

How modern Web Browsers work¹

Prof. Dr. Stefan Zander

07. March 2018

¹ Special Topic for the Module “Entwicklung Webbasierter Anwendungen”

Objectives:

- Learn about the basic building blocks of modern Web browsers
- Get acquainted with the processing internals and the DOM building logic
- Understand what happens inside the browser when you type in an URL

Contents

1	Preface	1
2	Introduction	2
2.1	What is a Web Browser	2
3	The Browser’s Main Functionality	4
4	The Browser’s High Level Structure	5
4.1	The Rendering Engine	7
4.2	HTML Content Parsing	8
4.3	The Parse Tree	9
4.4	The Parsing Algorithm	9
4.5	Error Tolerance	11
4.6	CSS Parsing	11
5	Processing Order	12
5.1	Scripts	12
5.2	Speculative Parsing	12
5.3	Style Sheets	13
6	Render Tree Construction	13

1 Preface

In the years of IE 90% dominance there was nothing much to do but regard the browser as a “black box”, but now, with open source browsers having more than half of the usage share, it’s a good time to take a peek under the

This lecture note is a revised summary of the excellent article “How Browsers Work: Behind the scenes of modern web browsers” published by Tali Garsiel and Paul Irish in 2011. The original article is available at: <https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>.

There is also a video available at vimeo about Tali’s talk: <http://vimeo.com/44182484>.

engine's hood and see what's inside a web browser. Well, what's inside are millions of C++ lines...

—Tali Garsiel

The following information and facts about the internal operation principles of WebKit and Gecko is the result of extensive research done by the Israeli developer **Tali Garsiel**. Over a few years, she reviewed all the published data about browser internals and spent a lot of time reading Web browser source code. Tali published her research on her site². In the following years, her research results have been revised and republished on numerous occasions and provided insights to a larger audience.

² See <http://taligarsiel.com/>

Why should you learn about browser internals?

Learning the internals of browser operations helps you make better decisions and know the justifications behind development best practices. It also helps you to identify performance bottlenecks and build lightning fast websites. As we will see, page loading time has an influence on the Google page rank—a page loading time > 2 sec. results in a lower rank in the Google search results and the Google crawler also crawls such pages less frequently, meaning that search index terms are less frequently updated and the time until new or updated page content will be considered by the Google search engine is extended.

TODO: Add refs

2 Introduction

*"I just had to take the hypertext idea and connect it to the TCP and DNS ideas and—ta-da!—the World Wide Web."*³

—Sir Tim Berners-Lee, inventor of the World Wide Web

³ One of the famous quotes Sir Tim Berners-Lee used to say about the development of the World Wide Web... [his]

Web browsers are the most widely used software. This lecture explains their fundamental operation principles so that students get an understanding about the things that happen internally when a website is requested, i.e., the time from typing in a website's URL in the browser's address bar until it is rendered by in the browser's viewport.

The complexity of Web browser software has significantly changed over recent years—as the following two screenshots indicate:

TODO: Add image of first NEXT browser; compare it with inspector of Google Chrome; add some description (refs are in the keynote file)

2.1 What is a Web Browser

See <https://seng130.wordpress.com/lectures-2/browser-architecture/>

This section outlines some of the central functionalities provided by a modern Web browser. It helps in understanding what a browser exactly is.

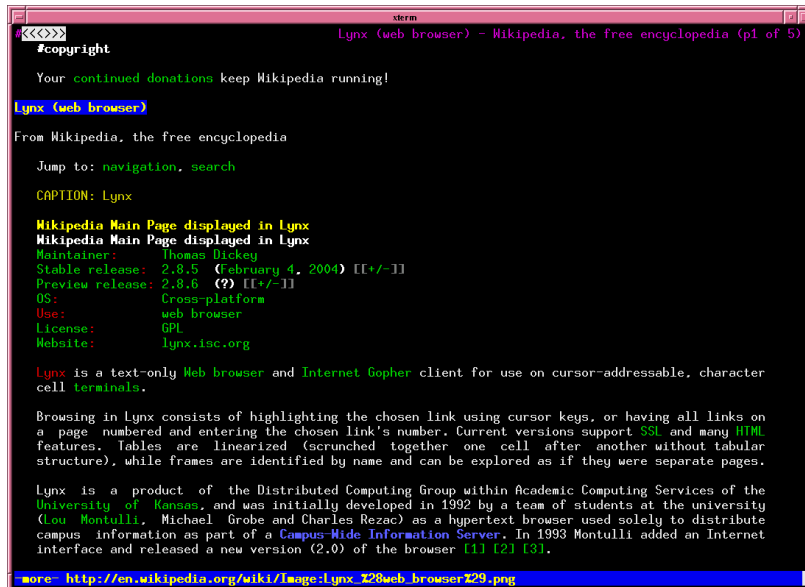


Figure 1: Screenshot of the Lynx Browser, the first and purely text-based browser for the World Wide Web.

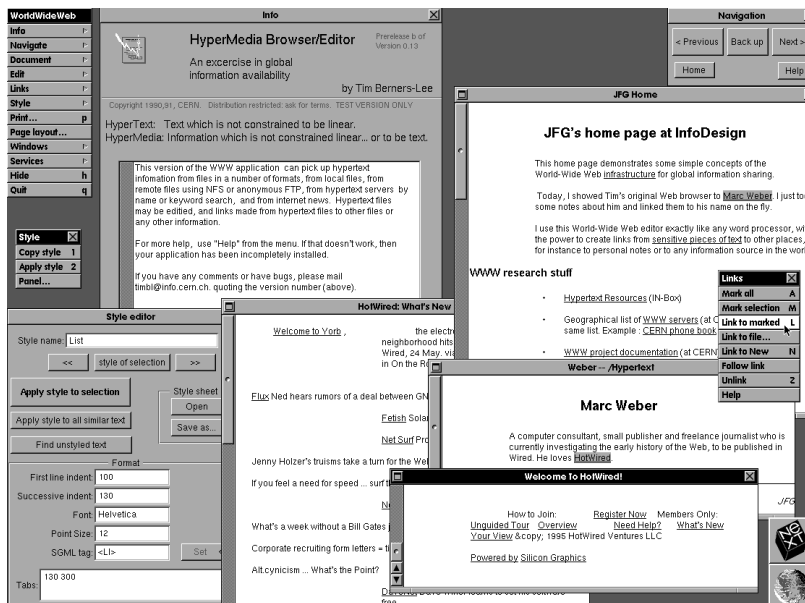


Figure 2: Screenshot of the first WWW browser with a graphical user interface, the NeXT Browser used at CERN.

1. Is an application that we use when we browse the World Wide Web.
2. It renders text-based HTML documents into visual pages, which are what we see inside a browser.
3. It speaks HTTP protocol and communicates with Web servers.
4. It understands URL and knows how to translates URL into Web resources, e.g., HTML text files, images, videos, etc. (*URL dereferencing*)
5. It is a virtual machine that runs the JavaScript programs embedded inside HTML documents.
6. It understands CSS rules and applies the rules to layout the pages.
7. It interacts with a user in front of a browser and translates user inputs into browser events, e.g., clicking a link, clicking a button, submitting text inside a text box.
8. It provides sophisticated tools for analyzing the structure of Web content and network traffic
9. It contains a JavaScript console to utilize its JavaScript engine

3 *The Browser's Main Functionality*

The main function of a browser is to present the web resource you choose, by requesting it from the server and displaying it in the browser window. The resource is usually an HTML document, but may also be a PDF, image, or some other type of content. The location of the resource is specified by the user using a URI (Uniform Resource Identifier).

The way the browser interprets and displays HTML files is specified in the HTML and CSS specifications. These specifications are maintained by the **World Wide Web Consortium** organization⁴, commonly denominated as **W3C**, which is the standards organization for the web. For years browsers conformed to only a part of the specifications and developed their own extensions. That caused serious compatibility issues for web authors. Today most of the browsers more or less conform to the specifications.

⁴ <https://www.w3.org/>

Browser user interfaces have a lot in common with each other. Among the common user interface elements are:

- Address bar for inserting a URI
- Back and forward buttons
- Bookmarking options
- Refresh and stop buttons for refreshing or stopping the loading of current documents

- Home button that takes you to your home page

Strangely enough, the browser's user interface is *not specified* in any formal specification, it just comes from good practices shaped over years of experience and by browsers imitating each other. The HTML5 specification does not define UI elements a browser must have, but lists some common elements. Among those are the address bar, status bar and tool bar. There are, of course, features unique to a specific browser like Firefox's downloads manager.

4 The Browser's High Level Structure

The following components are considered the main building blocks of modern Web browsers (cf. [Grosskurth and Godfrey, 2006]):

A reference architecture for modern Web browsers

1. **The user interface:** this includes the address bar, back/forward button, bookmarking menu, etc. Every part of the browser display except the window where you see the requested page.
2. **The browser engine:** marshals actions between the UI and the rendering engine.
3. **The rendering engine:** responsible for displaying requested content. For example if the requested content is HTML, the rendering engine parses HTML and CSS, and displays the parsed content on the screen.
4. **Networking:** Responsible for network calls such as HTTP requests, using different implementations for different platform behind a platform-independent interface.
5. **UI backend:** used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific. Underneath it uses operating system user interface methods.
6. **JavaScript interpreter:** Used to parse and execute JavaScript code.
7. **Data storage:** This is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem.

Figure 3 provides a graphical overview of the reference elements together with a visualization of their relationships among each other. Figure 4 shows an implementation of the reference architecture elements for the Firefox browser.

It is important to note that browsers such as Chrome run multiple instances of the rendering engine: one for each tab. Each tab runs in a separate process.

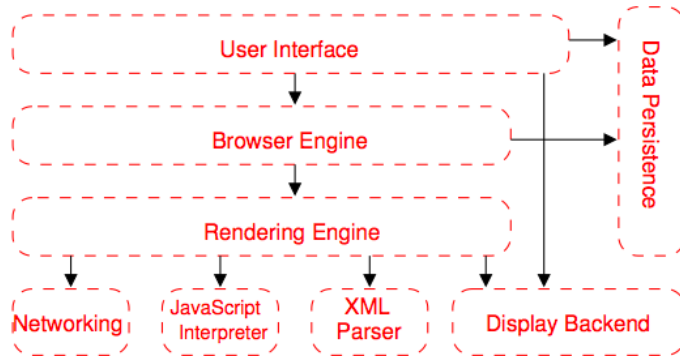


Figure 3: A reference architecture for Web browsers [Grosskurth and Godfrey, 2006]

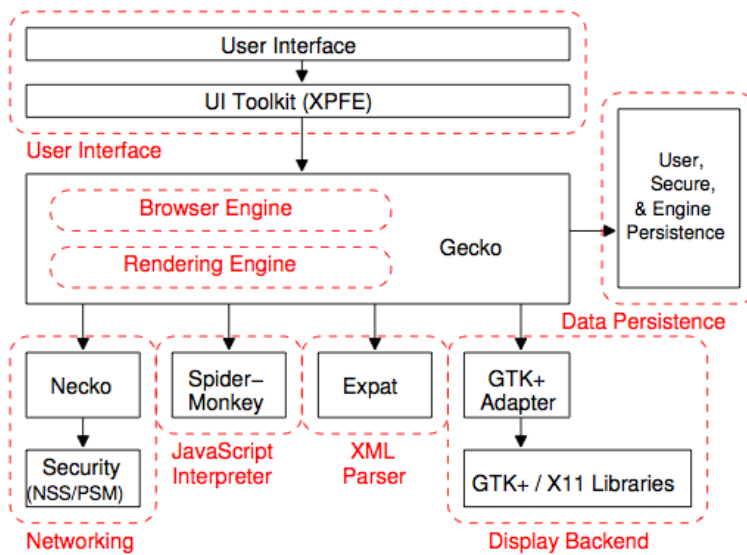


Figure 4: The Firefox browser architecture is based upon the reference architecture components [Grosskurth and Godfrey, 2006]

4.1 The Rendering Engine

Different browsers use different rendering engines: Internet Explorer uses Trident, Firefox uses Gecko, Safari uses WebKit. Chrome and Opera (from version 15) use Blink, a fork of WebKit.

WebKit is an open source rendering engine which started as an engine for the Linux platform and was modified by Apple to support Mac and Windows. See webkit.org for more details.

By default the rendering engine can display HTML and XML documents and images. It can display other types of data via plug-ins or extension; for example, displaying PDF documents using a PDF viewer plug-in. However, in this chapter we will focus on the main use case: displaying HTML and images that are formatted using CSS.

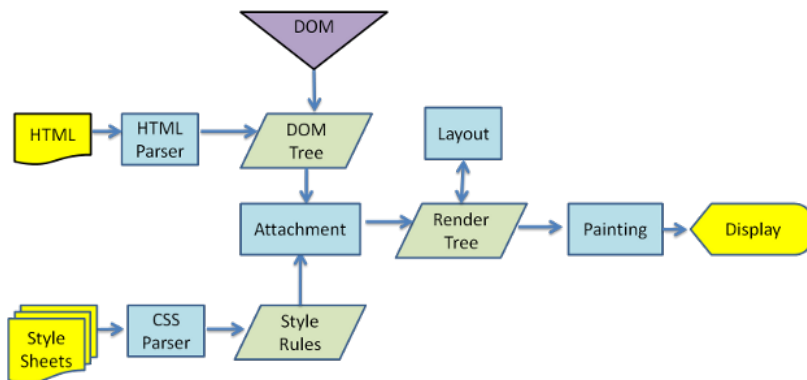


Figure 5: The default workflow of the WebKit render engine

Workflow The rendering engine will start parsing the HTML document and convert elements to DOM nodes in a tree called the **content tree**. The engine will parse the style data, both in external CSS files and in style elements. Styling information together with visual instructions in the HTML will be used to create another tree: the **render tree**.

The render tree contains rectangles with visual attributes like color and dimensions. The rectangles are in the right order to be displayed on the screen.

After the construction of the render tree it goes through a **layout process**. This means giving each node the exact coordinates where it should appear on the screen. The next stage is painting—the render tree will be traversed and each node will be painted using the UI backend layer.

It is important to understand that this is a *gradual process*. For better user experience, the rendering engine will try to display contents on the screen as soon as possible⁵. It will not wait until all HTML is parsed before starting to build and layout the render tree. Parts of the content will be parsed and displayed, while the process continues with the rest of the contents that keeps coming from the network.

⁵ This is referred to as the **Compositing Forest** (cf. <https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome>) since four conceptually different tree structures are processed in parallel:

1. DOM tree
2. RenderObject tree
3. RenderLayer tree
4. GraphicsLayer tree

Parsing Since parsing is a very significant process within the rendering engine, we will go into it a little more deeply. Let's begin with a little introduction about parsing.

Parsing a document means translating it to a structure the code can use. The result of parsing is usually a tree of nodes that represent the structure of the document. This is called a parse tree or a syntax tree.

Parsing can be separated into two sub processes: **lexical analysis** and **syntax analysis**.

Lexical analysis is the process of breaking the input into tokens. Tokens are the language vocabulary: the collection of valid building blocks. In human language it will consist of all the words that appear in the dictionary for that language.

Syntax analysis is the applying of the language syntax rules.

The parsing process is **iterative**. The parser will usually ask the lexer⁶ for a new token and try to match the token with one of the syntax rules. If a rule is matched, a node corresponding to the token will be added to the parse tree and the parser will ask for another token.

If no rule matches, the parser will store the token internally, and keep asking for tokens until a rule matching all the internally stored tokens is found. If no rule is found then the parser will raise an exception. This means the document was not valid and contained syntax errors.

4.2 HTML Content Parsing

The main task of the HTML parser is to parse the HTML markup into a **parse tree**.

There are programs that can create parsers for languages that are defined on the bases of a context free grammar^{7,8} by analyzing a grammar's vocabulary and syntax rules. Such grammars can usually be described using the Backus Naur Form (BNF). However, human language and HTML (as we will see) are not context free grammars. HTML cannot easily be defined by a context free grammar that parsers need. Therefore, a standard parser for parsing HTML does not exist⁹.

TODO: Add consequences

This appears strange at first sight; HTML is rather close to XML. There are lots of available XML parsers. There is an XML variation of HTML—XHTML—so what's the big difference?

The difference is that the HTML approach is more *forgiving*: it lets you omit certain tags (which are then added implicitly), or sometimes omit start or end tags, and so on. On the whole it's a *soft syntax*, as opposed to XML's stiff and demanding syntax.

Parsing is based on the syntax rules the document obeys: the language or format it was written in. Every format you can parse must have deterministic grammar consisting of vocabulary and syntax rules. It is called a context free grammar. Human languages are not such languages and therefore cannot be parsed with conventional parsing techniques.

⁶ The lexer (sometimes called *tokenizer*) that is responsible for breaking the input into valid tokens.

⁷ A language can be parsed by regular parsers if its grammar is a context free grammar.

⁸ An intuitive definition of a context free grammar is a grammar that can be entirely expressed in BNF. For a formal definition see your theoretical informatics' lecture notes.

⁹ There is a formal format for defining HTML-DTD (Document Type Definition), but it is not a context free grammar.

This seemingly small detail makes a world of a difference. On one hand this is the main reason why HTML is so popular: it forgives your mistakes and makes life easy for the Web author. On the other hand, it makes it difficult to write a formal grammar. So to summarize, HTML cannot be parsed easily by conventional parsers, since its grammar is not context free. HTML cannot be parsed by XML parsers.

This has some consequence for Web development:

4.3 The Parse Tree

The output tree (i.e., the **parse tree**) is a tree of DOM element and attribute nodes. DOM is short for **Document Object Model**. It is the object presentation of the HTML document and the interface of HTML elements to the outside world like JavaScript. The root of the tree is the Document object.

The DOM has an almost one-to-one relation to the markup.

For example, the HTML code in Listing 4.3 will be translated by a Web browser in the internal DOM tree structure as illustrated in Figure 6.

```

1 <html>
2   <body>
3     <p>
4       Hello World
5     </p>
6     <div> </div>
7   </body>
8 </html>

```

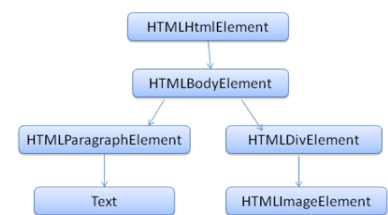


Figure 6: The corresponding internal DOM tree created out of the HTML code example

Like HTML, DOM is specified by the W3C organization¹⁰. It is a generic specification for manipulating documents. A specific module describes HTML specific elements¹¹.

By saying the tree contains DOM nodes, the more precise meaning is that the tree is constructed of elements that implement one of the DOM interfaces. Browsers use concrete implementations that have other attributes used by the browser internally.

4.4 The Parsing Algorithm

HTML cannot be parsed using the regular top down or bottom up parsers.

The reasons are:

- The **forgiving nature** of the language.

¹⁰ Seen www.w3.org/DOM/DOMTR

¹¹ The HTML definitions can be found here: www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.html.

- The fact that browsers have traditional **error tolerance** to support well known cases of invalid HTML.
- The parsing process is **reentrant**. For other languages, the source doesn't change during parsing, but in HTML, dynamic code (such as script elements containing `document.write()` calls) can add extra tokens, so the parsing process actually modifies the input.

Unable to use the regular parsing techniques, browsers create **custom parsers** for parsing HTML.

The **parsing algorithm** is described in detail by the HTML5 specification¹². The algorithm consists of two stages:

¹² See <http://www.whatwg.org/specs/web-apps/current-work/multipage/parsing.html>

1. Tokenization

Tokenization is the *lexical analysis*, i.e., parsing the input into tokens. Among HTML tokens are start tags, end tags, attribute names and attribute values.

The Tokenization Algorithm

The tokenizer recognizes the token, gives it to the tree constructor, and consumes the next character for recognizing the next token, and so on until the end of the input.

The algorithm's output is an HTML token. The algorithm is expressed as a *state machine*. Each state consumes one or more characters of the input stream and updates the next state according to those characters. The decision is influenced by the current tokenization state and by the tree construction state. This means the same consumed character will yield different results for the correct next state, depending on the current state. The algorithm is very complex to describe fully; an example to understand the principle can be found in xxx.

2. Tree Construction

During the tree construction stage the DOM tree with the Document in its root will be modified and elements will be added to it. Each node emitted by the tokenizer will be processed by the tree constructor. For each token the specification defines which DOM element is relevant to it and will be created for this token. The element is added to the DOM tree, and also the stack of open elements. This stack is used to correct nesting mismatches and unclosed tags. The algorithm is also described as a state machine. The states are called *insertion modes*.

Tree Construction Algorithm

When the HTML document parsing is finished, the browser will mark the document as interactive and start parsing **scripts** that are in **deferred mode**: those that should be executed *after* the document is parsed. The document state will be then set to complete and a load event will be fired.

4.5 Error Tolerance

You never get an Invalid Syntax error on an HTML page. Browsers fix any invalid content and go on. Take this HTML for example:

```

1 <html>
2   <mytag>
3   </mytag>
4   <div>
5   <p>
6   </div>
7     Really lousy HTML
8   </p>
9 </html>

```

Listing 1: A HTML document containing an intentionally large number of errors; however, it will still be processed as if it were error-free by modern Web browsers.

The code in Listing 1 figuratively violates about a million rules ("mytag" is not a standard tag, wrong nesting of the p and div elements and more) but the browser still shows it correctly and doesn't complain. So a lot of the parser code is fixing the HTML author mistakes.

Error handling is quite consistent in browsers, but amazingly enough it hasn't been part of HTML specifications. Like bookmarking and back/forward buttons, it's just something that developed in browsers over the years. There are known invalid HTML constructs repeated on many sites, and the browsers try to fix them in a way conformant with other browsers.

4.6 CSS Parsing

Unlike HTML, CSS is a context free grammar and can be parsed using standard parsers as described in the HTML Parsing section. The CSS lexical and syntax grammar as defined by the CSS specification is available at <http://www.w3.org/TR/CSS2/grammar.html>.

WebKit uses Flex¹³ and Bison¹⁴ parser generators to create parsers automatically from the CSS grammar files. As you recall from the parser introduction, Bison creates a bottom up shift-reduce parser. Firefox uses a top down parser written manually. In both cases each CSS file is parsed into a StyleSheet object. Each object contains CSS rules. The CSS rule objects contain selector and declaration objects and other objects corresponding to CSS grammar.

¹³ See http://en.wikipedia.org/wiki/Flex_lexical_analyser

¹⁴ See <http://www.gnu.org/software/bison/>

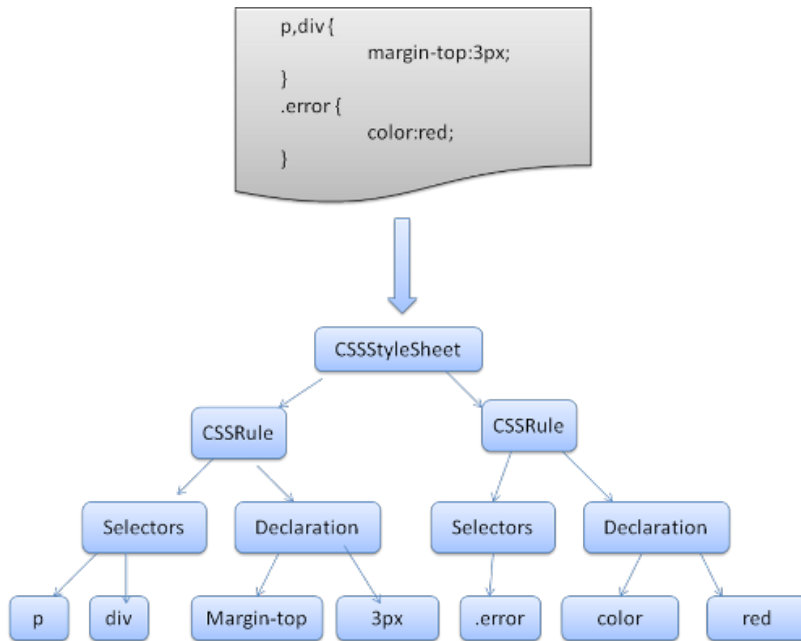


Figure 7: Example of a CSS StyleSheet Object tree created by a CSS Parser

5 Processing Order

5.1 Scripts

The model of the web is synchronous. Authors expect scripts to be parsed and executed immediately when the parser reaches a `<script>` tag. The parsing of the document halts until the script has been executed. If the script is external then the resource must first be fetched from the network—this is also done synchronously, and parsing halts until the resource is fetched. This was the model for many years and is also specified in HTML4 and 5 specifications. Authors can add the `defer` attribute to a script, in which case it will not halt document parsing and will execute after the document is parsed. HTML5 adds an option to mark the script as asynchronous so it will be parsed and executed by a different thread.

5.2 Speculative Parsing

Both WebKit and Firefox implement the speculative parsing optimization. While executing scripts, another thread parses the rest of the document and finds out what other resources need to be loaded from the network and loads them. In this way, resources can be loaded on parallel connections and overall speed is improved. Note: the speculative parser only parses references to external resources like external scripts, style sheets and images—it doesn't modify the DOM tree, that is left to the main parser.

WebKit and Gecko employ speculative parsing

Speculative parsers do not modify the DOM tree

5.3 *Style Sheets*

Style sheets on the other hand have a different model. Conceptually it seems that since style sheets don't change the DOM tree, there is no reason to wait for them and stop the document parsing. There is an issue, though, of scripts asking for style information during the document parsing stage. If the style is not loaded and parsed yet, the script will get wrong answers and apparently this caused lots of problems. It seems to be an edge case but is quite common. Firefox blocks all scripts when there is a style sheet that is still being loaded and parsed. WebKit blocks scripts only when they try to access certain style properties that may be affected by unloaded style sheets.

Scripts might request style information from styles the parsing of which has not finished yet.

Script execution is blocked when style information is requested from unfinished CSS parsing tasks.

6 *Render Tree Construction*

While the DOM tree is being constructed, the browser constructs another tree, the render tree. This tree is of visual elements in the order in which they will be displayed. It is the visual representation of the document. The purpose of this tree is to enable painting the contents in their correct order. Firefox calls the elements in the render tree *frames*. WebKit uses the term *renderer* or *render object*. A renderer knows how to lay out and paint itself and its children.

References

The world wide web of tim berners-lee. URL <http://history-computer.com/Internet/Maturing/Lee.html>.

Alan Grosskurth and Michael W. Godfrey. A Reference Architecture For Web Browsers, 2006. URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.1151>.