instead-of-tag-name parse error.

Create a comment token whose data is the empty string. Reconsume in the bogus comment state.

EOF

This is an eof-before-tag-name parse error. Emit a U+003C LESS-THAN SIGN character token and an end-of-file token.

Anything else

This is an invalid-first-character-of-tag-name parse error. Emit a U+003C LESS-THAN SIGN character token. Reconsume in the data state.

The input so far is "<p", and the "p" falls into ASCII alpha clause. At this point a start tag token is created (but not yet emitted), and then the "p" is reconsumed in the *tag name state*:

Consume the next input character:

U+0009 CHARACTER TABULATION (tab); U+000A LINE FEED (LF); U+000C FORM FEED (FF); U+0020 SPACE

Switch to the before attribute name state.

U+002F SOLIDUS (/)

Switch to the self-closing start tag state.

U+003E GREATER-THAN SIGN (>)

Switch to the data state. Emit the current tag token.

ASCII upper alpha

Append the lowercase version of the current input character (add 0x0020 to the character's code point) to the current tag token's tag name.

U+0000 NULL

This is an unexpected-null-character parse error. Append a U+FFFD REPLACEMENT CHARACTER character to the current tag token's tag name.

EOF

This is an eof-in-tag parse error. Emit an end-of-file token.

Anything else

Append the current input character to the current tag token's tag name.

The input is still "<p", and the "p" falls into the *Anything else* clause. The start tag token's name is now "p". The tokenizer stays in this state for the next character.

The input is now <p>; the ">" switches back to the *data state* and emits the start tag token. This token will immediately be handled by the tree builder before the tokenizer can continue.

The next few character are "Hello", handled in the *data state*. Each character emits a character token with the data being the given character, i.e., 5 character tokens "H" "e" "l" "l" "o".

The </p> goes through similar states as the start tag, but obviously creates an end tag token instead. The "/" switches to the *end tag open state*:

Consume the next input character:

ASCII alpha

Create a new end tag token, set its tag name to the empty string.

Reconsume in the tag name state.

U+003E GREATER-THAN SIGN (>)

This is a missing-end-tag-name parse error. Switch to the data state.

EOF

This is an eof-before-tag-name parse error. Emit a U+003C LESS-THAN SIGN character token, a U+002F SOLIDUS character token and an end-of-file token.

Anything else

This is an invalid-first-character-of-tag-name parse error. Create a comment token whose data is the empty string. Reconsume in the bogus comment state.

Finally, the end of the input stream is said to be an "EOF" character, which emits an end-of-file token. The series of tokens produced is thus:

Start tag (p), character (H), character (e), character (l), character (l), character (o), end tag (p), end-of-file.

This book contains a number of quizzes, which you should be able to answer with the information in this book. These quizzes originally took place on X. Here is the first quiz in this book:

#HTMLQuiz (don't cheat!): What kind of node will be inserted into the body for such contents?

```
<body></хелой></body>
```

- a self-closing tag
- a text node
- a comment node
- none (it will be ignored)

Note that that isn't the ASCII "x" but instead U+0445 CYRILLIC SMALL LETTER HA. If we look at the *end tag open state* above, we find:

Anything else

This is an invalid-first-character-of-tag-name parse error. Create a comment token whose data is the empty string. Reconsume in the bogus comment state.

The correct answer is thus a comment node.

Note that the *start tag open* state handles non-ASCII alpha differently; it will emit the < and the current input character as character tokens.

Attributes

#HTMLQuiz (don't cheat :)). What class will the `<div class="a" class="b">` have?

- "a"
- "b"
- both "a" and "b"

Given the following markup:

```
<p id="x">
```

The tokenizer goes through these states:

1. Start in the *data state*.
2. Consume "<": Switch to the *tag open state*.
3. Consume "p": Reconsume in the *tag name state*.
4. Consume "p": Append the current input character to the current tag token's tag name.
5. Consume " ": Switch to the *before attribute name state*.
6. Consume "i": Reconsume in the *attribute name state*.
7. Consume "i": Append the current input character to the current attribute's name.
8. Consume "d": Append the current input character to the current attribute's name.
9. Consume "=": Switch to the *before attribute value state*.
10. Consume "\"": Switch to the *attribute value (double-quoted) state*.
11. Consume "x": Append the current input character to the current attribute's value.
12. Consume "\"": Switch to the *after attribute value (quoted) state*.
13. Consume ">": Switch to the *data state*. Emit the current tag token.
14. Consume EOF: Emit an end-of-file token.

The *attribute name state* says:

When the user agent leaves the attribute name state (and before emitting the tag token, if appropriate), the complete attribute's name must be compared to the other attributes on the same token; if there is already an attribute on the token with the exact same name, then this is a duplicate-attribute parse error and the new attribute must be removed from the token.

The correct answer to the quiz is thus "a". Here's another quiz about attributes:

Let's try another one. What attributes will `<img>` contain in the following case?
#HTMLQuiz

```
<img src=1.png /re/>
```

- `{src:"1.png"}`
- `{src:"1.png /re":""}`
- `{src:"1.png", "/re":""}`
- `{src:"1.png", "re":""}`

Let's check. The first part, before the slash, is straightforward.

```
<img src=1.png
```

The tokenizer will be in the *before attribute name state* when it consumes the `/`, which says:

U+002F SOLIDUS (`/`); U+003E GREATER-THAN SIGN (`>`); EOF
Reconsume in the after attribute name state.

The *after attribute name state* says:

U+002F SOLIDUS (`/`)
Switch to the self-closing start tag state.

OK. Now we consume the `"r"`:

Anything else
This is an unexpected-solidus-in-tag parse error. Reconsume in the before attribute name state.

So it will start a new attribute at this point:

Anything else
Start a new attribute in the current tag token. Set that attribute name and value to the empty string. Reconsume in the attribute name state.

The *attribute name state*, for both the "r" and the "e":

Anything else

Append the current input character to the current attribute's name.

The second "/" is then treated as follows:

U+0009 CHARACTER TABULATION (tab); U+000A LINE FEED (LF); U+000C FORM FEED (FF); U+0020 SPACE; U+002F SOLIDUS (/); U+003E GREATER-THAN SIGN (>); EOF

Reconsume in the after attribute name state.

After attribute name state:

002F SOLIDUS (/)

Switch to the self-closing start tag state.

Self-closing start tag state:

U+003E GREATER-THAN SIGN (>)

Set the self-closing flag of the current tag token. Switch to the data state.

Emit the current tag token.

Ah, this time the tokenizer got what it expected, a > after a slash.

The correct answer is `{src:"1.png", "re":""}`.

On a historical aside, Internet Explorer 6 (and maybe some other versions) had a special behavior for the `style` attribute, where it would just append to the list of CSS declarations if there were multiple `style` attributes, instead of dropping duplicate attributes (which it did for all other attributes). No other browser matched IE, though, so we could get away with not supporting this.

Another interesting aspect of Internet Explorer 6 was that it treated ` as a quote character around attribute values. Other browsers did not do this. This could easily result in differences in the resulting DOM. If you were using a conforming HTML parser to sanitize user input, but also have the serializer leave attribute values unquoted when they could be, it would open up a security hole to let the attacker insert script and have it run for visitors using IE. For example, maybe you would roundtrip entered values in a form:

```
<input name=first-name value=Sam>
<input name=last-name value=Sneddon>
```

Now consider if Sam enters "" as the first name and `` autofocus onfocus=alert(document.cookie) " as the last name:

```
<input name=first-name value=`>
<input name=last-name value=""` autofocus onfocus=alert(document.cookie)
">
```

Oops. Yes, little gsnedders autofocus, [we call them](#).

To avoid this, the HTML standard [made ` in unquoted attribute values a parse error](#) in 2009.

Character references

Let's do a simpler one this time. How many named entities (", & and so on) are there in HTML? #HTMLQuiz

- 0..50
- 50..200
- 200..1000
- 1000..10000

XML has [5 predefined named entities](#): & < > " '.

HTML 4.01 had [252 named entities](#), but ' was not one of them.

HTML5 added ', which already worked in browsers except for IE, plus a few all-uppercase variations, like < and a bunch of non-conforming without the semicolon, like sp.

Then, as part of adding MathML to HTML in 2008, all of the MathML named entities were added to HTML. In total, the number is now 2231.

Two named character references have changed what they expand to since HTML 4.01: ⟨ and ⟩. The following [email](#) from Ian Hickson, from 2 March 2008, summarises what happened:

On Sun, 1 Jul 2007, Øistein E. Andersen wrote:

HTML5 currently maps `⟨` and `⟩` to U+3008 LEFT ANGLE BRACKET, U+3009 RIGHT ANGLE BRACKET, both belonging to 'CJK angle brackets' in U+3000--U+303F CJK Symbols and Punctuation.

HTML 4.01 maps them to U+2329 LEFT-POINTING ANGLE BRACKET, U+232A RIGHT-POINTING ANGLE BRACKET from 'Angle brackets' in the range U+2300--U+23FF Miscellaneous Technical.

Unicode 5.0 notes:

These are discouraged for mathematical use because of their canonical equivalence to CJK punctuation.

It would probably be better to use U+27E8 MATHEMATICAL LEFT ANGLE BRACKET, U+27E9 MATHEMATICAL RIGHT ANGLE BRACKET from 'Mathematical brackets' in U+27C0--U+27EF Miscellaneous Mathematical Symbols-A, characters that did not yet exist when HTML 4.01 was published.

I've made this change.

So now these map to the correct mathematical angle brackets that didn't exist in Unicode when HTML 4.01 was written.

An interesting aspect is parsing of named character references that lack the trailing semicolon. The parser will expand them to the corresponding character even when the next character is an alphanumeric.

`Arts&crafts`

Is equivalent to:

`Arts&crafts`

The interesting part is dealing with the `&not` and `&notin` character references. How does the parser know whether to expand the `&not` character reference if the next character is an "i"? The spec has the following example:

If the markup contains (not in an attribute) the string `I'm &notit; I tell you`, the character reference is parsed as "not", as in, `I'm -it; I tell you` (and this is a parse error). But if the markup was `I'm ∉ I tell you`, the character reference would be parsed as "notin;", resulting in `I'm ∉ I tell you` (and no parse error).

However, if the markup contains the string `I'm &notit; I tell you` in an attribute, no character reference is parsed and string remains intact (and there is no parse error).

The parser will consume characters in the *Named character reference state* so long as there is a prefix match for a name in the table of named character references, until there's only one match, or no match.

The example above touches on named character references in attributes, which is parsed slightly differently. If there is a match, in an attribute value, the spec says to treat it as follows:

If the character reference was consumed as part of an attribute, and the last character matched is not a U+003B SEMICOLON character (;), and the next input character is either a U+003D EQUALS SIGN character (=) or an ASCII alphanumeric, then, for historical reasons, flush code points consumed as a character reference and switch to the return state.

So the following would not expand the named character reference, even though it does outside attribute values:

```
<input value="Arts&ampcrafts">
```

This matched what IE did, but this was not interoperable back in 2006. One aspect that the standard now requires but that IE did not do, was to not expand the character reference if the next character is "=". It was added based on [web compatibility research](#) that I did in 2009 that found that web pages typically expected such cases to not expand the character reference.

On Sun, 14 Jun 2009 21:25:39 +0200, Ian Hickson <ian@hixie.ch> wrote:

It might still be reasonable to change the parsing rules to make the above case less surprising:

3. Tweak the parsing rules so that = is treated the same as 0-9a-zA-Z.

It would be different from what IE does, but I would be surprised if Web compat requires the IE behavior here.

I'd really like to not risk changes to the parsing rules in this area. It took a lot of careful study to get to where we are now, and without repeating that work, I'd be very reluctant to experiment.

Data:

<https://philip.html5.org/data/entities-without-semicolon-followed-by-equals.txt>

The ones below are those that would be affected by this change. This is 50 occurrences out of 425K pages.

As far as I can tell, all of these seem to expect the literal text treatment rather than the entity treatment.

Typically, pages would have unescaped ampersands in URLs, like this:

```
<script src="_fuse/1/elements.asp?RD=2&GT=627&Regen=78"></script>
```

Note that **&GT** is a named character reference.

Apart from named character references, there are also numeric character references.

```
&#65;  
&#x41;
```

The first one is a decimal character reference, the second one is hexadecimal. They both map to the character "A". The hexadecimal form is case-insensitive, including the "x". The semicolon is required for authors, but the parser will infer it if it's missing. If the number is 0, or outside Unicode's range (greater than 0x10FFFF), or if it's in the surrogate range (0xD800 to 0xDFFF), then it will expand to the replacement character (U+FFFD), and it will also be a parse error.

A difference from HTML 4.01, and from XML for that matter, is what some of the numerical character references map to, that would otherwise map to control characters. The HTML standard has this mapping table for numeric character references:

Number	Code point	Character name
0x80	0x20AC	EURO SIGN (€)
0x82	0x201A	SINGLE LOW-9 QUOTATION MARK (,))
0x83	0x0192	LATIN SMALL LETTER F WITH HOOK (ƒ)
0x84	0x201E	DOUBLE LOW-9 QUOTATION MARK („)
0x85	0x2026	HORIZONTAL ELLIPSIS (...)
0x86	0x2020	DAGGER (†)
0x87	0x2021	DOUBLE DAGGER (‡)
0x88	0x02C6	MODIFIER LETTER CIRCUMFLEX ACCENT (^)
0x89	0x2030	PER MILLE SIGN (‰)
0x8A	0x0160	LATIN CAPITAL LETTER S WITH CARON (Š)
0x8B	0x2039	SINGLE LEFT-POINTING ANGLE QUOTATION MARK (‹)
0x8C	0x0152	LATIN CAPITAL LIGATURE OE (Œ)
0x8E	0x017D	LATIN CAPITAL LETTER Z WITH CARON (Ž)
0x91	0x2018	LEFT SINGLE QUOTATION MARK (')
0x92	0x2019	RIGHT SINGLE QUOTATION MARK (')
0x93	0x201C	LEFT DOUBLE QUOTATION MARK (")
0x94	0x201D	RIGHT DOUBLE QUOTATION MARK (")
0x95	0x2022	BULLET (•)
0x96	0x2013	EN DASH (–)
0x97	0x2014	EM DASH (—)
0x98	0x02DC	SMALL TILDE (~)
0x99	0x2122	TRADE MARK SIGN (™)
0x9A	0x0161	LATIN SMALL LETTER S WITH CARON (š)
0x9B	0x203A	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK (›)
0x9C	0x0153	LATIN SMALL LIGATURE OE (œ)
0x9E	0x017E	LATIN SMALL LETTER Z WITH CARON (ž)
0x9F	0x0178	LATIN CAPITAL LETTER Y WITH DIAERESIS (ÿ)

Browsers had this mapping already when the HTML parser was specified, and there were web pages that relied on it. These are parse errors, however, so don't use them.

One final character reference that is a parse error is `` which maps to U+000D CARRIAGE RETURN. It's the only way to get such a character in the DOM from the parser; newlines are otherwise normalized to U+000A LINE FEED in the *preprocessing* stage.

Comments

Comments ought to be pretty simple; they start with `<!--` and end with `-->`, and that's that. Right?

Well, not quite. Both SGML and web browsers had different aspects of complexity for comments. Let's tackle the SGML story first. Ian Hickson wrote the following in [a blog post](#) in January 2006:

January 1999. I'm nineteen, in my first year studying Physics at Bath University. I read an SGML tutorial ([maybe this one](#) from 1995). I wrote a testcase. I filed a bug, in which I wrote:

Comment delimiters are "--" while inside tags.

Thus: `<!-- in -- -- in -- -- in -->` where "in" shows what is commented.

On the test page quoted, all is explained.

February 1999. The bug is fixed.

October 1999. The code for the fix is turned on along with the standards-mode HTML parser. Mozilla is now the first "major" browser to support SGML-style comments.

September 2000. The UN Web site breaks because it triggers standards mode but uses incorrect comment syntax. Mozilla drops full SGML comment parsing.

March 2001. Mozilla re-enables its strict comment parsing; evangelism is used to convince the broken sites to fix their markup.

May 2003. Netscape devedge publishes a document on the matter to help the Mozilla evangelists explain this to authors.

July 2003. I open a bug in the Opera bug database to get Opera to implement SGML comment parsing.

January 2004. I file another bug in the Opera bug database, having forgotten about the earlier one, to get Opera to implement SGML comment parsing.

February 2005. Håkon and I write the first draft of the Acid2 test.

March 2005. While giving a workshop on how to create test cases at Opera, I find that <https://www.wassada.com/> renders correctly in Mozilla and fails to render in Opera precisely because Mozilla renders comments according to the SGML way and Opera doesn't. Over Håkon's objections, I insist on including a test for the SGML comment syntax in Acid2, citing the Wassada site as proof that we need to get interoperability on the matter. Acid2 is announced.

April 2005. Safari fixes SGML comment parsing as part of their Acid2 work. Hyatt confesses bemusement regarding this feature, joining Håkon in thinking I was wrong to insist we include this part of the test.

June 2005. Konqueror fixes SGML comment parsing as part of their Acid2 work.

October 2005. Opera fixes SGML comment parsing as part of their Acid2 work, after many complaints internally telling me I was wrong to include this part of the test. I point to the Wassada site. They point to the dozens of sites that break because of this change. I point to the fact that they aren't broken in Mozilla. They realise their fix was not quite right, and make things work, but still grumble about it being stupid.

November 2005. Mark writes a long document explaining the SGML comment parsing mode. Håkon proposes removing this part of the test from Acid2. I point out that as long as the specs require this, we don't have a good reason to remove it from the test.

December 2005. Prince implement SGML comment parsing in their efforts to pass Acid2, but privately raise concerns about this parsing requirement.

January 2006. I realise I was wrong.

I've now fixed the spec and fixed the Acid2 test.

I'd like to apologise to everyone whose time I've wasted by insisting on following the specs on this matter for the past seven years. You probably number in the hundreds by now. Sometimes, the spec is wrong, and we just have to fix it. I'm sorry it took me so long to realise that this was the case here.

The SGML comment syntax might make more sense if you consider that it works the same in markup declarations in the DTD. For example, the following is the definition of the `param` element in the HTML 4.01 DTD:

```
<!ELEMENT PARAM - O EMPTY -- named property value --
<!ATTLIST PARAM
  id          ID          #IMPLIED -- document-wide unique id --
  name        CDATA       #REQUIRED -- property name --
  value       CDATA       #IMPLIED -- property value --
  valuetype   (DATA|REF|OBJECT) DATA -- How to interpret value --
  type        %ContentType; #IMPLIED -- content type for value
                                     when valuetype=ref --
>
```

OK, so we don't need to worry about the SGML comment syntax anymore.

What were browsers doing, then? They generally tried to keep it simple, with comments ending with "-->", but with a twist. If the browser reached the end of the input stream inside a comment, *it would rewind the input stream* and reparse the comment in a different tokenizer state that ends a comment at the first ">".

```
<!-- This > is a comment -->
<!-- Where > does this end?
```

Reparsing is something that was carefully avoided in the standard. Apart from being a problem for streaming parsers, it also presents a [security issue](#):

On Mon, 23 Jan 2006, Lachlan Hunt wrote:

I don't understand these security concerns. How is reparsing it after reaching EOF any different from someone writing exactly the same script without opening a comment before it? Won't the script be executed in exactly the same way in both cases?

The difference is that a sanitiser script would notice a `<script>` element, but would not notice the contents of a comment. Comments are considered safe, the publisher would not expect the contents of a comment to suddenly be invoked.

The comment could be, e.g.:

```
<!--
    Let's hope nobody ever manages to sneak this into our site
    through a
    cross-site scripting attack!:

    <script> doSomethingEvil(); </script>

    That would be terrible!

    Oh well. There's no way they could aCONNECTION TERMINATED BY
    PEER
```

Browsers also did reparsing in some other situations, such as an unclosed `title` element and, in particular, an unclosed `<!--` in a script element (which looks like a comment but is actually text). More on this in the [Script states](#) section. Switching to not doing reparsing was not without facing web compatibility problems. In March 2008, I sent the following [email](#) to the public-html mailing list:

We were fixing our bugs regarding reparsing, but were a bit scared to fix reparsing of comments and escaped text spans, so I asked in #whatwg if someone could be kind enough to provide some data on the matter...

Philip` found 128 pages with open `<!--` out of 130K pages, listed in <https://philip.html5.org/data/pages-with-unclosed-comments.txt>. I looked through the first 82 pages. 40 of those would work better if we reparse, 1 would work slightly worse, and the rest would be unaffected. This means that about 0.05% of pages would break if we didn't reparse.

Opera currently doesn't reparse comments in limited/no quirks mode, but a few pages below break in Opera because of that. (We still reparse open escaped text spans even in no quirks mode.)

Also found during this research was that a lot of pages use `--!>` and expect it to close the comment. `--!>` closes comments in WebKit and Gecko. We'll probably make `--!>` close comments given this data.

We will probably not stop reparsing comments (in quirks mode) or escaped text spans (at least for script and style), at least not until other browsers do so. Maybe we can limit reparsing of escaped text spans to quirks mode, but we don't particularly like parsing differences between modes.

(We will come back to "escaped text spans" in the [Script states](#) section.)

To counter the web compatibility problems, the string `--!>` was added as a way the parser can close a comment. Reparsing was not specified, but browsers continued to do that (until they rewrote their parsers).

An IEism that was adopted in the standard was that `<!-->` and `<!-->` represent empty comments. That is, the dashes in the `<!--` can overlap the dashes in the `-->`.

Bogus comments

Have you ever seen an HTML page with an XML declaration at the top?

```
<?xml version="1.0"?>
<!DOCTYPE html>
```

If so, then you have stumbled across a "bogus comment". In HTML, some things cause the tokenizer to switch to the *bogus comment state*, which looks for the first `>` to end the comment (rather than `-->` or `--!>`). The following is thus equivalent:

```
<!--?xml version="1.0"?-->  
<!DOCTYPE html>
```

Apart from `<?`, the sequence `</` followed by something that is not `a-zA-Z`, or `<!` that is not followed by **doctype** (case-insensitive) or `--` or, in foreign content, `[CDATA[` (case-sensitive), starts a bogus comment.

Doctypes

There are 16 tokenizer states dedicated to doctypes, not including the *tag open state* (`<`) or the *markup declaration open state* (`<!`):

- DOCTYPE state
- Before DOCTYPE name state
- DOCTYPE name state
- After DOCTYPE name state
- After DOCTYPE public keyword state
- Before DOCTYPE public identifier state
- DOCTYPE public identifier (double-quoted) state
- DOCTYPE public identifier (single-quoted) state
- After DOCTYPE public identifier state
- Between DOCTYPE public and system identifiers state
- After DOCTYPE system keyword state
- Before DOCTYPE system identifier state
- DOCTYPE system identifier (double-quoted) state
- DOCTYPE system identifier (single-quoted) state
- After DOCTYPE system identifier state
- Bogus DOCTYPE state

The reason is that the doctype used to have more stuff in it than just `<!doctype html>`. This is the doctype for HTML 4.01:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

There's the doctype name (**HTML**), the keyword **PUBLIC** (which could also be **SYSTEM**), the public identifier (`-//W3C//DTD HTML 4.01//EN`), and the system identifier (`http://www.w3.org/TR/html4/strict.dtd`). (In SGML, the public and system identifiers both identify a DTD. The public identifier would be used by an SGML parser to look up a local DTD in a catalog.)

Since the doctype is used for [determining rendering mode](#), and since the strings are exposed in the DOM, the tokenizer can't just skip to the first ">" and then emit the token; it needs to collect the public and system identifiers.

What happens if you have garbage in the doctype? It depends on where that garbage is; stuff after the system identifier is silently ignored. Unexpected characters elsewhere will set the *force-quirks flag* and switch to the *bogus DOCTYPE state*, which looks for a ">" to end the doctype.

Using the "PUBLIC" or "SYSTEM" keywords but omitting the strings will set the *force-quirks flag*.

CDATA sections

CDATA sections are only supported in foreign content, i.e., when the current node is an SVG or MathML element. The effect is that text between the `<![CDATA[` and `]]>` markers are treated as text rather than as markup, so you can use `&` and `<` without escaping them as character references.

The *markup declaration open state* says:



Case-sensitive match for the string "[CDATA[" (the five uppercase letters "CDATA" with a U+005B LEFT SQUARE BRACKET character before and after)
Consume those characters. If there is an adjusted current node and it is not an element in the HTML namespace, then switch to the CDATA section state. Otherwise, this is a *CDATA-in-html-content* parse error.
Create a comment token whose data is the "[CDATA[" string. Switch to the bogus comment state.

So in HTML content, it ends up as a comment instead.

Where CDATA sections *are* supported, the tokenizer emits normal character tokens for the text. This means that such text ends up being normal **Text** nodes in the DOM, rather

than `CDATASection` nodes, which the DOM also has.

As part of writing this book, I found a [bug](#) in Safari and Chrome: CDATA sections are not supported in HTML integration points or MathML text integration points (more on this in [The foreign lands: SVG and MathML](#)). So avoid using it in, e.g., the SVG `title` element.

Most likely, the only case you will see CDATA sections in HTML is in the SVG `script` element, where it is supported in all browsers.

RCDATA, RAWTEXT and PLAINTEXT states

When the tree builder sees certain start tag tokens, it will switch state of the tokenizer to RCDATA, RAWTEXT, or PLAINTEXT.

Those start tag tokens are:

- RCDATA: `title`, `textarea`.
- RAWTEXT: `style`, `xmp`, `iframe`, `noembed`, `noframes`, `noscript` (if scripting is enabled).
- PLAINTEXT: `plaintext`.

When in the RCDATA state, the tokenizer still supports character references, but will not enter the *tag open state* when seeing a `<`; instead it switches to the *RCDATA less-than sign state*. It will continue through a path of RCDATA-specific states to tokenize an end tag. If a matching end tag is found, it will create an end tag token. Otherwise, it will emit the consumed characters as character tokens, and switch back to the RCDATA state.

The RAWTEXT state is similar to RCDATA, except that character references are not supported.

The PLAINTEXT state is similar to RAWTEXT, except that it can never switch to any other state:

Consume the next input character:

U+0000 NULL

This is an unexpected-null-character parse error. Emit a U+FFFD REPLACEMENT CHARACTER character token.

EOF

Emit an end-of-file token.

Anything else

Emit the current input character as a character token.

Effectively, the rest of the document is unconditionally treated as plain text.

Script states

Another #HTMLQuiz (don't cheat :)). What will be alerted here?

```
<script>alert('<!--<script>x</script>-->')</script>
```

- [nothing; syntax error]
- [empty string]
- x
- <!--<script>x</script>-->

Tokenizing script elements is complicated. The following states govern how to tokenize script elements:

- Script data state
- Script data less-than sign state
- Script data end tag open state
- Script data end tag name state
- Script data escape start state
- Script data escape start dash state
- Script data escaped state
- Script data escaped dash state
- Script data escaped dash dash state
- Script data escaped less-than sign state
- Script data escaped end tag open state
- Script data escaped end tag name state
- Script data double escape start state
- Script data double escaped state

- Script data double escaped dash state
- Script data double escaped dash dash state
- Script data double escaped less-than sign state
- Script data double escape end state

"Script data double escaped dash dash state"?! Well, let's start from the beginning, shall we?

Before the HTML parser was specified, browsers would parse script elements similarly to how they parse style elements. It was raw text, until the right end tag was found. But there was a twist: what looked like an HTML comment, i.e., `<!-- -->`, would allow `</script>` to be embedded inside without closing the script. (And similarly for style, and also xmp, title, textarea, etc.)

Now, there was also a complication on top of that. Remember how unclosed comments would cause browsers to rewind the input stream and reparse in a different state? That applied here as well. If you reached end-of-file while in an "escaped text span", browsers would rewind and close the script on the first script end tag even if it was after a `<!--`.

As it turns out, there were web pages that relied on both of these features.

In June 2007, Ian Hickson [specified](#) the first aspect, but not reparsing. The commit message said "Support the insane comment stuff in CDATA and RCDATA blocks". (CDATA was later renamed to RAWTEXT.)

In August 2009, Henri Sivonen sent an [email](#) to the public-html mailing list with the subject "Issues arising from not reparsing":

Firefox nightlies have had an HTML5 parser implementation behind a pref for a month now. The Web compat issues that have been uncovered have been surprisingly few, which is great.

However, there are three Web compat issues that don't have trivial fixes. They all are related to the HTML5 parsing algorithm not recovering from errors by rewinding the stream and reparsing with different rules. As such, if these are treated as bugs, they are spec bugs.

1. When the string `<!--` occurs inside a string literal in JavaScript, it starts and escape that hides `</script>` and the rest of the page is eaten into the script. https://bugzilla.mozilla.org/show_bug.cgi?id=503632
2. When a script starts with `<script><!--` but doesn't end with `-->` `</script>` (ends with only `</script>`), the rest of the page is eaten into

the script. https://bugzilla.mozilla.org/show_bug.cgi?id=504941

3. When there's no `</title>` end tag, the page gets eaten into the title.

https://bugzilla.mozilla.org/show_bug.cgi?id=508075 see also

https://bugs.webkit.org/show_bug.cgi?id=3905

https://bugzilla.mozilla.org/show_bug.cgi?id=42945

Personally, I'd like to avoid reparsing if at all possible, because it's a security risk and because it complicates the parser.

I did some research and [proposed a solution](#), which was adopted in the standard:

On Thu, 13 Aug 2009 09:26:39 +0200, Henri Sivonen <hsivonen@iki.fi> wrote:

On Aug 12, 2009, at 22:55, Ian Hickson wrote:

On Wed, 12 Aug 2009, Henri Sivonen wrote:

On Aug 12, 2009, at 12:10, Henri Sivonen wrote:

I think I'll create a wiki page with requirements and a proposed delta spec first, though, because others on #whatwg were interested in pondering alternative solutions given a set of requirements.

Wiki page created:

https://wiki.whatwg.org/wiki/CDATA_Escapes

Wow. Please can we stick to just the current magic escapes and not add even more magic?

The current magic without all the magic that current browsers implement lead to some incompatibilities with existing content. I don't know how often a user would hit these issues, but when the problems do occur, they wreck the whole page. Therefore, I think we should seriously try to improve the magic so that it substitutes the current browser magic better in practice while still not doing reparsing.

<https://philip.html5.org/data/script-open-in-escape.txt> has 622 pages.

<https://philip.html5.org/data/script-close-in-escape-without-script-open-2.txt>

has 708 pages.

Most of these look like they would break with what's currently specced.

The two sets might overlap. Some of the pages are not relevant, because the extract might appear inside an HTML comment. The breakage can be up to around 1300 pages out of 425000.

The common pattern is:

A.

```
<script><!--  
...  
//--></script>
```

However, there are several patterns that break with that is currently specced:

B.

```
<script><!--  
...  
</script>
```

C.

```
<script><!--  
...  
//-->  
<!--</script>
```

D.

```
<script><!--  
...  
//-- ></script>
```

E.

```
<script><!--  
...  
//- -></script>
```

F.

```
<script><!--  
...  
//- - ></script>
```

G.

```
<script><!--  
...  
//-></script>
```

etc.

where ... can be

1. `document.write('<script></script>');`
2. `document.write('<script></script><script></script>');`
3. `document.write('<script></script>');` `document.write('<script></script>');`
4. `document.write('<script>');` `document.write('</script>');`
5. `document.write('<scr'+ 'ipt></scr'+ 'ipt>');`
6. `document.write('<scr'+ 'ipt></script>');`
7. `document.write('<script></scr'+ 'ipt>');`

Proposal #3 in https://wiki.whatwg.org/wiki/CDATA_Escapes reads:

For script, when in an escaped text span, set a flag after having seen "<script" followed by whitespace or slash or greater-than. "</script" followed by whitespace or slash or greater-than only closes the element if the flag is not

set, and otherwise emits the text and resets the flag. Exiting an escaped text span also resets the flag.

It breaks with (6) combined with any of A-G. I found 3 sites doing this.

www.grandparents.com/gp/content/expert-advice/family-matters/article/thatevildaughterinlaw.html

www.celebrity-link.com/c106/showcelebrity_categoryid-10687.html

me.yaplog.jp/viewBoard.blog?boardId=975

It also breaks for (7) combined with B or D-G (note that what's currently specced also breaks here). I found 1 site doing this.

www.jeuxactu.com/images-fiche-soul-calibur-legends-8219-4-6.html

The sites appear to have one or two (or three) pages with the relevant script. This makes proposal #3 break for something on the order of 10 pages out of 425000. This is surprisingly close to the current behavior of doing reparsing. (Not reparsing leads to better performance since you don't need to wait for the whole page to have loaded before deciding where the script should end, and it doesn't have the security issue.)

I can't come up with a different proposal that breaks less pages.

In March 2010, in response to someone being confused about the script states, Henri Sivonen [wrote](#):

The purpose of those states is to support existing script content without ever rewinding the input stream.

The problem is basically this:

1. Some pages assume they can use the string "`</script>`" inside a script if they enclose the script content in `<!-- ... -->`
2. Other pages Have `<!--` at the start of the script but forget `-->` from the end.

Complexity ensues. So far, what's in the spec looks like a successful solution. I've implemented in experimentally in Gecko, and I haven't seen bug reports about it.

Going back to the quiz, the correct answer is that `<!--<script>x</script>-->` will be alerted. The tokens produced for the markup are (combining adjacent character tokens to a single token):

- Start tag (script)
- Characters (`<!--<script>x</script>-->`)
- End tag (script)
- End-of-file

OK. What about a more "problematic" case?

```
Before
<script><!--
document.write('<script></scr'+ 'ipt>');
</script>
After
```

After emitting the script start tag, the tree builder switches the tokenizer's state to the *script data state*. For the `<!--`, we step through these states:

- Script data less-than sign state
- Script data escape start state
- Script data escape start dash state
- Script data escaped dash dash state

The newline then switches to the *script data escaped state*. Then we stay in that state for the `"document.write("` part.

The nested `<script>` goes through these states:

- Script data escaped less-than sign state
- Script data double escape start state (stays in this state for the "script" characters)
- Script data double escaped state

OK, now the tokenizer is in a state that will ignore the next script end tag. Since the embedded end tag is split up, the tokenizer skips past it without changing state. Then, the script end tag goes through these states:

- Script data double escaped less-than sign state
- Script data double escape end state (stays in this state for the "script" characters)

- Script data escaped state

It then stays in the *script data escaped state* until the end-of-file. The tokens produced are:

- Characters (Before\n)
- Start tag (script)
- Characters (<!--\ndocument.write('<script></scr'+ 'ipt>');\n</script>\nAfter)
- End-of-file

The resulting DOM is:

```
#document
├── html
│   ├── head
│   └── body
│       ├── #text: Before
│       └── script
│           └── #text: <!-- document.write('<script></scr'+ 'ipt>');
</script> After
```

Since there is no actual script end tag, the script will not be executed. Executing scripts is, apart from constructing the DOM, part of the tree construction stage.

Tree construction

Parsing a simple document

Let's try a simple example and see what happens in the tree construction stage.

```
<!doctype html>
<div>Divitis is a serious condition.</div>
```

The tokenizer will produce these tokens:

- Doctype
- Character (\n)
- Start tag (div)

- Characters (`Divitis is a serious condition.`)
- End tag (`div`)
- End-of-file

The tree construction stage (or the *tree builder*) will then take the stream of tokens as its input, and mutate a **Document** object as its result.

There are a number of *insertion modes*, which govern how the tokens are handled. Initially, the insertion mode is the "initial" insertion mode (unsurprisingly). This insertion mode is the one that does something with doctype tokens (more on this in the next section, [Determining rendering mode](#)).

In this case, a **DocumentType** node is appended to the **Document**, and then the insertion mode is changed to "before html".

In the "before html" insertion mode, we handle the next token, the "\n" character token.

A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), U+000D CARRIAGE RETURN (CR), or U+0020 SPACE
Ignore the token.

OK, so whitespace after the doctype is ignored. Moving on.

The `div` start tag token is handled by the "before html" insertion mode as follows:

Anything else

Create an html element whose node document is the Document object.
Append it to the Document object. Put this element in the stack of open elements.

If the Document is being loaded as part of navigation of a browsing context, then: run the application cache selection algorithm with no manifest, passing it the Document object.

Switch the insertion mode to "before head", then reprocess the token.

Notice the reference to the *stack of open elements*. This stack is used throughout the tree builder, for example when handling an end tag token. When an element is inserted, it is also added to the stack of open elements.

We'll gloss over the application cache stuff, as it's not significant to parsing and the application cache feature is in the process of being removed anyway.

We then switch the insertion mode to "before head" and process the same token again. That insertion mode will insert a **head** element and switch to "in head" and reprocess the token. That insertion mode will pop the **head** element off the stack of open elements, switch to "after head", and again reprocess the same div start tag token. *That* insertion mode will insert a **body** element, switch to "in body", and, you guessed it, reprocess the token.

At this point the DOM looks like this:

```
#document
├── DOCTYPE: html
└── html
    ├── head
    └── body
```

The "in body" insertion mode is the mode that handles most of the tags in a typical document. Let's see what it does with the **div** start tag token:

A start tag whose tag name is one of: "address", "article", "aside", "blockquote", "center", "details", "dialog", "div", "dl", "fieldset", "figcaption", "figure", "footer", "header", "hgroup", "main", "menu", "nav", "ol", "p", "section", "summary", "ul"

If the stack of open elements has a p element in button scope, then close a p element.

Insert an HTML element for the token.

The stack of open elements has just html and body, so there's no p element to close. (We'll discuss the details of this in the [Implied tags](#) section.)

"Insert an HTML element" will insert a div element, and push it to the stack of open elements. The stack is now: html, body, div. The DOM is:

```
#document
├── DOCTYPE: html
└── html
    ├── head
    └── body
        └── div
```

We stay in the "in body" insertion mode. Next, we have a character token (D).

Any other character token

Reconstruct the active formatting elements, if any.

Insert the token's character.

Set the frameset-ok flag to "not ok".

Reconstructing active formatting elements is discussed in the [Misnested tags](#) section.

"Insert the token's character" will check if there is a **Text** node immediately before, and if so, append to it. Otherwise it creates a new **Text** node. In this case, there is no **Text** node yet, but for the subsequent character tokens there is.

The frameset-ok flag is discussed in the [Frameset](#) section.

Finally, the end tag.

An end tag whose tag name is one of: "address", "article", "aside", "blockquote", "button", "center", "details", "dialog", "dir", "div", "dl", "fieldset", "figcaption", "figure", "footer", "header", "hgroup", "listing", "main", "menu", "nav", "ol", "pre", "section", "summary", "ul"

If the stack of open elements does not have an element in scope that is an HTML element with the same tag name as that of the token, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not an HTML element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an HTML element with the same tag name as the token has been popped from the stack.

The stack of open elements is still: html, body, div. So we run the steps above.

Step 2 means that, if you had, e.g., `<div><span></div>`, then it would be a parse error when handling the div end tag token. Step 3 means that elements are closed until a div is closed.

In our case, the most recently added element is indeed a div, so there's no parse error, and we pop it off the stack.

Are we done? Not quite. There is an end-of-file token, too.

An end-of-file token

If the stack of template insertion modes is not empty, then process the token using the rules for the "in template" insertion mode.

Otherwise, follow these steps:

1. If there is a node in the stack of open elements that is not either a dd element, a dt element, an li element, an optgroup element, an option element, a p element, an rb element, an rp element, an rt element, an rtc element, a tbody element, a td element, a tfoot element, a th element, a thead element, a tr element, the body element, or the html element, then this is a parse error.
2. Stop parsing.

The template stuff is about handling unclosed template elements.

Step 1 says that there is a parse error if there's still an open element except for those that have optional end tags.

"Stop parsing" will, among other things, execute deferred scripts, fire the **DOMContentLoaded** event, and wait until there is nothing that *delays the load event* (like images), at which point it will fire the **load** event on the Window.

And now we're done.

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       └── div
│           └── #text: Divitis is a serious condition.
```

Determining rendering mode

[#HTMLQuiz](#) (don't cheat :)) Which doctype does not trigger quirks mode?
+@RRverser <!DOCTYPE ...

- YOLO>
- HTML SYSTEM>
- HTML PUBLIC "" "" ROFL>

- `HTML PUBLIC "HTML" "LOL">`

The doctype determines the document's rendering mode.

The following cases result in the document using quirks mode:

- The token's *force quirks flag* is set. (See the [Doctypes](#) section.)
- The token's name is not `"html"`.
- A list of 61 comparisons of the public identifier and sometimes the system identifier, compared case-insensitively. Here is a subset of the list:
 - The public identifier is set to: `"-//W3O//DTD W3 HTML Strict 3.0//EN/"`
 - The public identifier is set to: `"-//W3C//DTD HTML 4.0 Transitional//EN"`
 - The public identifier is set to: `"HTML"`
 - The system identifier is set to:
`"http://www.ibm.com/data/dtd/v11/ibmhtml1-transitional.dtd"`
 - The public identifier starts with: `"+//Silmaril//dtd html Pro v0r11 19970101/"`
 - The public identifier starts with: `"-//AS//DTD HTML 3.0 asWedit + extensions/"`
 - The public identifier starts with: `"-//AdvaSoft Ltd//DTD HTML 3.0 asWedit + extensions/"`
 - The public identifier starts with: `"-//IETF//DTD HTML 2.0 Level 1/"`
 - The public identifier starts with: `"-//IETF//DTD HTML 2.0 Level 2/"`
 - The public identifier starts with: `"-//IETF//DTD HTML 2.0 Strict Level 1/"`
 - ...

The following cases trigger limited quirks mode:

- The public identifier starts with: `"-//W3C//DTD XHTML 1.0 Frameset/"`
- The public identifier starts with: `"-//W3C//DTD XHTML 1.0 Transitional/"`
- The system identifier is not missing and the public identifier starts with:
`"-//W3C//DTD HTML 4.01 Frameset/"`

- The system identifier is not missing and the public identifier starts with:
"-//W3C//DTD HTML 4.01 Transitional//"

If the document is "an iframe srcdoc document", then the mode is no-quirks mode regardless of the doctype, and the doctype is optional in such a document.

Other doctypes leave the rendering mode as no-quirks mode.

Note that most comparisons for the public identifier uses a prefix match instead of comparing the full string. The reason for this is that web pages sometimes (around [0.1% of pages, in 2008](#)) changed the "-//EN" to the language code for the *page*, but it is supposed to be a language code for the DTD. Internet Explorer and Opera used to have "lax" comparison of the public identifier, while Safari and Firefox compared the full string. The pages that changed the "-//EN" to something else usually expected quirks mode rendering.

Let's go back to the quiz. <!DOCTYPE YOLO> triggers quirks mode because the name is not html. <!DOCTYPE HTML SYSTEM> triggers quirks mode since the *force-quirks flag* is set by the tokenizer. <!DOCTYPE HTML PUBLIC "HTML" "LOL"> triggers quirks mode because HTML is in the list of public identifiers that trigger quirks mode.

What about <!DOCTYPE HTML PUBLIC "" "" ROFL>? The empty string is different from absent public and system identifier, and is not in the list of things that trigger quirks mode. Trailing garbage in the doctype does not trigger the *force-quirks flag*. So it does not trigger quirks mode. The reason trailing garbage is ignored is that some web developers (about [0.02% of pages](#)) were overzealous in converting to XHTML that they thought the doctype ought to have a trailing slash as well.

Noscript

[#HTMLQuiz](#) Which tag is implied where when scripting is *disabled*? <head>
<noscript><basefont><noscript><base>

- <body> before <noscript>
- </noscript> before <basefont>
- </noscript> before <noscript>
- </noscript> before <base>

noscript is parsed differently when scripting is enabled and when it is disabled.

When scripting is enabled, the tree builder, when handling the start tag token, switches the tokenizer's state to the *RAWTEXT state*. So it is parsed the same as, e.g., the `style` element.

When scripting is disabled, how `noscript` is parsed depends on if it is in **head** or in **body**. Let's do the in **body** case first, since it is simpler. It is parsed the same as "ordinary" elements (which includes unknown elements):

Any other start tag

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Note: This element will be an ordinary element.

That is, it is inserted in the DOM and the contents are parsed as normal. For example:

```
<!doctype html>
<body><noscript><p>This page requires JavaScript.</p></noscript></body>
```

Resulting DOM:

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       └── noscript
│           └── p
│               └── #text: This page requires JavaScript.
```

When `noscript` is found in **head**, the tree builder switches to the "in head noscript" insertion mode. Walking through the example from the quiz:

```
<head><noscript><basefont><noscript><base>
```

basefont is handled as follows in this insertion mode:

A character token that is one of U+0009 CHARACTER TABULATION, U+000A LINE FEED (LF), U+000C FORM FEED (FF), U+000D CARRIAGE RETURN (CR), or U+0020 SPACE; A comment token; A start tag whose tag name is one of: "basefont", "bgsound", "link", "meta", "noframes", "style"

Process the token using the rules for the "in head" insertion mode.

The result is that a **basefont** element is inserted to the **noscript** element.

What about the second **noscript** start tag?

A start tag whose tag name is one of: "head", "noscript"; Any other end tag
Parse error. Ignore the token.

It's ignored. The final tag, the **base** start tag, is handled under the anything else clause:

Anything else

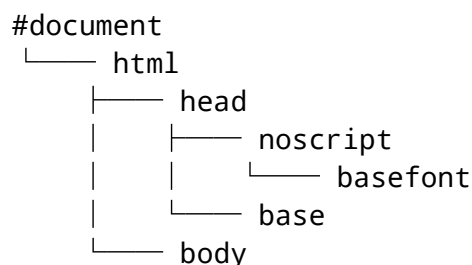
Parse error.

Pop the current node (which will be a noscript element) from the stack of open elements; the new current node will be a head element.

Switch the insertion mode to "in head".

Reprocess the token.

It closes the **noscript** element, and the token is reprocessed. So the correct answer is "</noscript> before <base>".



The takeaway is that, out of the conforming elements in HTML, you can only use **link**, **meta**, and **style** in **noscript** in **head**. The typical use case is including a different stylesheet when scripting is disabled.

Back in 2007, when **noscript** in head was [specified](#), browsers did different things (of course). In Firefox, **noscript** in **head** would imply a **<body>** start tag before it. In IE, the **noscript** element would be inserted in the **head**, but if it contained something that was not allowed in **head** (like an "X" character), then it would create an ill-formed DOM. Opera instead didn't insert any **noscript** element to the DOM. Safari changed in 2007 to allow **noscript** in **head**, and the specification was updated as a result, although what Safari did was different to what the specification said.

Frameset

Frameset is a feature that was introduced in HTML4 and immediately deprecated (and is now obsolete). It's like the `iframe`, but the whole page is a set of frames, in rows and columns. Such pages do not have a `body` element; instead they have a `frameset` element.

A frameset page might look like this:

```
<!doctype html>
<html lang="en">
  <head>
    <title>Framed art</title>
  </head>
  <frameset rows="150,*">
    <frame src="header.html">
    <frameset cols="30%,*">
      <frame src="nav.html">
      <frame src="main.html">
    </frameset>
  </frameset>
</html>
```

If the tree builder finds a `frameset` start tag token in the "after head" insertion mode, then it creates a "frameset" page. But it's also possible for the parser to have inserted a `body` element, and later swapping it for a `frameset` element.

```
<!doctype html>
<p><frameset>Who am I?
```

```
#document
├── DOCTYPE: html
└── html
    ├── head
    └── frameset
        └── #text:
```

How does the parser decide if the page is a "frameset" page or a "body" page?

You may recall from the [Parsing a simple document](#) section a mention of a frameset-ok flag. This flag determines whether, upon finding a `frameset` start tag token in the "in body" insertion mode, the page should be a frameset page.

The following things set the frameset-ok flag to "not ok".

- A character token that is not U+0000 or ASCII whitespace.

- One of these start tags: `pre`, `listing`, `li`, `dd`, `dt`, `button`, `applet`, `marquee`, `object`, `table`, `area`, `br`, `embed`, `img`, `keygen`, `wbr`, `input`, `hr`, `textarea`, `xmp`, `iframe`, `select`.
- A `br` end tag.

If the flag is "not ok", then `frameset` start tags are ignored. Otherwise, the parser removes the `body` element and its children from the DOM and inserts a `frameset` element, and switches the insertion mode to "in frameset". This insertion mode only inserts elements for `frameset`, `frame`, and `noframes` start tag tokens, and only inserts `Text` nodes for ASCII whitespace character tokens. Everything else is dropped on the floor.

Forms

Forms have some unusual behaviors.

Form controls, such as the `input` element, are associated with a `form` element. This association is used in form submission. There is also an API to access all elements that are associated with a form (`form.elements`).

```
<!doctype html>
<form><input></form>
<script>
console.assert(document.forms[0].elements[0] ===
                document.querySelector('input'));
</script>
```

That's cool, but what does it have to do with parsing? Can't the relationship just be based on the ancestor elements in the DOM?

It turns out that it can't. The association needs to happen even if the form element is not an ancestor of the form control when the form control is parsed. So long as the form end tag hasn't been seen, form controls will be associated with an "open" form, even if it is no longer on the stack of open elements.

```
<!doctype html>
<div><form></div>
<input>
```

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   └── body
│       ├── div
│       └── form
```

```

└── #text:
└── input

```

The **input** is not a descendant of the **form**, but it is still associated with it. How?

The parser has a *form element pointer*, which is set to the **form** element when handling a **form** start tag token (if it creates an element). This pointer is only reset to null when seeing an explicit **form** end tag, even if it is in the "wrong" place.

The parser ignores a **form** start tag token if the *form element pointer* is not null. That is, nesting forms doesn't work.

```

<form>
A
<div>
B
<form></form>
C
</div>
D
</form>

```

This results in this DOM:

```

#document
└── html
    ├── head
    └── body
        ├── form
        │   ├── #text: A
        │   └── div
        │       └── #text: B C
        └── #text: D

```

There's only one form element in the DOM, but otherwise the DOM is as we'd expect—wait, why is the "D" text node a child of **body**, and not the **form**? The "C" is in the same text node as the "B", so the **form** end tag didn't close the **div** and the **form**. What happened?

Let's back up a bit. Up to and including the "B", parsing is straightforward.

```

<form>
A
<div>
B

```

Then we have the **form** start tag. Since the *form element pointer* is not null, we ignore that tag. Very well. What about the inner **form end** tag?

Let's see.

An end tag whose tag name is "form"

If there is no template element on the stack of open elements, then run these substeps:

1. Let *node* be the element that the form element pointer is set to, or null if it is not set to an element.
2. Set the form element pointer to null.
3. If *node* is null or if the stack of open elements does not have *node* in scope, then this is a parse error; return and ignore the token.
4. Generate implied end tags.
5. If the current node is not *node*, then this is a parse error.
6. Remove *node* from the stack of open elements.

We clear the form element pointer, step 3 doesn't apply (*node* is the form), and we don't have any implied end tags to generate. Step 5 applies since the current node is a **div**.

The stack of open elements is:

- **html**
- **body**
- **form**
- **div**

In step 6 we remove the **form**, so that the stack of open elements is:

- **html**
- **body**
- **div**

Huh, it didn't remove the **div**! Usually, when the parser closes an element, it pops elements off the stack until the relevant element has been popped. But here, only the **form** is removed, leaving the rest of the stack intact. So the current node is still the **div**. When we get to the "C", we thus insert that to the **div**.

```
<form>
A
<div>
```

```
B
<form></form>
C
```

```
#document
├── html
│   ├── head
│   └── body
│       └── form
│           ├── #text: A
│           └── div
│               └── #text: B C
```

Next, we find the `div` end tag. Since the current node is a `div`, this will just pop the `div` off the stack of open elements:

- `html`
- `body`

At this point, the current node is the `body`, which is where the "D" ends up being inserted.

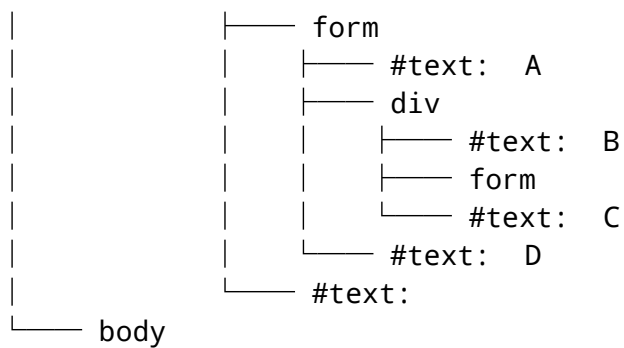
This special handling is, as you might suspect, necessary for web compatibility. The specification [used to](#) handle `form` end tags like `div` end tags, but [it was found](#) to break web sites, so it was [changed](#) in December 2008 to what it says now.

Did you notice that the handling of the `form` end tag had a check for a `template` element? What happens inside `templates`?

```
<template>
  <form>
    A
    <div>
      B
      <form></form>
      C
    </div>
    D
  </form>
</template>
```

The document's DOM, and the `template` element's *template contents* (more on this in the [Templates](#) section), are as follows:

```
#document
├── html
│   ├── head
│   └── template
│       └── #document-fragment (template contents)
│           └── #text:
```



There's a nested **form**! And the "D" Text node is where we'd expect (child of the outer form).

In **templates**, forms are parsed more like **divs**, and aren't using the form element pointer.

Tables and foster parenting

#HTMLQuiz In which order will the numbers appear for such bad HTML?

```
<table><tr><td>1</td></tr>2<br/><tr>3</tr>
```

- 1; 3; 2
- 2; 3; 1
- 1; 2; 3
- 2; 1; 3

Unexpected text or tags in tables (outside table cells) are, for historical reasons, placed *before* the table. This is called foster parenting.

A simple example:

```
<table>1
```

```

#document
├── html
│   ├── head
│   └── body
│       ├── #text: 1
│       └── table
  
```

So how does this happen? Let's step through the spec.

First, we parse the table start tag. We insert it as normal and switch to "in table".

```
#document
├── html
│   ├── head
│   └── body
│       └── table
```

Then we get a character token in the "in table" insertion mode.

A character token, if the current node is **table**, **tbody**, **tfoot**, **thead**, or **tr** element

Let the pending table character tokens be an empty list of tokens.

Let the original insertion mode be the current insertion mode.

Switch the insertion mode to "in table text" and reprocess the token.

Reprocessing in "in table text":

Any other character token

Append the character token to the pending table character tokens list.

So this collects all of the character tokens in a list. The reason for this is that, if the characters are all whitespace, then it shouldn't be foster-parented, but if there is any non-whitespace, then all those character tokens should be foster-parented together (consider spaces between words).

The next token is end-of-file:

Anything else

If any of the tokens in the pending table character tokens list are character tokens that are not ASCII whitespace, then this is a parse error: reprocess the character tokens in the pending table character tokens list using the rules given in the "anything else" entry in the "in table" insertion mode.

Otherwise, insert the characters given by the pending table character tokens list.

Switch the insertion mode to the original insertion mode and reprocess the token.

OK, so we need to check what "anything else" in "in table" says.

Anything else

Parse error. Enable foster parenting, process the token using the rules for the "in body" insertion mode, and then disable foster parenting.

Aha, this says something about foster parenting! The rules in "in body" for a non-U+0000, non-ASCII whitespace character token, is to "[insert the token's character](#)", which is an algorithm which calls into another algorithm, for finding the "[appropriate place for inserting a node](#)":

If foster parenting is enabled and target is a **table**, **tbody**, **tfoot**, **thead**, or **tr** element

[...]

If last table has a parent node, then let adjusted insertion location be inside last table's parent node, immediately before last table, and abort these substeps.

This is the part that says to insert the node before the table.

The example in the quiz is thus equivalent to:

```
2<br>3<table><tbody><tr><td>1</td></tr><tr></tr></tbody></table>
```

Notice that the **tbody** tags were not in the quiz, yet the above is equivalent. This is because, similarly to the **body** element, **tbody** has optional start and end tags. It will be inferred when handling a **tr** start tag token.

```
<table><tr><td>
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── table
│       │   ├── tbody
│       │   │   ├── tr
│       │   │   └── td
```

Although the **tr** element's start tag is *not* optional, the parser will still infer it if it is missing; the following markup produces the same DOM as the above.

```
<table><td>
```

The `colgroup` element also has optional start and end tags. Using a `col` start tag will imply a `colgroup` start tag before it.

All table-related elements except for the `table` element itself have optional end tags (including `</caption>`).

Table-related tags (except for `table` itself) are ignored outside tables (except in templates).

```
<body><caption>Tableless <tr>web <td>design
```

```
#document
├── html
│   ├── head
│   └── body
│       └── #text: Tableless web design
```

The last of the parsing quirks

[#HTMLQuiz](#) the HTML parser has a single difference in quirks mode compared to no-quirks. What is it?

- `<p>` can contain `<table>`
- `<h1></h2>` closes `h1`
- `--!>` closes a comment
- `<font color=chucknorris>`

In 2009, Henri Sivonen found that the HTML parser needed to retain a quirk for web compatibility. Here is his blog post with the timeline: [The Last of the Parsing Quirks](#) (copyright Mozilla Foundation, licensed under [CC BY 4.0](#); modified to omit links).

I implemented a single quirk for HTML5 parsing yesterday.

March 1995

Netscape 1.1 beta 1 is released with table support. Table closes a paragraph implicitly.

August 1995

Internet Explorer 1.0 is released. Table does not close a paragraph.

May 1996

The IETF publishes experimental RFC 1942 which says that table is block level content like paragraphs (i.e. closes paragraph implicitly).

January 1997

The W3C publishes HTML 3.2 with a DTD that makes tables close paragraphs implicitly in an SGML parser.

December 1997

The W3C publishes HTML 4.0 with a DTD that makes tables close paragraphs implicitly in an SGML parser.

April 1998

The W3C revises HTML 4.0 without changing the DTD on the point of paragraphs and tables.

December 1999

The W3C publishes HTML 4.01 without changing the DTD on the point of paragraphs and tables.

May 2000

IE5 for Mac is released. It is the first shipping browser to have a quirks mode and a standards mode.

July 2001

A bug is filed saying that Mozilla is wrong in not making table close a paragraph implicitly and that Mozilla should start closing paragraphs in the standards mode.

October 2001

IE6 is released. It is the first version of IE for Windows that has a quirks mode and a standards mode. A table doesn't close a paragraph in either mode.

June 2003

The Mozilla bug is fixed making Mozilla close paragraphs upon tables in the standards mode. The quirks-mode behavior is left to not closing a paragraph upon a table.

April 2005

The Web Standards Project publishes the Acid2 test made by Ian Hickson and Håkon Lie. To pass the test, a user agent must close a paragraph upon table (in the standards mode).

April 2005

In order to pass Acid2, Safari is changed to make a table close a paragraph in the standards mode. The quirks-mode behavior is left to not closing a paragraph upon a table.

May 2005

In order to pass Acid2, Opera is changed to make a table close a paragraph in the standards mode. The quirks-mode behavior is left to not closing a paragraph upon a table.

January 2006

Ian Hickson changes the comment parsing part of Acid2 and blogs about it.

February 2006

Ian Hickson publishes the first draft of the HTML5 parsing algorithm. It makes a table close a paragraph but the source code of the spec contains a comment saying "XXX quirks: don't do this".

November 2006

IE7 is released. A table doesn't close a paragraph in either mode.

March 2008

A preliminary version of IE8 passes Acid2 when hosted on **www.webstandards.org**.

February 2009

I file a spec bug requesting parsing quirks be defined.

March 2009

IE8 is released. It has four layout modes. To pass Acid2, the new ones make a table close a paragraph. The parser behavior of `<p><table>` is now the only HTML parsing difference between the quirks and standards modes that is interoperably implemented in all of the top 4 browser engines.

March 31st 2009

Ian Hickson asks for vendor input about parsing quirks.

April 1st 2009

I start a thread about finding vendor input in Mozilla's platform development newsgroup. The `<p><table>` issue seems to be the only quirk left.

April 1st 2009

Philip Taylor uses the Validator.nu HTML Parser to compile a list of dmoz-listed pages where closing paragraph vs. not closing would lead to different parse trees.

April 21st 2009

Simon Pieters analyzes 50 sites from Philip's list concluding that "our options regarding `<p><table>` parsing are (1) having the quirk, and (2) changing Acid2".

April 22nd 2009

I check in an implementation of the quirk into the Gecko HTML5 parsing repository.

May 26th 2009

Hixie checks in the definition of the quirk into the HTML5 spec. The commit also includes this comment: "i hate myself (this quirk was basically caused by acid2; if i'd realised we could change the specs when i wrote acid2, we could have avoided having any parsing-mode quirks) - Hixie"

A big thank you to Philip Taylor and Simon Pieters for their research (both the feasibility research and the timeline research).

Scripts

When seeing a **script** start tag, the tree builder switches the tokenizer's state to the *script data state*, and changes the insertion mode to "text" (which is also used for RAWTEXT and RCDATA elements).

When seeing the **script** end tag, the tree builder executes the script. The details of how this works is... complicated. See the [document.write\(\)](#) section of [Chapter 4. Scripting complications](#).

The parser will not continue parsing until the script has been downloaded (if applicable) and executed, and also until pending stylesheets have been loaded. See the [Blocking the parser](#) section.

But, if we ignore those things, then handling of the **script** end tag is easy. The **script** element is popped off the stack of open elements, and the insertion mode is switched back to what it was before entering the "text" insertion mode.

Templates

The **template** element ([added to HTML in June 2013](#)) is used to declare fragments of HTML that can be cloned and inserted in the document by script. The HTML standard has the following example:

For example, consider the following document:

```
<!doctype html>
<html lang="en">
  <head>
    <title>Homework</title>
  <body>
    <template id="template"><p>Smile!</p></template>
    <script>
      let num = 3;
      const fragment =
document.getElementById('template').content.cloneNode(true);
      while (num-- > 1) {

fragment.firstChild.before(fragment.firstChild.cloneNode(true));
```

```

        fragment.firstChild.textContent +=
fragment.lastChild.textContent;
    }
    document.body.appendChild(fragment);
</script>
</html>

```

The **p** element in the **template** is *not* a child of the **template** in the DOM; it is a child of the **DocumentFragment** returned by the **template** element's **content** IDL attribute.

The content of the **template** do not end up as children of the **template** element. Instead they are inserted into a separate **DocumentFragment**, called the *template contents*. In the *template contents*, elements are inert (scripts do not run, images are not loaded, etc.). Each **template** element has its own *template contents*.

Apart from HTML parser-level syntax requirements, the *template contents* has no conformance requirements. An attribute that is normally required is optional in the *template contents*. Microsyntax requirements for attribute values do not apply in the *template contents*. Content model restrictions (how elements are allowed to be nested) can be violated. And so on. The reason for this is that templates usually need to have placeholders that are replaced with other content upon processing the template. For example:

```

<template>
  <article>
    
    <h1></h1>
  </article>
</template>

```

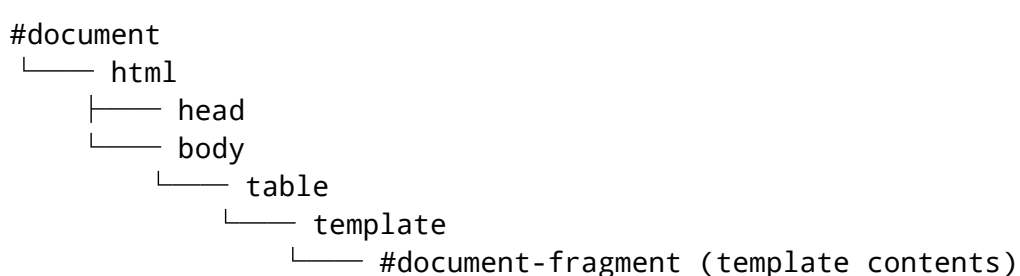
"[[src]]" is not a valid URL, but this is thus OK in the *template contents*.

template elements are allowed essentially anywhere, and allow essentially any contents. For example, a **template** element is not foster parented:

```

<table><template>

```



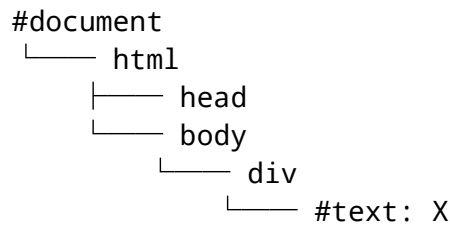
```

#document
├── html
│   ├── head
│   └── body
│       └── table
│           └── template
│               └── #document-fragment (template contents)

```

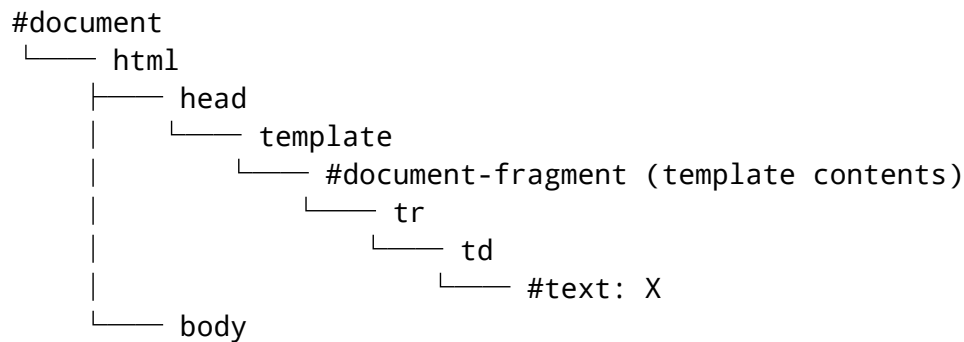
Generally, table markup outside a **table** is ignored:

```
<div><tr><td>X
```



However, in **templates**, it just works:

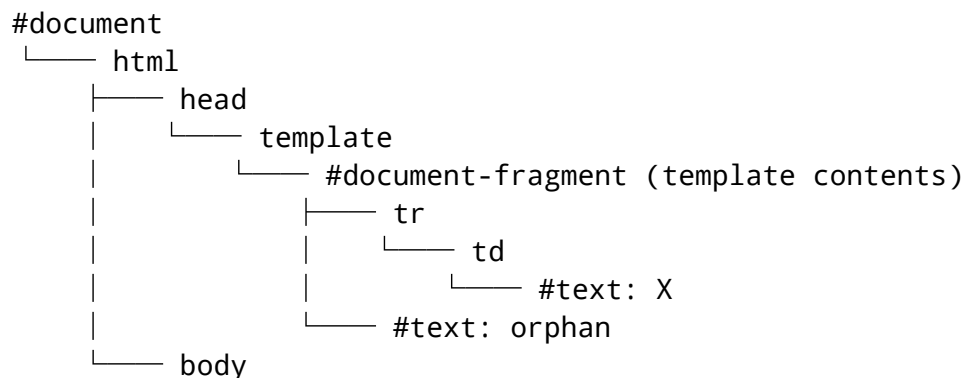
```
<template><tr><td>X
```



If you have unexpected content between the table row and the table cell, it would normally be foster-parented (end up before the table), but here there is no **table** element. Instead it ends up at the end of the **template** element:

```
<template><tr>orphan<td>X
```

...results in the following *template contents*:



The tree builder changes where to insert nodes for template elements in the "[appropriate place for inserting a node](#)" algorithm.

If the adjusted insertion location is inside a **template** element, let it instead be inside the **template** element's template contents, after its last child (if any).

Custom elements

Custom elements ([added to HTML in April 2016](#)) come in two variants:

- autonomous custom elements
- customized built-in elements

Autonomous custom elements have a custom element name, with these requirements:

- it needs to start with a-z (since otherwise it wouldn't parse as a start tag token by the HTML tokenizer)
- It can only use characters that are legal in XML, minus the colon (since otherwise you can't create it with `document.createElement()`, and can't use it in XML)
- It needs to contain at least one dash. The dash is required to prevent clashes with future additions to HTML.

These elements parse just like unknown elements, or like some inline elements like `abbr`, or `dfn`. Which is to say, they don't implicitly close other elements, or have other special parsing behavior.

```
<flag-icon country="nl"></flag-icon>
```

Customized built-in elements are normal HTML elements, with a special `is` attribute. These are parsed just like they usually are, but the parser pays attention to the `is` attribute when [creating the element](#).

```
<button is="plastic-button">Click Me!</button>
```

Some JavaScript is needed to create a definition of custom elements, so that they can do something useful. If you're interested in learning about this, check out [Using custom elements on MDN](#) or the section on [custom elements in the HTML standard](#).

The select element

[#HTMLQuiz](#) how many select elements in the DOM?

```
<select><select><select><select>
```

- 1
- 2

- 4

`select` is a bit special. It generally ignores unexpected tags.

```
<select><div><b><iframe><style><plaintext></select>X
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── select
│       └── #text: X
```

The elements that can be nested in `select` are: `option`, `optgroup`, `script`, `template`.

There are 3 tags that implicitly close a `select` and then be reprocessed: `input`, `keygen`, and `textarea`.

```
<select><input>
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── select
│       └── input
```

The `select` start tag is treated just like the `select end` tag. Therefore, the answer to the quiz is "2".

```
#document
├── html
│   ├── head
│   └── body
│       ├── select
│       └── select
```

`select` inside `tables` are parsed in a separate insertion mode, "in select in table". This is handled the same as "in select", except that table markup closes the `select` element and is then reprocessed.

```
<table><tr><td><select><td>X
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── table
│       │   ├── tbody
│       │   │   ├── tr
│       │   │   │   └── td
│       │   └── 
```

```

|      |_____ select
|_____ td
|      |_____ #text: X

```

Implied tags

[#HTMLQuiz](#) (don't cheat :)) Which elements will be children of body for this?

```
<!doctype html></p><br></br></p>
```

- br
- br, br
- br, br, p
- p, br, br, p

Tags can in various situations be implied by other tags, or by text content. In the [Tables and foster parenting](#) section we discussed table-specific implied tags, e.g., that the `tr` start tag is implied by a `td` or `th` start tag when "in table". The `html`, `head` and `body` start and end tags are optional. (See the [Optional tags](#) section in [Chapter 2. The HTML syntax](#) for the full list of optional tags.)

The `br` end tag is treated as a `br` start tag. This is handled from the "before html" insertion mode through to "in body".

```
</br>
```

```

#document
|_____ html
|      |_____ head
|      |_____ body
|      |_____ br

```

A `p` end tag, when there's no `p` element "in button scope", it implies a `p` start tag before it, so that it ends up as an empty `p` element. However, this only happens in the "in body" insertion mode.

This means that, for the markup in the quiz, the first `p` end tag is ignored. The first `br` start tag steps through the insertion modes to "in body", and inserts a `br` element. Then the `br` end tag inserts another `br` element. Finally, the `p` end tag inserts an empty `p` element. So the correct answer is: `br, br, p`.

Certain end tags are optional, and are implied by some other start tag or by an ancestor's end tag. The HTML standard has an algorithm to *generate implied end tags*:

When the steps below require the UA to generate implied end tags, then, while the current node is a **dd** element, a **dt** element, an **li** element, an **optgroup** element, an **option** element, a **p** element, an **rb** element, an **rp** element, an **rt** element, or an **rtc** element, the UA must pop the current node off the stack of open elements.

For example, one can omit tags in a **ruby** element (this is the Japanese text 漢字, annotated with its reading in hiragana, with parentheses in **rp** elements for browsers that do not support ruby):

```
...<ruby>漢<rp> (<rt>かん<rp>) </rp>字<rp> (<rt>じ<rp>) </ruby>...
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── #text: ...
│       ├── ruby
│       │   ├── #text: 漢
│       │   ├── rp
│       │   │   ├── #text: (
│       │   │   ├── rt
│       │   │   │   ├── #text: かん
│       │   │   │   └── rp
│       │   │   │       ├── #text: )
│       │   │   └── #text: 字
│       │   ├── rp
│       │   │   ├── #text: (
│       │   │   ├── rt
│       │   │   │   ├── #text: じ
│       │   │   │   └── rp
│       │   │       ├── #text: )
│       └── #text: ...
```

This would render as follows:

かん じ
... 漢字 ...

If you have something between the head end tag and the body start tag (where only whitespace is allowed), some tags cause an element to be inserted into the **head** (**base**, **basefont**, **bgsound**, **link**, **meta**, **noframes**, **script**, **style**, **template**, **title**), while other tags or non-whitespace text implicitly opens the **body** element.

```
<!doctype html>
<head>
```

```

</head>
<script></script>
<noscript></noscript>

```

```

#document
├── DOCTYPE: html
├── html
│   ├── head
│   │   ├── #text:
│   │   └── script
│   ├── #text:
│   └── body
│       └── noscript

```

When seeing an `a` start tag if there's an `a` element in the *list of active formatting elements* (see the [Active formatting elements and Noah's Ark](#) section), then it implies an `a` end tag before it, but this is a parse error; the `a` end tag is *not* optional. The following example has two `a` start tags (end tag is mistyped as a start tag):

```

<p><a href="1108470371">Anchor Bar reportedly opening Las Vegas
location<a>.

```

```

#document
├── html
│   ├── head
│   └── body
│       ├── p
│       │   ├── a href="1108470371"
│       │   │   └── #text: Anchor Bar reportedly opening Las Vegas
location
│       └── a
│           └── #text: .

```

Similarly, a `table` start tag in a `table` (not in a table cell) implies a `table` end tag before it (and this is also a parse error). This also happens for `h1-h6` elements; any `h1-h6` start tag token implicitly closes an open `h1-h6` element, even if the tag names don't match.

```

<h1>What is an Open Title?
<h2>Intentionally Left Blank</h2>

```

```

#document
├── html
│   ├── head
│   └── body
│       ├── h1
│       │   └── #text: What is an Open Title?
│       └── h2
│           └── #text: Intentionally Left Blank

```

When in foreign content (SVG or MathML), certain start tags imply closure of open foreign content elements and are then reprocessed. More on this in [The foreign lands](#):

Misnested tags

[#HTMLQuiz](#) HTML allows you to nest `p` in `a`. It also generally allows you to omit `</p>`. Can you do both?

```
<a><p></a>
```

- Yes, that's valid.
- Nope.

What happens if you close elements in the wrong order? It depends on what the markup is.

Some cases are easy, for example, the `h1-h6` elements can be closed by any other `h1-h6` end tag.

```
<h1>Syntax for Headlines</h2>
<h2>WikiMatrix - Compare them all</h1>
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── h1
│       │   └── #text: Syntax for Headlines
│       ├── #text:
│       └── h2
│           └── #text: WikiMatrix - Compare them all
```

The "default" handling of misnested markup, which is used for unknown elements, autonomous custom elements, and some inline elements such as `span`, `dfn`, `kbd`, is to close all open elements until the one given in the end tag has been closed.

```
<span>20 ways to <dfn>commute</span> to</dfn> work.
```

```
#document
├── html
│   ├── head
│   └── body
│       ├── span
│       │   ├── #text: 20 ways to
│       │   └── dfn
│       └──
```

```
|           | #text: commute
|           |
|           | #text:  to work.
```

Other elements are slightly more complicated, such as the **b**, **i**, and **a** elements, which are so-called *formatting elements*.

Active formatting elements and Noah's Ark

The *formatting elements* are:

- a
- b
- big
- code
- em
- font
- i
- nobr
- s
- small
- strike
- strong
- tt
- u

Note that not all elements with a "formatting" default *style* is included in the list, for example the **kbd** element has a monospace by default but is not a formatting element.

A formatting element gets reopened across other elements until it is explicitly closed, like this:

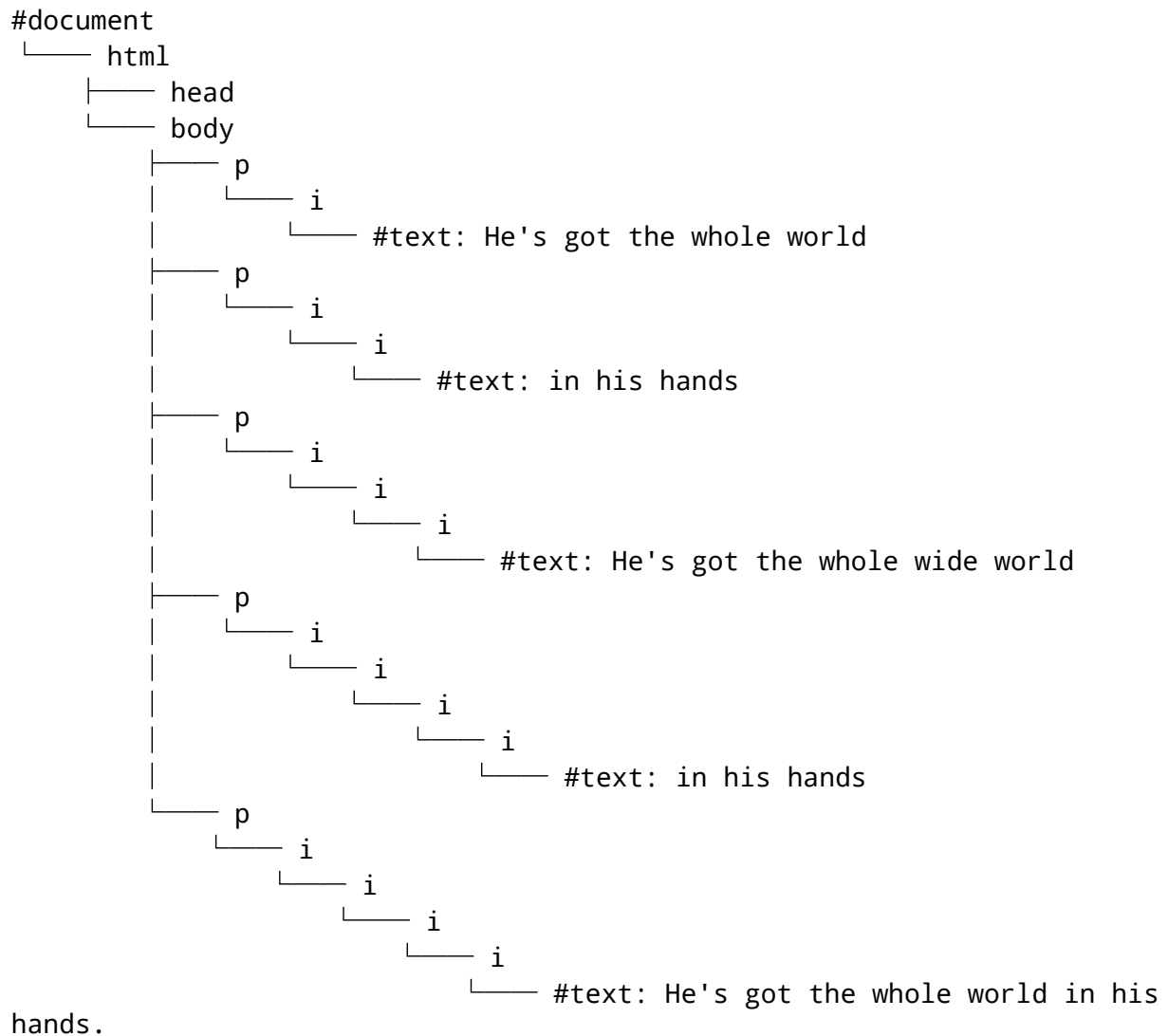
```
<p><i>He's got the whole world
<p>in his hands
```

```
#document
|  html
|    |  head
|    |  body
|    |    |  p
|    |    |    |  i
|    |    |    |    |  #text: He's got the whole world
|    |    |    |
|    |    |    |  p
|    |    |    |    |  i
|    |    |    |    |  #text: in his hands
```

Notice that the second paragraph also has an **i** element.

OK, but what is Noah doing in an HTML parser? Well, in case of a flood, he saves not two but three elements per family.

```
<p><i>He's got the whole world  
<p><i>in his hands  
<p><i>He's got the whole wide world  
<p><i>in his hands  
<p><i>He's got the whole world in his hands.
```

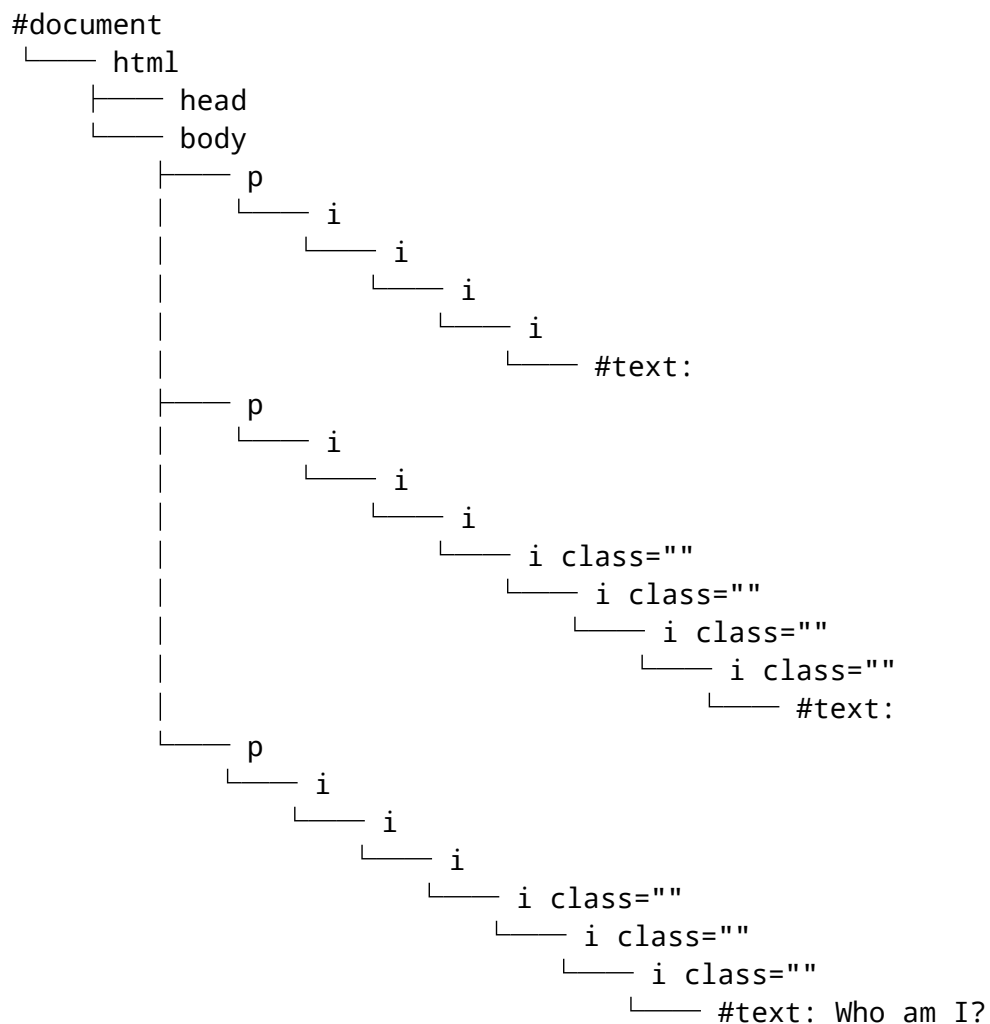


In the third and fourth paragraphs, three `i` elements are reopened, and they open one more themselves, but the number of reopened `i` elements doesn't continue to increase beyond three. This is to avoid insanely huge DOMs for this kind of markup.

The Noah's Ark clause also checks the attributes, not just the tag name.

```
<p><i><i><i><i>  
<p><i class><i class><i class><i class>  
<p>Who am I?
```

The DOM will be:



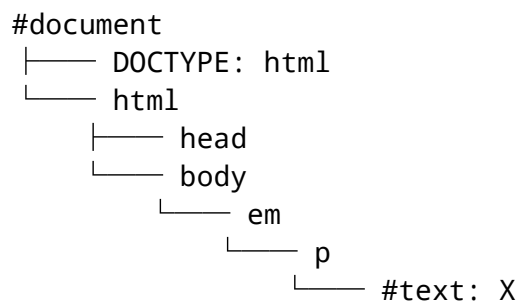
Adoption Agency Algorithm

Do you recall the misnested blocks in inlines case in the [History of HTML parsers](#) section?

```
<!DOCTYPE html><em><p>X</em>Y</p>
```

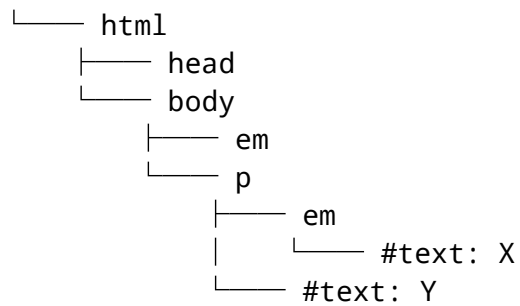
The *Adoption Agency Algorithm* (AAA) governs how to deal with this.

Up to and including the "X", nothing surprising happens. The `p` element is inserted into the `em` element.



When seeing the `em` end tag, the AAA kicks in. It will insert the `p` to the `body`, and insert the `em` to the `p`, and then close the `em` element. The resulting DOM is thus:

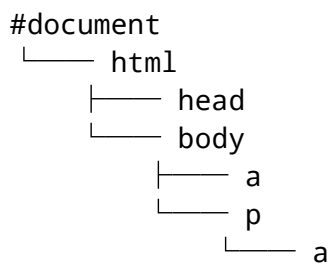




So what happens with the markup in the quiz?

```
<a><p></a>
```

It is exactly the same as the `<em><p></em>` case above, that is, it also triggers AAA, and is thus a parse error and is invalid.



In July 2013, there was [a change to the AAA](#). Before the change, content could sometimes end up in the "wrong" order (not matching source order). This change has been implemented in Firefox, but, as of October 2018 (TODO), has not been implemented in other browsers. Sad panda.

TODO loop limits, marker.

Hoisting attributes

#HTMLQuiz (don't cheat! :)). What attributes will `document.body` have?

```
<body a="1" b="2">Hello!<body b="3" c="4">
```

- {a: 1, b: 2}
- {b: 3, c: 4}
- {a: 1, b: 2, c: 4}
- {a: 1, b: 3, c: 4}

Are you familiar with *hoisting variables* in JavaScript? This is kinda similar, except in HTML, if you use an `html` start tag or a `body` start tag when those elements are already open, it will set the attributes of the token (but not those that already exist on the element) on the `html` or `body` element in the DOM. This doesn't happen for any other element.

The correct answer to the quiz is thus `{a: 1, b: 2, c: 4}`.

Using `html` or `body` tags where they are not expected is, of course, a parse error, so don't do this.

The foreign lands: SVG and MathML

[#HTMLQuiz](#) how many children will the `<svg>` element have in the DOM?

```
<!doctype html><svg><font/><font face/></svg>
```

- 1 child, 1 grandchild
- 2 children
- 1 child, 1 sibling
- no children, 2 siblings

Support for parsing inline SVG and MathML in HTML was [added in April 2008](#). There are both similarities and differences to the XML syntax.

- The `"/>"` empty element syntax is supported.
- CDATA sections are supported.
- Some namespaced attributes such as `xmlns`, `xmlns:xlink`, `xml:lang`, are supported.
- Arbitrary namespaces are *not* supported.
- Namespace prefixes (other than hard-coded `xml` and `xlink`) are not supported.
- Element and attribute names are case-insensitive. SVG elements and attributes (like `viewBox`) with mixed case are fixed up to the correct case in the tree builder.
- Attributes are tokenized just like for HTML elements (e.g., unquoted attributes are allowed).

- HTML (or nested SVG/MathML) can be used in SVG and MathML at certain integration points:
 - SVG `foreignObject`, `desc`, `title`.
 - MathML `mi`, `mo`, `mn`, `ms`, `mtext`, `annotation-xml` (if it has `encoding="text/html"` or `encoding="application/xhtml+xml"`).
- Certain HTML tags in foreign content (not at an integration point) will break out of foreign content back to an integration point or to an HTML element, and create a sibling HTML element for the token.

The last bullet point is specified like this:

A start tag whose tag name is one of: "b", "big", "blockquote", "body", "br", "center", "code", "dd", "div", "dl", "dt", "em", "embed", "h1", "h2", "h3", "h4", "h5", "h6", "head", "hr", "i", "img", "li", "listing", "menu", "meta", "nobr", "ol", "p", "pre", "ruby", "s", "small", "span", "strong", "strike", "sub", "sup", "table", "tt", "u", "ul", "var";
 A start tag whose tag name is "font", if the token has any attributes named "color", "face", or "size"

Parse error.

If the parser was originally created for the HTML fragment parsing algorithm, then act as described in the "any other start tag" entry below.
 (fragment case)

Otherwise:

Pop an element from the stack of open elements, and then keep popping more elements from the stack of open elements until the current node is a MathML text integration point, an HTML integration point, or an element in the HTML namespace.

Then, reprocess the token.

Note that `font` is handled differently depending on its attributes. SVG has a `font` element, but so does HTML. Before SVG and MathML were added to HTML, there were web pages that used "bogus" `<svg>` or `<math>` start tags and then used HTML inside and expected the HTML to be rendered like they did in contemporary browsers. In order to not regress those pages, this breakout list of tags was specified.

The answer to the quiz is therefore: 1 child, 1 sibling. The first `<font/>` is parsed as an SVG element, and the `<font face/>` breaks out of foreign content and creates a sibling HTML `font` element.

The image parser macro

A parser macro in HTML is like a macro in a text editor: a shorthand that expands to something else. The HTML standard has a single parser macro (but used to have another, see [the isindex parser macro](#)): an `image` start tag token is treated as an `img` start tag token by the tree builder.

A start tag whose tag name is "image"

Parse error. Change the token's tag name to "img" and reprocess it. (Don't ask.)

It says "Don't ask", and so evelynn was apparently obliged to ask on X:

I know the MDN literally says "don't ask" but I simply HAVE TO know more about what makes the `<image>` tag so vile

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/image>

[According to Sam Sneddon](#), this parser macro dates back to (at least) Mosaic:

As far as I'm aware, it's not even that. It was "img" from Mosaic's first implementation in 1993. "image" I believe existed as an alias due to developers getting confused why "image" didn't work.

I replied:

Browsers did this before the HTML parser was specified. The spec has this in the source:

```
<!-- As of 2005-12, studies showed that around 0.2% of pages used  
the <image> element. -->
```

HTML+ had `<image>`, but that spec was largely ignored:

https://w3.org/MarkUp/HTMLPlus/htmlplus_21.html

So, browsers had been doing this since forever, and when the HTML parser was specified, enough web content relied on it to cement the behavior to this day.

I wanted to see how prevalent `image` usage is today, so I ran a new [query in HTTP Archive](#). Since there are now pages using inline SVG in text/html, and SVG has an `image`

element, I needed to exclude those in the query. I included only matches that use the `src` attribute, and excluded any matches that use `xlink:href` or `href`, and limited the search to top pages (not including stuff in iframes, scripts or stylesheets):

```
SELECT
  page,
  markup
FROM (
  SELECT
    page,
    ARRAY_TO_STRING(REGEXP_EXTRACT_ALL(body, r'(?i)(<image\s+[^>]*>)'),
    '\n') AS markup
  FROM
    `httparchive.response_bodies.2021_08_01_mobile`
  WHERE
    page = url )
WHERE
  markup != ''
  AND REGEXP_CONTAINS(markup, r'(?i)(\ssrc\s*=)')
  AND NOT REGEXP_CONTAINS(markup, r'(?i)(\s(xlink\:)?href\s*=)')
```

This resulted in [5,840 matched pages](#), out of the total dataset of 7,447,270 pages. This amounts to about 0.08%.

The research method I used here is a bit different from what Ian Hickson did back in 2005, so the numbers aren't directly comparable. We can however conclude that usage is still non-zero. It would be possible to query historical data in HTTP Archive, to figure out if there's a trend (does usage decline over time?). This is left as an exercise to the reader.

Tags that are no longer supported

The vast majority of idiosyncrasies in HTML parsing survive, but not all. This section lists some of the things that were once special parsing behaviors, but have at some point been removed completely.

The `isindex` parser macro

The [very first draft for HTML included an `isindex` tag](#).

This tag informs the reader that the document is an index document. As well as reading it, the reader may use a keyword search.

The effect it had was to present a form with a text field and a search button. When submitted, the typed keywords were added to the URL after a "?", and separated from each other with "+".

The [first draft of HTML 4.0 marked `isindex` as deprecated](#).

When the HTML parser was specified in 2006 (already part of [the initial commit](#)), `isindex` was defined as a parser macro, expanding into:

```
<form><hr><p><label>...text...<input name="isindex" ...attributes...>...text...
</label></p></form>
```

The "...text..." depended on the user's preferred language, per spec:

The two streams of character tokens together should, together with the `input` element, express the equivalent of "This is a searchable index. Insert your search keywords here: (input field)" in the user's preferred language.

The "...attributes..." part was all the attributes from the "isindex" token, except with the "name" attribute set to the value "isindex" (ignoring any explicit "name" attribute).

Internet Explorer and Opera implemented `isindex` as a parser macro already, while Firefox and Safari treated it more like its own element, like a widget. One practical difference is that `document.createElement('isindex')` created an unknown element in Internet Explorer and Opera (and per spec), but worked in Firefox and Safari (before their HTML parser rewrites).

The standard was then tweaked a few times to [support the `action` and `prompt` attributes](#), [remove the `<p>` from the macro](#), [change the default label text](#), until ultimately in 2016, [isindex support was removed altogether](#).

[Chromium removed `isindex` in 2014](#), and [EdgeHTML had also removed it before the spec change](#). [WebKit removed it in 2016](#), and [Gecko in 2017](#).

The motivation for the removal is for security -- the [blink-dev thread](#) points to [this XSS vector](#).

```
<isindex type=image src=1 onerror=alert(1)>
```

Because IE treats the `isindex` element (a very old html element) as a input tag you can specify the same attributes and execute javascript.

Similarly, you could use `<isindex action="javascript:...">` or `<isindex formaction="javascript:...">` or other variants. Today, now that `isindex` parses as an unknown element, the `isindex`-specific XSS variants don't work.

The menuitem element

TODO

Chapter 4. Scripting complications

Revised overview of the HTML parser

TODO

`document.write()`

TODO

Blocking the parser

TODO

Speculative parsing a.k.a. preload scanner

TODO

Other parser APIs

TODO

Window DOMParser Element,ShadowRoot innerHTML Element outerHTML Element
insertAdjacentHTML Range createContextualFragment

innerHTML and friends

TODO some introduction before getting into the weeds...

[#htmlpubquiz](#) How do you get a Siamese twins document (i.e. two `<head>`s and two `<body>`s) using only `innerHTML/outerHTML`?

Correct answer:

```
<!DOCTYPE html>
<script>
document.head.outerHTML = '';
document.body.outerHTML = '';
</script>
```

When the parser reaches `</script>`, before running the script, the **body** element hasn't been created yet:

```
#document
├── DOCTYPE: html
├── html
│   ├── head
│   │   └── script
│   │       └── #text: document.head.outerHTML = '';
└── document.body.outerHTML = '';
```

The first line in the script sets `document.head.outerHTML` to the empty string. `outerHTML` is like `innerHTML` but it replaces the element with the parsed nodes. The spec for [outerHTML](#) will invoke the [fragment parsing algorithm](#) on the given value, and then call the DOM [replace](#) algorithm on the [context object](#) with the parsed result.

The fragment parsing algorithm then calls the [HTML fragment parsing algorithm](#), with *context* being the `html` element (the parent of the `head` element). This will set up a new instance of the HTML parser, with the state of the HTML parser as appropriate for *context*. In particular, this step:

10. Reset the parser's insertion mode appropriately.

...which [says](#):

15. If *node* is an `html` element, run these substeps:

1. If the head element pointer is null, switch the insertion mode to "before head" and return. (fragment case)

So when this parser parses the markup given (the empty string), it starts in the "before head" insertion mode. It immediately reaches EOF, so steps through the usual states and appends a **head** and a **body** element.

At this point, if we were to inspect the DOM right after the `document.head.outerHTML` assignment, it looks like this:

```
#document
├── DOCTYPE: html
└── html
    ├── head
    └── body
```

The parser-created **head** has been replaced by fragment parser-created **head** and **body** elements. Now, `document.body` is no longer null, since a **body** element exists, even though the still running main parser hasn't created one yet.

Next, the `document.body.outerHTML = ''` line does basically the same thing but for the new **body** element: replace it with new **head** and **body** elements:

```
#document
├── DOCTYPE: html
└── html
    ├── head
    ├── head
    └── body
```

The first **head** didn't go away; `outerHTML` only replaces the element you call it on, not other siblings.

Now the script is done, and the main parser is allowed to continue. The insertion mode is "in head", since the **script** element was in **head**. The next token is end-of-file, so the insertion mode switches to "after head", where it inserts a **body** element and switches to "in body", and then it stops parsing.

```
#document
├── DOCTYPE: html
└── html
    ├── head
    ├── head
    ├── body
    └── body
```

DOM manipulation

TODO

Modifying the DOM during parsing

Script can execute during parsing, and those scripts can modify the DOM. This can lead to some interesting effects.

Simple example:

```
<!doctype html>
<body>
<script>
  document.body.remove();
</script>
Oops.
```

The resulting DOM is:

```
#document
├── DOCTYPE: html
└── html
    └── head
```

At least it didn't lose its head...

Note that the text "Oops.", which the parser processed *after* running the script, is not in the DOM. It was inserted into the **body** element, that the script had removed.

[#HTMLQuiz](#) what happens?

```
<iframe id=x></iframe>
<script>
x.contentDocument.body.appendChild(x);
</script>
```

- wild `DOMException` appears
- `iframe` escapes

Correct answer: `iframe` escapes.

Nothing prevents the `iframe` element from being moved to its own document (`about:blank` is same-origin). So the `iframe` element is removed from its original

document.

The spec for [appendChild\(\)](#) does have various checks in place to make sure that the resulting DOM wouldn't violate invariants; e.g., if you tried to append an element to a text node, that would throw. But appending an element to another element is allowed, even across (same-origin) documents.

When an `iframe` is removed from a document, its [browsing context](#) disappears. So the child document does not have a browsing context when the `iframe` element is inserted into it. Therefore the `iframe`, after the move, does not have a new child browsing context (there's no infinite recursion happening). The [spec for the `iframe` element](#) says:



When an `iframe` element is removed from a document, the user agent must discard the element's nested browsing context, if it is not null, and then set the element's nested browsing context to null.

If the script had [saved a reference](#) to the `iframe`'s `window`, the script would still be able to access it, its document, and the moved `iframe` element, after the move.

Chapter 5. Serializing

TODO

Chapter 6. Security implications

Introduction

TODO

Case studies

TODO

Best practice

TODO

Sanitizing HTML

TODO <https://github.com/cure53/DOMPurify>

Appendix A.

Implementations

TODO

Appendix B. Conformance checkers

DTD-based validators

TODO

Validator.nu

TODO

Most common errors

I asked Mike Smith, who contributes to the Validator.nu code base and maintains validator.w3.org and checker.html5.org, about error logs from the validator. He kindly gave me raw logs for one of the instances. I filtered for the messages that come from the Validator.nu HTML parser and normalized the error messages by replacing variable parts with "X", then counted a particular error for a given URL only once. The distribution of these errors are given in the table below.

Percent	Message	Example	Error correction
21.15%	Stray end tag "X".	<code><div></div></code>	<code><div></div></code>
11.97%	Start tag "X" seen but an element of the same type was already open.	<code><a>click <a>here</code>	<code><a>click <a>here</code>
9.57%	No "X" element in scope but a "X" end tag seen.	<code><div></p></div></code>	<code><div><p></p></div></code>
9.11%	End tag "X" seen, but there were open elements.	<code><div></div></code>	<code><div> </div></code>

Percent	Message	Example	Error correction
6.69%	No space between attributes.	<code><div class=""title=""></code>	<code><div class="" title=""></code>
5.77%	Bad start tag in "X" in "head".	<code><head><noscript><div></code>	<code><head><noscript></noscript></head><body><div></code>
5.40%	Duplicate attribute "X".	<code><div role="none" role="navigation"></code>	<code><div role="none"></code>
4.50%	Stray start tag "X".	<code><div><tr></div></code>	<code><div></div></code>
3.32%	End tag for "X" seen, but there were unclosed elements.	<code><i>(EOF)</code>	<code><i></i></body></html>(EOF)</code>
2.58%	Quote "X" in attribute name. Probable cause: Matching quote missing somewhere earlier.	<code><div class=" title=""></code>	<code><div class=" title=""="">></code>
2.46%	Self-closing syntax ("/>") used on a non-void HTML element. Ignoring the slash and treating as a start tag.	<code><div/></code>	<code><div></code>
2.43%	End tag "br".	<code></br></code>	<code>
</code>
2.30%	End tag "X" violates nesting rules.	<code><i><p></i></code>	Adoption agency algorithm
1.40%	Named character reference was not terminated by a semicolon. (Or "&" should have been escaped as "&".)	<code>&nbsp;sp</code>	<code>&nbsp;sp;</code>
1.39%	"X" in an unquoted attribute value. Probable causes: Attributes running together or a URL query string in an unquoted attribute value.	<code></code>	<code></code>

Percent	Message	Example	Error correction
1.25%	Saw "<" when expecting an attribute name. Probable cause: ">" missing immediately before.	<div <p>	<div <p="">
0.98%	A slash was not immediately followed by ">".	<div / >	<div>
0.93%	Stray doctype.	<!doctype html> 	
0.92%	Saw "<!--" within a comment. Probable cause: Nested comment (not allowed).	<!-- <!-- --> -->	The comment is closed on first -->
0.71%	"X" element between "head" and "body".	<head></head><link>	<head><link></head>
0.71%	End tag "X" implied, but there were open elements.	 	
0.70%	Non-space character inside "noscript" inside "head".	<head><noscript>X	<head><noscript> </noscript></head> <body>X
0.62%	Start tag "X" seen in "table".	<table><div>	<table>
0.56%	Saw "X" when expecting an attribute name. Probable cause: Missing "=" immediately before.	<div class="" ">	<div class="" ""="">
0.53%	Bad character "X" after "<". Probable cause: Unescaped "<". Try escaping it as "<".	2<5	2<5
0.41%	Bogus comment.	<!x>	<!--x-->

Percent	Message	Example	Error correction
0.33%	"X" start tag in table body.	<code><table><td></code>	<code><table><tbody><tr><td></code>
0.31%	Heading cannot be a child of another heading.	<code><div> <h2>Introduction<h2> <p>...</p></div></code>	<code><div> <h2>Introduction</h2> <h2><p>...</p></h2> </div></code>
0.23%	End of file seen and there were open elements.	<code><div>(EOF)</code>	<code><div></div>(EOF)</code>
0.23%	Character reference was not terminated by a semicolon.	<code>&#xD</code>	<code>&#xD;</code>
0.23%	End tag had attributes.	<code></div class=main></code>	<code></div></code>
0.15%	A numeric character reference expanded to the C1 controls range.	<code>&#x80;</code>	<code>&#x20AC;</code>
0.15%	Saw a "form" start tag, but there was already an active "form" element. Nested forms are not allowed. Ignoring the tag.	<code><form class="outer"> <form class="inner"></code>	<code><form class="outer"></code>

The table is truncated at 0.1%.

About 73% of the errors are about mismatched tags (e.g., the first 4 errors, 'Bad start tag in "X" in "head".', 'Stray start tag "X".', and so on).

The error "No space between attributes." is pretty harmless. In fact it was conforming in HTML4 to omit space between attributes (when they use single or double quotes).

"Duplicate attribute "X"." is pretty high in the list. Only the first attribute is used, subsequent attributes with the same name are ignored.

"Quote "X" in attribute name. Probable cause: Matching quote missing somewhere earlier." means that the author likely somehow messed up quoting around attribute values.

'Self-closing syntax ("`/>`") used on a non-void HTML element. Ignoring the slash and treating as a start tag.' means that "`/>`" was incorrectly used on a regular HTML element,

which is not supported. It might help to avoid using `</>` syntax altogether in HTML (except for SVG and MathML).

Appendix C. Microsyntaxes

Microsyntaxes in HTML are technically not part of the HTML parser. Instead they are a layer above, (usually) operating on attribute values. For example, boolean attributes have a simple microsyntax, where the allowed value is either the empty string or the same as the attribute name, case-insensitively, and the processing is to ignore the value.

These are thus valid:

```
<input disabled="">  
<input disabled="disabled">  
<input disabled="DISABLED">
```

This is invalid, but is treated the same as the above (the input is disabled):

```
<input disabled="false">
```

Some of the more interesting microsyntaxes are explained in this chapter.

Numbers

HTML has the following kinds of numbers:

- Integers
 - Signed integers
 - Non-negative integers
- Dimensions
 - Percentages and lengths
- Non-zero percentages and lengths
- Floating-point numbers

It further specifies lists of floating-point numbers (used for [image map coordinates](#)), and lists of dimensions (used by the `cols` and `rows` attributes on `frameset`, not covered in this book).

Integers

The format of a signed integer is an optional "-" followed by one or more ASCII digits. The format of non-negative integers is just ASCII digits.

The processing of signed integers is as follows:

- Leading whitespace is skipped.
- A "+" sign before the number is ignored (and is non-conforming).
- A "-" sign before the number makes the number negative.
- The following ASCII digits, if any, are collected.
- Trailing garbage is ignored.
- If there is leading garbage, or if there is no number, then an error is returned. Otherwise the parsed number is returned.

The processing of non-negative integers is the same as that of signed integers, except that negative values result in an error.

Dimensions

The allowed format for dimensions in HTML (for example for the `width` attribute of `img`) is simply that of non-negative integers.

The processing allows for fractions and percentages, but that is non-conforming to use.

- Leading whitespace is skipped.
- A "+" sign before the number is ignored (and is non-conforming).
- A "-" sign before the number makes the number negative.
- The following ASCII digits, and the fraction (if any), are collected.
- If there is a "%" sign after the number, it marks the number as a percentage.
- Trailing garbage is ignored.

- If there is leading garbage, or if there is no number, then an error is returned. Otherwise the return value is the parsed number and the kind of value (percentage or length).

The processing of non-zero dimensions is the same as that of dimensions, except that negative values result in an error.

Floating-point numbers

HTML, JavaScript and CSS all have their own definitions of floating-point numbers. HTML differs from the other two in the format in that a leading "+" sign is not allowed, and if the number is a fraction of one, the leading "0" cannot be omitted. HTML and CSS further cannot represent the Infinity or NaN values.

The following are examples of HTML floating-point numbers.

1

-5.2

1.9e3

1.9E+3

1.9e-3

The numbers with an "e" are using so-called scientific notation, and means the number before the "e" times 10 to the power of the number after the "e". 1.9e3 thus means 1900.

The processing is as follows:

- Leading whitespace is skipped.
- A "+" sign before the number is ignored (and is non-conforming).
- A "-" sign before the number makes the number negative.
- The following ASCII digits, the fraction (if any), the "e" or "E" and the exponent (if any), are collected.
- Trailing garbage is ignored.
- If there is leading garbage, or if there is no number, then an error is returned. Otherwise the parsed number is returned, converted to a finite IEEE 754 double-precision floating-point value. (TODO link)

Image map coordinates

The **area** element represents an area of an image that is a hyperlink. The coordinates of this area is described using the **coords** attributes, which is a list of floating-point numbers, each separated by a "," character (and no other characters, e.g, no whitespace).

```

<map name="cats">
  <area href="hedral.html" shape="rect" coords="50,50,150,200"
alt="Hedral">
  <area href="pillar.html" shape="circle" coords="300,150,100"
alt="Pillar">
</map>
```

The processing is as follows:

- Leading whitespace, commas and semicolons are ignored.
- For each value in the list:
 - Leading garbage (anything but whitespace, comma, semicolon, ASCII digits, "." or "-") is ignored.
 - The number is parsed as a floating-point number. If that returns an error, zero is used instead.
 - Trailing garbage is ignored.
 - Whitespace, comma, or semicolon separates one value from the next.

In January 2016, I changed the specification for parsing lists of floating-point numbers (TODO link ead6cfe392d338b66ed85fa84855061fd0990431). The commit message is as follows:

- Revamp coords parsing to be more compatible and less insane*
- The old parser tried to mimic IE as close as possible. Now Edge is instead interested in aligning with Gecko/WebKit. This new algorithm was designed by studying implementations as well as invalid Web content.*
- At the same time, support parsing of floating point numbers, as suggested by Travis Leithead in the bug below.*
- Fixes https://www.w3.org/Bugs/Public/show_bug.cgi?id=28148.*

Before the change, only integers were allowed, and using a fraction in a number caused that value to be ignored, which was not particularly useful. The handling of bogus values was also especially strange, sometimes dropping all subsequent values.

Responsive images

In May 2012, Ian Hickson added a `srcset` attribute to the `img` element, to address the needs of being able to use images of different resolution depending on the resolution of the screen, and images of different size depending on the viewport size.

Separately, a group of web developers were advocating for an element-based solution instead (`picture`), similar to the markup for the `audio` and `video` elements, citing that the proposed `srcset` syntax was hard to grasp. The Responsive Images Community Group (RICG) was started. [TODO link](#)

In the end (2014), both the `picture` element and the `srcset` attribute were specified, since they could complement each other. A `sizes` attribute was also added, to be used together with "width" descriptors in the `srcset` attribute. For an introduction to responsive images, see the relevant section in the HTML standard [TODO link](#), or [TODO other link](#).

Srcset

The format of the `srcset` attribute is as follows:

- One or more *image candidate strings*, separated by a comma.
- An *image candidate string* has this format:
 - Optional whitespace.
 - A non-empty URL that doesn't start or end with a comma.
 - Optionally a descriptor:
 - A *width descriptor*. Whitespace, a non-negative integer, and a "w" character.
 - A *pixel density descriptor*. Whitespace, a floating-point number, and an "x" character.
 - Optional whitespace.
- If an *image candidate string* has no descriptors and no trailing whitespace, then the next *image candidate string* must begin with whitespace (otherwise it would get jammed together with the previous URL).

A naïve processing would be to split the string on commas and then split on whitespace, to get a list of URLs and their descriptors. However, this would fail to correctly parse URLs

that contain commas (for example data: URLs), and, for the purpose of compatibility with possible future complex descriptors, the parsing of those are more involved, too.

The processing is as follows:

- Leading whitespace and commas are skipped. Leading commas are non-conforming.
- If the end of the string is reached, return the parsed candidates.
- Any non-whitespace is collected for the URL.
- If the URL ends with a comma, then all trailing commas are removed (only a single trailing comma is conforming). Otherwise, descriptors for the current item are parsed:
 - A state machine is used to tokenize descriptors. This is to handle whitespace and commas inside parentheses. For example, `size(50, 50, 30)` is tokenized to a single descriptor. A top-level comma ends the tokenizer.
- The tokenized descriptors are parsed into *density*, *width*, and *future-compat-h*. The last one is for gracefully handling future web content that uses not-yet-specified *height* descriptors in addition to *width* descriptors. If any of the descriptors are invalid, the entire candidate is dropped.
- Run the above steps in a loop.

Sizes

Colors

Meta refresh

Appendix D. Bibliography

[Web Browser Engineering](#)

Web browsers are ubiquitous, but how do they work? This book explains, building a basic but complete web browser, from networking to JavaScript, in a thousand lines of Python. By Pavel Panchekha & Chris Harrelson.

[Web Platform Contribution Guide](#)

Getting started with a history, skill guide and how-to of web standards. By Boaz Sender, Jory Burson, Leo Balter, Mike Pennisi, Simon Pieters, Valerie Young.

HTML 6 | Rubber Duck Engineering

A series of live-streamed live coding an HTML parser in Dart. By Eric Seidel and Adam Barth.

- [HTML 6 | Rubber Duck Engineering | Episode #89](#)
- [HTML 6.2 | Rubber Duck Engineering | Episode #90](#)
- [HTML 2 \$\pi\$ | Rubber Duck Engineering | Episode #91](#)
- [HTML Tree Building | Rubber Duck Engineering | Episode #92](#)