How Browsers Work: Behind the scenes of modern web browsers


By Tali Garsiel and Paul Irish

Published: August 5th, 2011
Comments: 10

Preface

This comprehensive primer on the internal operations of WebKit and Gecko is the result of much research done by Israeli developer Tali Garsiel. Over a few years, she reviewed all the published data about browser internals (see Resources) and spent a lot of time reading web browser source code. She wrote:

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

Tali published her research on her site, but we knew it deserved a larger audience, so we've cleaned it up and republished it here.

As a web developer, **learning the internals of browser operations helps you make better decisions and know the justifications behind development best practices**. While this is a rather lengthy document, we recommend you spend some time digging in; we guarantee you'll be glad you did. *Paul Irish, Chrome Developer Relations*

> This article has been translated into a few languages: HTML5 Rocks hosts the German, Spanish, Japanese, Portuguese, Russian and Simplified Chinese versions. You can view the externally hosted translations of Korean and Turkish as well.
>
> You can also watch Tali Garsiel give a talk on this topic on Vimeo:
>
> 
>

Introduction

Web browsers are the most widely used software. In this primer, I will explain how they work behind the scenes. We will see what happens when you type `google.com` in the address bar until you see the Google page on the browser screen.

Table of Contents

        2. Speculative parsing
        3. Style sheets

4. Render tree construction

    1. The render tree relation to the DOM tree
    2. The flow of constructing the tree
    3. Style Computation

        1. Sharing style data
        2. Firefox rule tree

            1. Division into structs
            2. Computing the style contexts using the rule tree

        3. Manipulating the rules for an easy match
        4. Applying the rules in the correct cascade order

            1. Style sheet cascade order
            2. Specificity
            3. Sorting the rules

    4. Gradual process

5. Layout

    1. Dirty bit system
    2. Global and incremental layout
    3. Asynchronous and synchronous layout
    4. Optimizations
    5. The layout process
    6. Width calculation
    7. Line breaking

6. Painting

    1. Global and incremental
    2. The painting order

The browsers we will talk about

There are five major browsers used on desktop today: Chrome, Internet Explorer, Firefox, Safari and Opera. On mobile, the main browsers are Android Browser, iPhone, Opera Mini and Opera Mobile, UC Browser, the Nokia S40/S60 browsers and Chrome–all of which, except for the Opera browsers, are based on WebKit. I will give examples from the open source browsers Firefox and Chrome, and Safari (which is partly open source). According to StatCounter statistics (as of June 2013) Chrome, Firefox and Safari make up around 71% of global desktop browser usage. On mobile, Android Browser, iPhone and Chrome constitute around 54% of usage.

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 W3C (World Wide Web Consortium) organization, 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.

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 doesn't 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.

The browser's high level structure

The browser's main components are (1.1):

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**: 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.

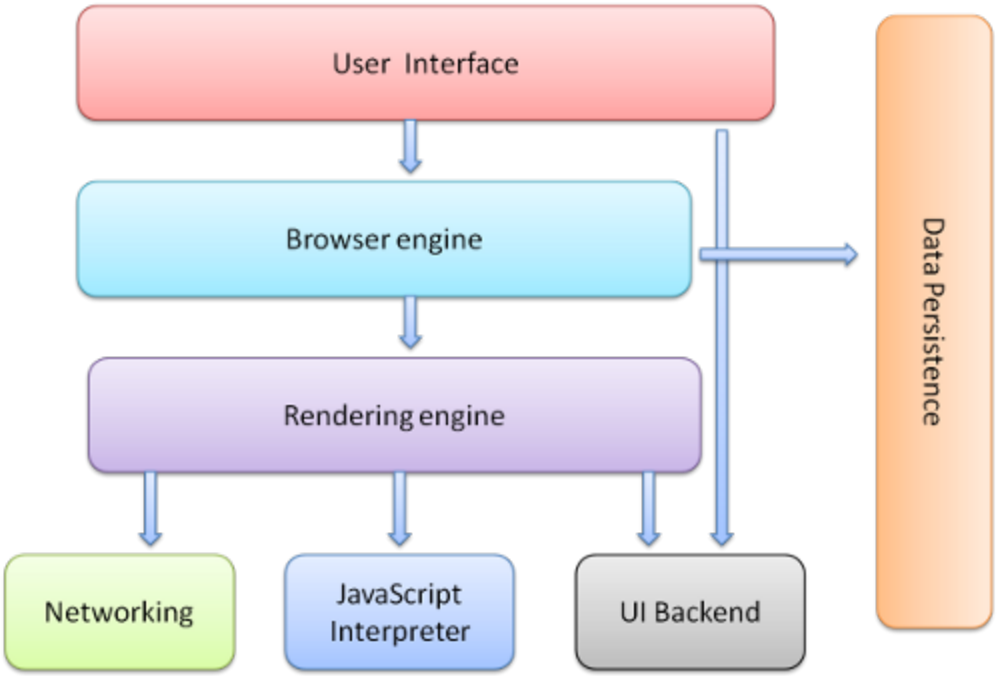


Figure : Browser components

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.

The rendering engine

The responsibility of the rendering engine is well... Rendering, that is display of the requested contents on the browser screen.

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.

Rendering engines

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.

The main flow

The rendering engine will start getting the contents of the requested document from the networking layer. This will usually be done in 8kB chunks.

After that, this is the basic flow of the rendering engine:

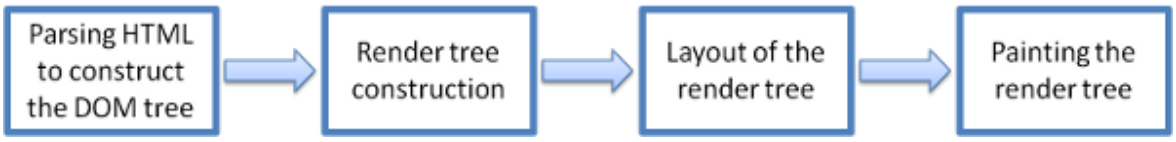


Figure : Rendering engine basic flow

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's 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 possible. 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.

Main flow examples

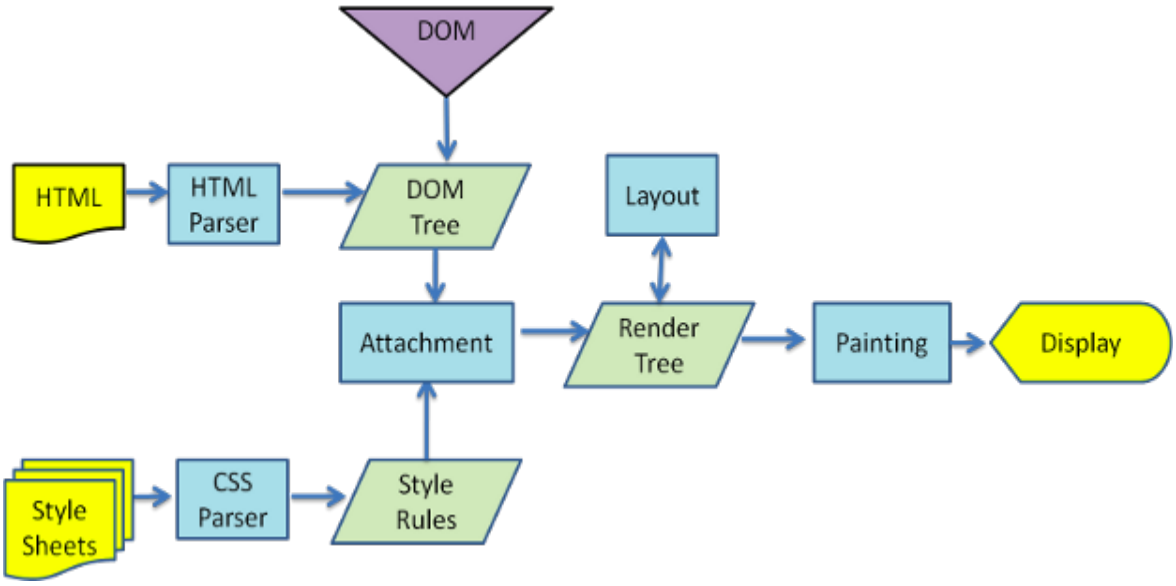


Figure : WebKit main flow

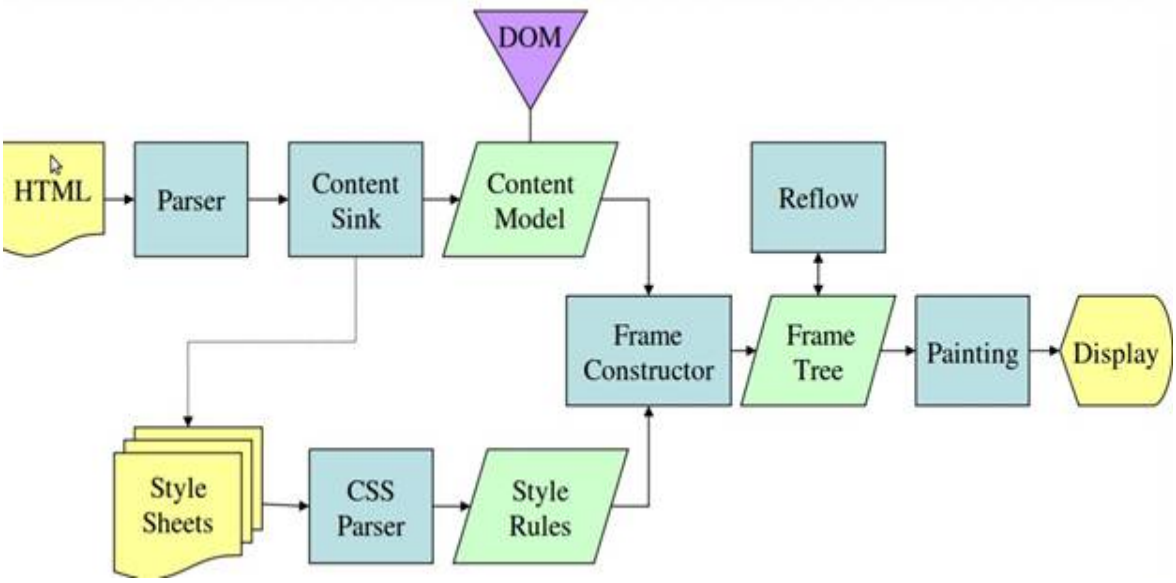


Figure : Mozilla's Gecko rendering engine main flow (3.6)

From figures 3 and 4 you can see that although WebKit and Gecko use

slightly different terminology, the flow is basically the same.

Gecko calls the tree of visually formatted elements a "Frame tree". Each element is a frame. WebKit uses the term "Render Tree" and it consists of "Render Objects". WebKit uses the term "layout" for the placing of elements, while Gecko calls it "Reflow". "Attachment" is WebKit's term for connecting DOM nodes and visual information to create the render tree. A minor non-semantic difference is that Gecko has an extra layer between the HTML and the DOM tree. It is called the "content sink" and is a factory for making DOM elements. We will talk about each part of the flow:

Parsing–general

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.

For example, parsing the expression `2 + 3 - 1` could return this tree:

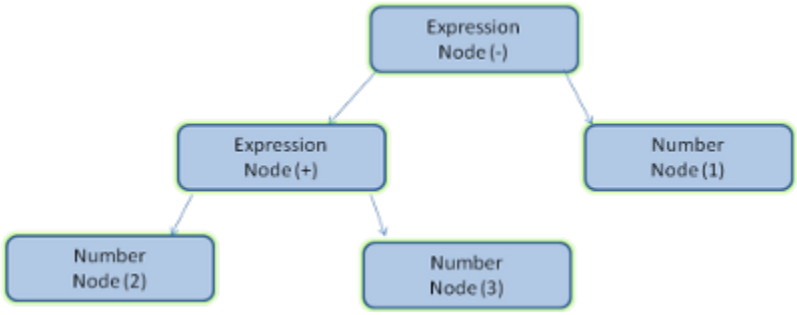


Figure : mathematical expression tree node

Grammars

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.

Parser–Lexer combination

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.

Parsers usually divide the work between two components: the **lexer** (sometimes called tokenizer) that is responsible for breaking the input into valid tokens, and the **parser** that is responsible for constructing the parse tree by analyzing the document structure according to the language syntax rules. The lexer knows how to strip irrelevant characters like white spaces and line breaks.

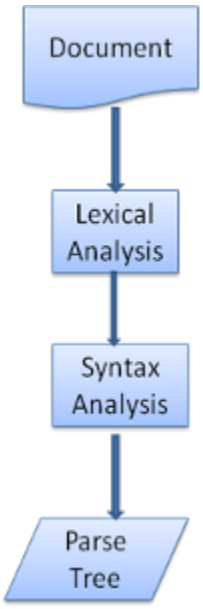


Figure : from source document to parse trees

The parsing process is iterative. The parser will usually ask the lexer 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.

Translation

In many cases the parse tree is not the final product. Parsing is often used in translation: transforming the input document to another format. An example is compilation. The compiler that compiles source code into machine code first parses it into a parse tree and then translates the tree into a machine code document.

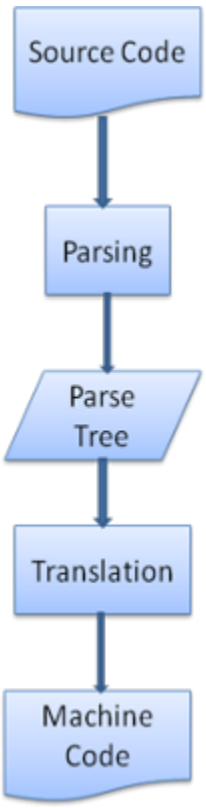


Figure : compilation flow

Parsing example

In figure 5 we built a parse tree from a mathematical expression. Let's try to define a simple mathematical language and see the parse process.

Vocabulary: Our language can include integers, plus signs and minus signs.

Syntax:

1. The language syntax building blocks are expressions, terms and operations.
2. Our language can include any number of expressions.
3. An expression is defined as a "term" followed by an "operation" followed by another term
4. An operation is a plus token or a minus token
5. A term is an integer token or an expression

Let's analyze the input `2 + 3 - 1`.
The first substring that matches a rule is `2`: according to rule #5 it is a term. The second match is `2 + 3`: this matches the third rule: a term followed by an operation followed by another term. The next match will only be hit at the end of the input. `2 + 3 - 1` is an expression because we already know that `2 + 3` is a term, so we have a term followed by an operation followed by another term. `2 + +` will not match any rule and therefore is an invalid input.

Formal definitions for vocabulary and syntax

Vocabulary is usually expressed by regular expressions.

For example our language will be defined as:

```
INTEGER: 0|[1-9][0-9]*
PLUS: +
MINUS: -
```

As you see, integers are defined by a regular expression.

Syntax is usually defined in a format called BNF. Our language will be defined as:

```
expression :=  term  operation  term
operation :=  PLUS | MINUS
term := INTEGER | expression
```

We said that a language can be parsed by regular parsers if its grammar is a context free grammar. An intuitive definition of a context free grammar is a grammar that can be entirely expressed in BNF. For a formal definition see Wikipedia's article on Context-free grammar

Types of parsers

There are two types of parsers: top down parsers and bottom up parsers. An intuitive explanation is that top down parsers examine the high level structure of the syntax and try to find a rule match. Bottom up parsers start with the input and gradually transform it into the syntax rules, starting from the low level rules until high level rules are met.

Let's see how the two types of parsers will parse our example.

The top down parser will start from the higher level rule: it will identify `2 + 3` as an expression. It will then identify `2 + 3 - 1` as an expression (the process of identifying the expression evolves, matching the other rules, but the start point is the highest level rule).

The bottom up parser will scan the input until a rule is matched. It will then replace the matching input with the rule. This will go on until the end of the input. The partly matched expression is placed on the parser's stack.

Stack	Input
	`2 + 3 - 1`
term	`+ 3 - 1`
term operation	`3 - 1`

expression	- 1
expression operation	1
expression	-

This type of bottom up parser is called a shift-reduce parser, because the input is shifted to the right (imagine a pointer pointing first at the input start and moving to the right) and is gradually reduced to syntax rules.

Generating parsers automatically

There are tools that can generate a parser. You feed them the grammar of your language–its vocabulary and syntax rules–and they generate a working parser. Creating a parser requires a deep understanding of parsing and it's not easy to create an optimized parser by hand, so parser generators can be very useful.

WebKit uses two well known parser generators: Flex for creating a lexer and Bison for creating a parser (you might run into them with the names Lex and Yacc). Flex input is a file containing regular expression definitions of the tokens. Bison's input is the language syntax rules in BNF format.

HTML Parser

The job of the HTML parser is to parse the HTML markup into a parse tree.

The HTML grammar definition

The vocabulary and syntax of HTML are defined in specifications created by the W3C organization.

Not a context free grammar

As we have seen in the parsing introduction, grammar syntax can be defined formally using formats like BNF.

Unfortunately all the conventional parser topics do not apply to HTML (I didn't

bring them up just for fun–they will be used in parsing CSS and JavaScript). HTML cannot easily be defined by a context free grammar that parsers need.

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

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.

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.

HTML DTD

HTML definition is in a DTD format. This format is used to define languages of the SGML family. The format contains definitions for all allowed elements, their attributes and hierarchy. As we saw earlier, the HTML DTD doesn't form a context free grammar.

There are a few variations of the DTD. The strict mode conforms solely to the specifications but other modes contain support for markup used by browsers in the past. The purpose is backwards compatibility with older content. The current strict DTD is here: www.w3.org/TR/html4/strict.dtd

DOM

The output tree (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:

```
<html>
  <body>
    <p>
      Hello World
    </p>
    <div> <img src="example.png"/></div>
  </body>
</html>
```

This markup would be translated to the following DOM tree:

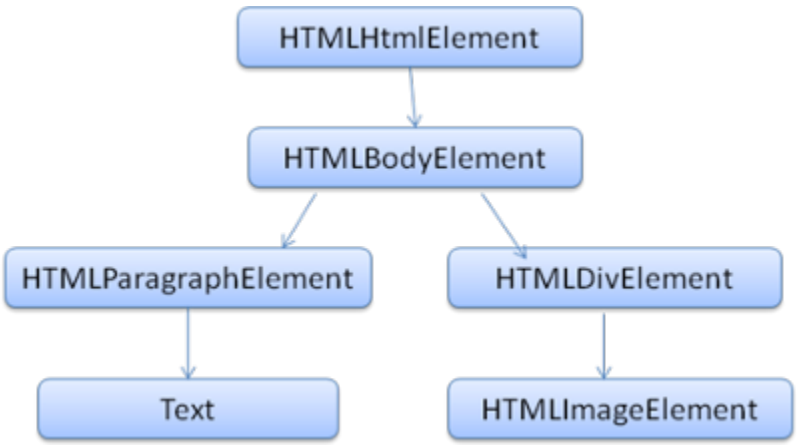


Figure : DOM tree of the example markup

Like HTML, DOM is specified by the W3C organization. See www.w3.org/DOM/DOMTR. It is a generic specification for manipulating documents. A specific module describes HTML specific elements. The HTML definitions can be found here: www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.html.

When I say the tree contains DOM nodes, I mean 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.

The parsing algorithm

As we saw in the previous sections, HTML cannot be parsed using the regular top down or bottom up parsers.

The reasons are:

1. The forgiving nature of the language.
2. The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.
3. 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 specification. The algorithm consists of two stages: tokenization and tree construction.

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

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.

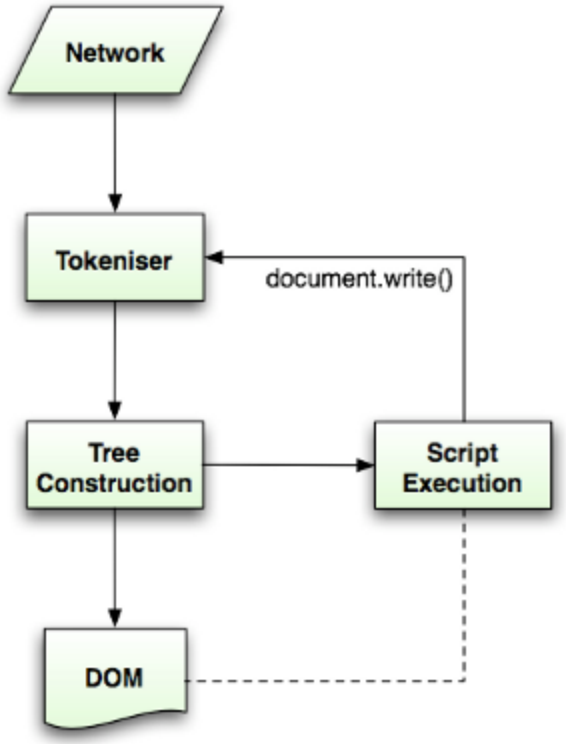


Figure : HTML parsing flow (taken from HTML5 spec)

The tokenization algorithm

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 too complex to describe fully, so let's see a simple example that will help us understand the principle.

Basic example–tokenizing the following HTML:

```
<html>
  <body>
    Hello world
  </body>
```

```
</html>
```

The initial state is the "Data state". When the `<` character is encountered, the state is changed to **"Tag open state"**. Consuming an `a-z` character causes creation of a "Start tag token", the state is changed to **"Tag name state"**. We stay in this state until the `>` character is consumed. Each character is appended to the new token name. In our case the created token is an `html` token.

When the `>` tag is reached, the current token is emitted and the state changes back to the **"Data state"**. The `<body>` tag will be treated by the same steps. So far the `html` and `body` tags were emitted. We are now back at the **"Data state"**. Consuming the `H` character of `Hello world` will cause creation and emitting of a character token, this goes on until the `<` of `</body>` is reached. We will emit a character token for each character of `Hello world`.

We are now back at the **"Tag open state"**. Consuming the next input `/` will cause creation of an `end tag token` and a move to the **"Tag name state"**. Again we stay in this state until we reach `>`.Then the new tag token will be emitted and we go back to the **"Data state"**. The `</html>` input will be treated like the previous case.

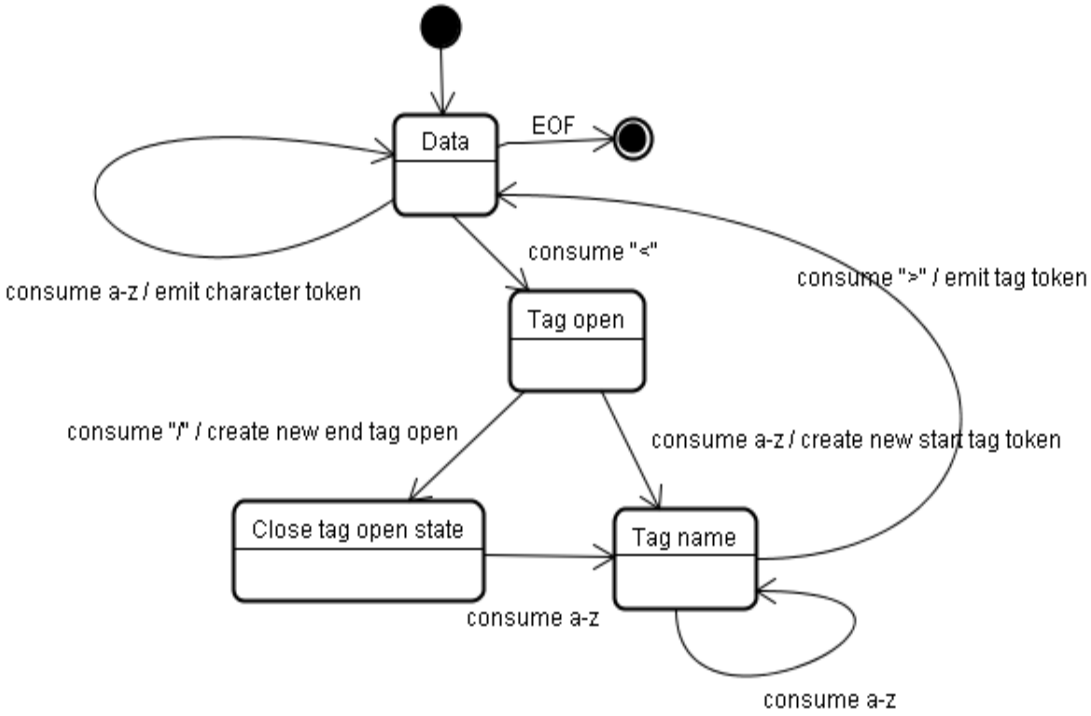


Figure : Tokenizing the example input

Tree construction algorithm

When the parser is created the Document object is created. 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".

Let's see the tree construction process for the example input:

```
<html>
  <body>
    Hello world
```

```
  </body>
</html>
```

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The first mode is the **"initial mode"**. Receiving the "html" token will cause a move to the **"before html"** mode and a reprocessing of the token in that mode. This will cause creation of the HTMLHtmlElement element, which will be appended to the root Document object.

The state will be changed to **"before head"**. The "body" token is then received. An HTMLHeadElement will be created implicitly although we don't have a "head" token and it will be added to the tree.

We now move to the **"in head"** mode and then to **"after head"**. The body token is reprocessed, an HTMLBodyElement is created and inserted and the mode is transferred to **"in body"**.

The character tokens of the "Hello world" string are now received. The first one will cause creation and insertion of a "Text" node and the other characters will be appended to that node.

The receiving of the body end token will cause a transfer to **"after body"** mode. We will now receive the html end tag which will move us to **"after after body"** mode. Receiving the end of file token will end the parsing.

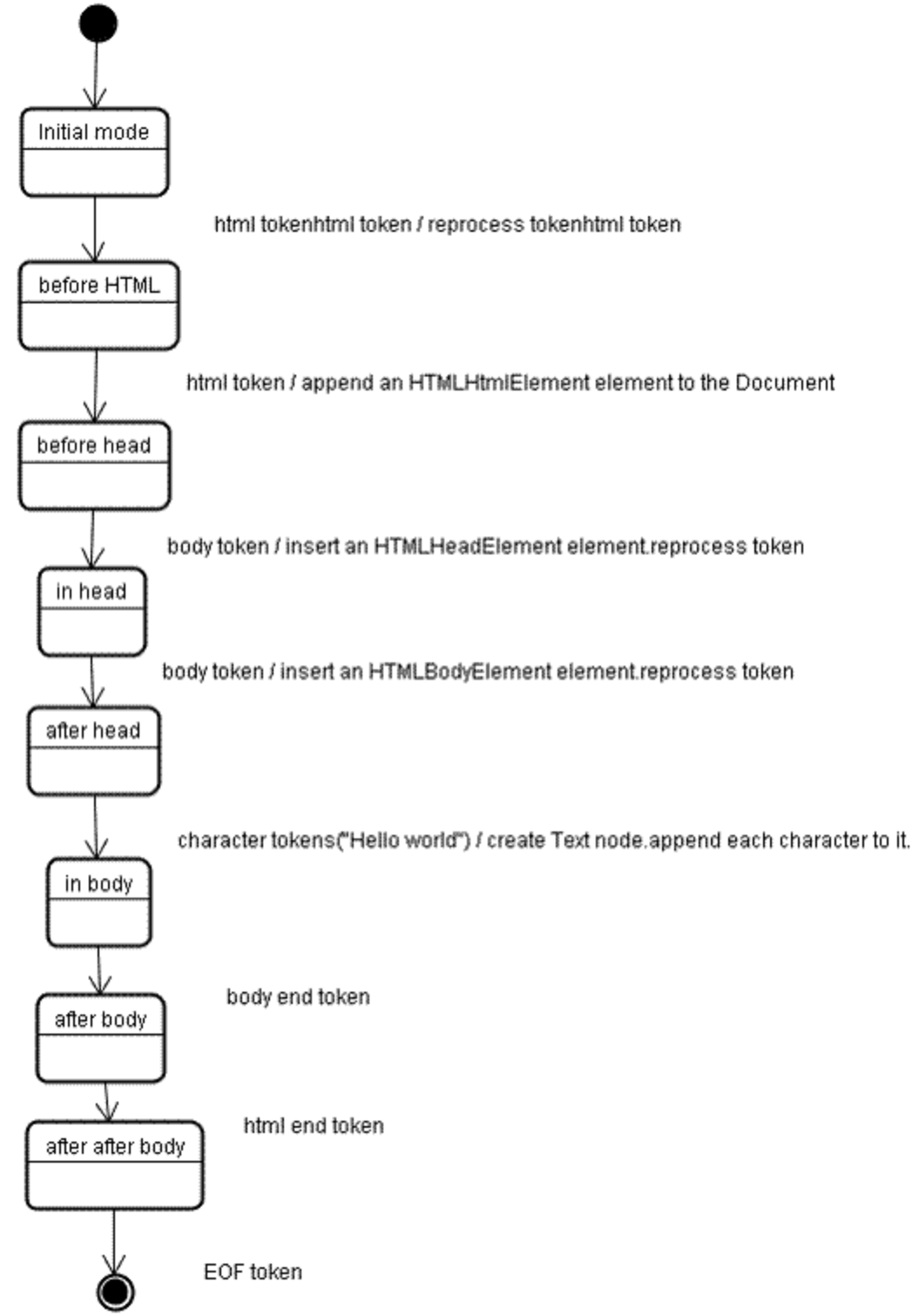


Figure : tree construction of example html

Actions when the parsing is finished

At this stage 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.

You can see the full algorithms for tokenization and tree construction in the HTML5 specification

Browsers' 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:

```
<html>
  <mytag>
  </mytag>
  <div>
  <p>
  </div>
    Really lousy HTML
  </p>
</html>
```

I must have violated 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.

The HTML5 specification does define some of these requirements. (WebKit

summarizes this nicely in the comment at the beginning of the HTML parser class.)

> *The parser parses tokenized input into the document, building up the document tree. If the document is well-formed, parsing it is straightforward.*
>
> *Unfortunately, we have to handle many HTML documents that are not well-formed, so the parser has to be tolerant about errors.*
>
> *We have to take care of at least the following error conditions:*
>
> 1. *The element being added is explicitly forbidden inside some outer tag. In this case we should close all tags up to the one which forbids the element, and add it afterwards.*
> 2. *We are not allowed to add the element directly. It could be that the person writing the document forgot some tag in between (or that the tag in between is optional). This could be the case with the following tags: HTML HEAD BODY TBODY TR TD LI (did I forget any?).*
> 3. *We want to add a block element inside an inline element. Close all inline elements up to the next higher block element.*
> 4. *If this doesn't help, close elements until we are allowed to add the element–or ignore the tag.*

Let's see some WebKit error tolerance examples:

</br> instead of

Some sites use </br> instead of
. In order to be compatible with IE and Firefox, WebKit treats this like
.
The code:

```
if (t->isCloseTag(brTag) && m_document->inCompatMode()) {
     reportError(MalformedBRError);
```

```
        t->beginTag = true;
}
```

Note that the error handling is internal: it won't be presented to the user.

A stray table

A stray table is a table inside another table, but not inside a table cell.

For example:

```
<table>
    <table>
        <tr><td>inner table</td></tr>
    </table>
    <tr><td>outer table</td></tr>
</table>
```

WebKit will change the hierarchy to two sibling tables:

```
<table>
    <tr><td>outer table</td></tr>
</table>
<table>
    <tr><td>inner table</td></tr>
</table>
```

The code:

```
if (m_inStrayTableContent && localName == tableTag)
        popBlock(tableTag);
```

WebKit uses a stack for the current element contents: it will pop the inner table out of the outer table stack. The tables will now be siblings.

Nested form elements

In case the user puts a form inside another form, the second form is ignored. The code:

```
if (!m_currentFormElement) {
        m_currentFormElement = new HTMLFormElement(formTag,
m_document);
}
```

A too deep tag hierarchy

The comment speaks for itself.

---

www.liceo.edu.mx is an example of a site that achieves a level of nesting of about 1500 tags, all from a bunch of <b>s. We will only allow at most 20 nested tags of the same type before just ignoring them all together.

```
bool HTMLParser::allowNestedRedundantTag(const AtomicString&
tagName)
{

unsigned i = 0;
for (HTMLStackElem* curr = m_blockStack;
         i < cMaxRedundantTagDepth && curr && curr->tagName ==
tagName;
     curr = curr->next, i++) { }
return i != cMaxRedundantTagDepth;
}
```

Misplaced html or body end tags

Again–the comment speaks for itself.

Support for really broken HTML. We never close the body tag, since some

stupid web pages close it before the actual end of the doc. Let's rely on the end() call to close things.

```
if (t->tagName == htmlTag || t->tagName == bodyTag )
        return;
```

So web authors beware–unless you want to appear as an example in a WebKit error tolerance code snippet–write well formed HTML.

CSS parsing

Remember the parsing concepts in the introduction? Well, unlike HTML, CSS is a context free grammar and can be parsed using the types of parsers described in the introduction. In fact the CSS specification defines CSS lexical and syntax grammar.

Let's see some examples:
The lexical grammar (vocabulary) is defined by regular expressions for each token:

```
comment   \/\*[^*]*\*+([^/*][^*]*\*+)*\/
num   [0-9]+|[0-9]*"."[0-9]+
nonascii  [\200-\377]
nmstart   [_a-z]|{nonascii}|{escape}
nmchar    [_a-z0-9-]|{nonascii}|{escape}
name    {nmchar}+
ident   {nmstart}{nmchar}*
```

"ident" is short for identifier, like a class name. "name" is an element id (that is referred by "#")

The syntax grammar is described in BNF.

```
ruleset
  : selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
selector
  : simple_selector [ combinator selector | S+ [ combinator?
selector ]? ]?
  ;
simple_selector
  : element_name [ HASH | class | attrib | pseudo ]*
  | [ HASH | class | attrib | pseudo ]+
  ;
class
  : '.' IDENT
  ;
element_name
  : IDENT | '*'
  ;
attrib
  : '[' S* IDENT S* [ [ '=' | INCLUDES | DASHMATCH ] S*
    [ IDENT | STRING ] S* ] ']'
  ;
pseudo
  : ':' [ IDENT | FUNCTION S* [IDENT S*] ')' ]
  ;
```

Explanation: A ruleset is this structure:

```
div.error, a.error {
  color:red;
  font-weight:bold;
}
```

div.error and a.error are selectors. The part inside the curly braces contains the rules that are applied by this ruleset. This structure is defined formally in this definition:

```
ruleset
```

```
: selector [ ',' S* selector ]*
  '{' S* declaration [ ';' S* declaration ]* '}' S*
;
```

This means a ruleset is a selector or optionally a number of selectors separated by a comma and spaces (S stands for white space). A ruleset contains curly braces and inside them a declaration or optionally a number of declarations separated by a semicolon. "declaration" and "selector" will be defined in the following BNF definitions.

WebKit CSS parser

WebKit uses Flex and Bison 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.

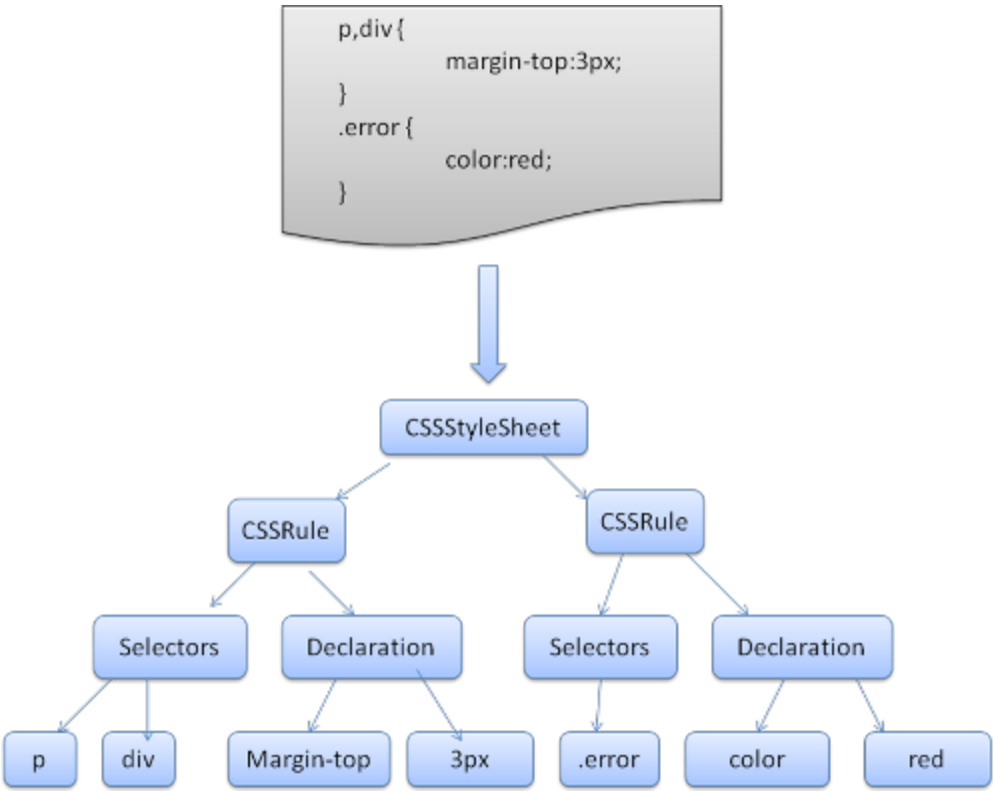


Figure : parsing CSS

The order of processing scripts and style sheets

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.

Speculative parsing

Both WebKit and Firefox do this 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.

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.

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.
WebKit's RenderObject class, the base class of the renderers, has the following definition:

```
class RenderObject{
  virtual void layout();
  virtual void paint(PaintInfo);
  virtual void rect repaintRect();
  Node* node;  //the DOM node
  RenderStyle* style;  // the computed style
  RenderLayer* containgLayer; //the containing z-index layer
}
```

Each renderer represents a rectangular area usually corresponding to a node's CSS box, as described by the CSS2 spec. It includes geometric information like width, height and position.
The box type is affected by the "display" value of the style attribute that is relevant to the node (see the style computation section). Here is WebKit code for deciding what type of renderer should be created for a DOM node, according to the display attribute:

```
RenderObject* RenderObject::createObject(Node* node,
RenderStyle* style)
{
    Document* doc = node->document();
    RenderArena* arena = doc->renderArena();
    ...
    RenderObject* o = 0;

    switch (style->display()) {
        case NONE:
            break;
        case INLINE:
            o = new (arena) RenderInline(node);
            break;
        case BLOCK:
            o = new (arena) RenderBlock(node);
            break;
        case INLINE_BLOCK:
            o = new (arena) RenderBlock(node);
            break;
        case LIST_ITEM:
            o = new (arena) RenderListItem(node);
```

```
            break;
        ...
    }

    return o;
}
```

The element type is also considered: for example, form controls and tables have special frames.
In WebKit if an element wants to create a special renderer, it will override the `createRenderer()` method. The renderers point to style objects that contains non geometric information.

The render tree relation to the DOM tree

The renderers correspond to DOM elements, but the relation is not one to one. Non-visual DOM elements will not be inserted in the render tree. An example is the "head" element. Also elements whose display value was assigned to "none" will not appear in the tree (whereas elements with "hidden" visibility will appear in the tree).

There are DOM elements which correspond to several visual objects. These are usually elements with complex structure that cannot be described by a single rectangle. For example, the "select" element has three renderers: one for the display area, one for the drop down list box and one for the button. Also when text is broken into multiple lines because the width is not sufficient for one line, the new lines will be added as extra renderers.
Another example of multiple renderers is broken HTML. According to the CSS spec an inline element must contain either only block elements or only inline elements. In the case of mixed content, anonymous block renderers will be created to wrap the inline elements.

Some render objects correspond to a DOM node but not in the same place in the tree. Floats and absolutely positioned elements are out of flow, placed in a different part of the tree, and mapped to the real frame. A placeholder frame is where they should have been.

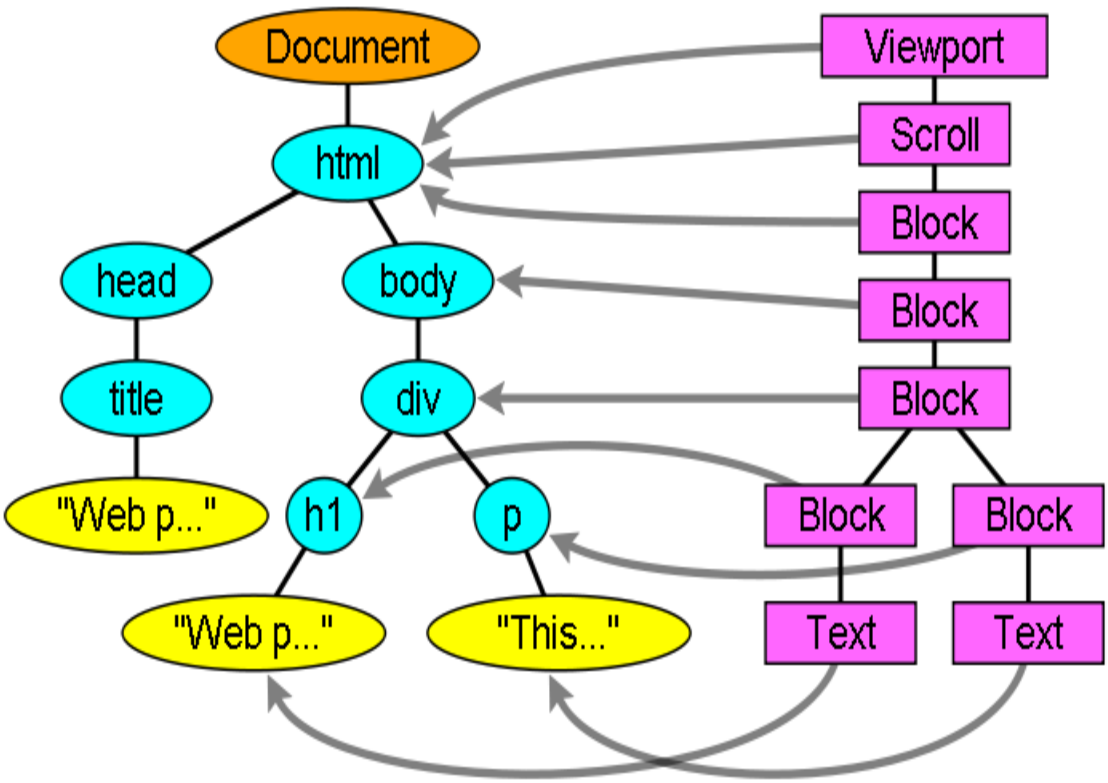


Figure : The render tree and the corresponding DOM tree (3.1). The "Viewport" is the initial containing block. In WebKit it will be the "RenderView" object

The flow of constructing the tree

In Firefox, the presentation is registered as a listener for DOM updates. The presentation delegates frame creation to the `FrameConstructor` and the constructor resolves style (see style computation) and creates a frame.

In WebKit the process of resolving the style and creating a renderer is called "attachment". Every DOM node has an "attach" method. Attachment is synchronous, node insertion to the DOM tree calls the new node "attach" method.

Processing the html and body tags results in the construction of the render tree root. The root render object corresponds to what the CSS spec calls the containing block: the top most block that contains all other blocks. Its dimensions are the viewport: the browser window display area dimensions.

Firefox calls it `ViewPortFrame` and WebKit calls it `RenderView`. This is the render object that the document points to. The rest of the tree is constructed as a DOM nodes insertion.

See the CSS2 spec on the processing model.

Style Computation

Building the render tree requires calculating the visual properties of each render object. This is done by calculating the style properties of each element.

The style includes style sheets of various origins, inline style elements and visual properties in the HTML (like the "bgcolor" property).The later is translated to matching CSS style properties.

The origins of style sheets are the browser's default style sheets, the style sheets provided by the page author and user style sheets–these are style sheets provided by the browser user (browsers let you define your favorite styles. In Firefox, for instance, this is done by placing a style sheet in the "Firefox Profile" folder).

Style computation brings up a few difficulties:

1. Style data is a very large construct, holding the numerous style properties, this can cause memory problems.
2. Finding the matching rules for each element can cause performance issues if it's not optimized. Traversing the entire rule list for each element to find matches is a heavy task. Selectors can have complex structure that can cause the matching process to start on a seemingly promising path that is proven to be futile and another path has to be tried.

   For example–this compound selector:

   ```
   div div div div{
   ```

```
  ...
}
```

Means the rules apply to a `<div>` who is the descendant of 3 divs. Suppose you want to check if the rule applies for a given `<div>` element. You choose a certain path up the tree for checking. You may need to traverse the node tree up just to find out there are only two divs and the rule does not apply. You then need to try other paths in the tree.

3. Applying the rules involves quite complex cascade rules that define the hierarchy of the rules.

Let's see how the browsers face these issues:

Sharing style data

WebKit nodes references style objects (RenderStyle) These objects can be shared by nodes in some conditions. The nodes are siblings or cousins and:

1. The elements must be in the same mouse state (e.g., one can't be in :hover while the other isn't)
2. Neither element should have an id
3. The tag names should match
4. The class attributes should match
5. The set of mapped attributes must be identical
6. The link states must match
7. The focus states must match
8. Neither element should be affected by attribute selectors, where affected is defined as having any selector match that uses an attribute selector in any position within the selector at all
9. There must be no inline style attribute on the elements
10. There must be no sibling selectors in use at all. WebCore simply throws a global switch when any sibling selector is encountered and disables style sharing for the entire document when they are present. This includes the + selector and selectors like :first-child and :last-child.

Firefox rule tree

Firefox has two extra trees for easier style computation: the rule tree and style context tree. WebKit also has style objects but they are not stored in a tree like the style context tree, only the DOM node points to its relevant style.

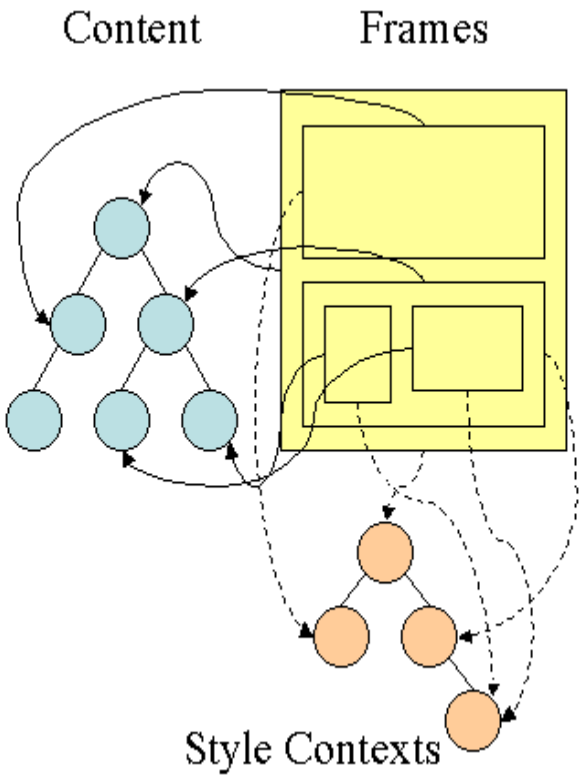


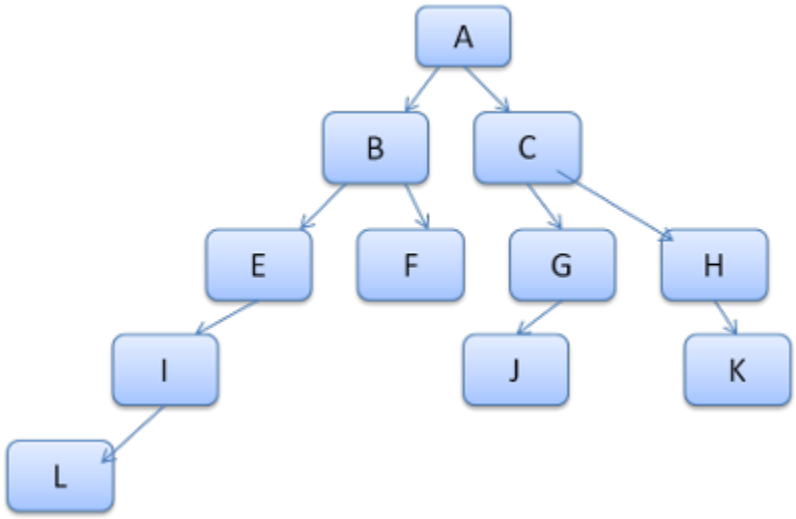
Figure : Firefox style context tree(2.2)

The style contexts contain end values. The values are computed by applying all the matching rules in the correct order and performing manipulations that transform them from logical to concrete values. For example, if the logical value is a percentage of the screen it will be calculated and transformed to absolute units. The rule tree idea is really clever. It enables sharing these values between nodes to avoid computing them again. This also saves space.

All the matched rules are stored in a tree. The bottom nodes in a path have higher priority. The tree contains all the paths for rule matches that were found. Storing the rules is done lazily. The tree isn't calculated at the

beginning for every node, but whenever a node style needs to be computed the computed paths are added to the tree.

The idea is to see the tree paths as words in a lexicon. Lets say we already computed this rule tree:



Suppose we need to match rules for another element in the content tree, and find out the matched rules (in the correct order) are B-E-I. We already have this path in the tree because we already computed path A-B-E-I-L. We will now have less work to do.

Let's see how the tree saves us work.

Division into structs

The style contexts are divided into structs. Those structs contain style information for a certain category like border or color. All the properties in a struct are either inherited or non inherited. Inherited properties are properties that unless defined by the element, are inherited from its parent. Non inherited properties (called "reset" properties) use default values if not defined.

The tree helps us by caching entire structs (containing the computed end values) in the tree. The idea is that if the bottom node didn't supply a definition for a struct, a cached struct in an upper node can be used.

Computing the style contexts using the rule tree

When computing the style context for a certain element, we first compute a path in the rule tree or use an existing one. We then begin to apply the rules in the path to fill the structs in our new style context. We start at the bottom node of the path–the one with the highest precedence (usually the most specific selector) and traverse the tree up until our struct is full. If there is no specification for the struct in that rule node, then we can greatly optimize–we go up the tree until we find a node that specifies it fully and simply point to it–that's the best optimization–the entire struct is shared. This saves computation of end values and memory.
If we find partial definitions we go up the tree until the struct is filled.

If we didn't find any definitions for our struct then, in case the struct is an "inherited" type, we point to the struct of our parent in the **context tree**. In this case we also succeeded in sharing structs. If it's a reset struct then default values will be used.

If the most specific node does add values then we need to do some extra calculations for transforming it to actual values. We then cache the result in the tree node so it can be used by children.

In case an element has a sibling or a brother that points to the same tree node then the **entire style context** can be shared between them.

Lets see an example: Suppose we have this HTML

```
<html>
  <body>
    <div class="err" id="div1">
      <p>
        this is a <span class="big"> big error </span>
        this is also a
        <span class="big"> very  big  error</span> error
      </p>
    </div>
    <div class="err" id="div2">another error</div>
  </body>
```

```
</html>
```

And the following rules:

```
1. div {margin:5px;color:black}
2. .err {color:red}
3. .big {margin-top:3px}
4. div span {margin-bottom:4px}
5. #div1 {color:blue}
6. #div2 {color:green}
```

To simplify things let's say we need to fill out only two structs: the color struct and the margin struct. The color struct contains only one member: the color The margin struct contains the four sides.
The resulting rule tree will look like this (the nodes are marked with the node name: the number of the rule they point at):

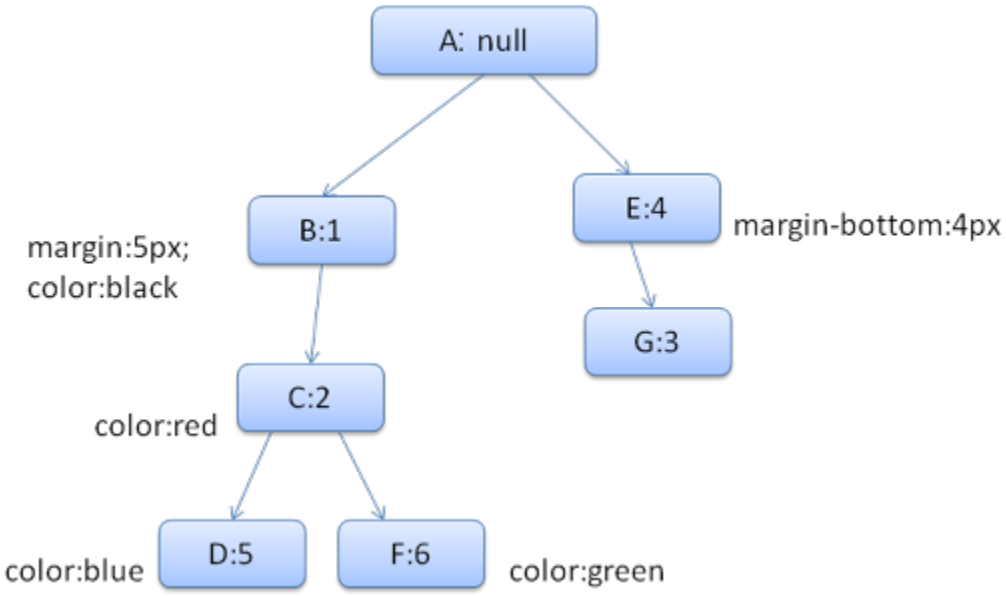


Figure : The rule tree

The context tree will look like this (node name: rule node they point to):

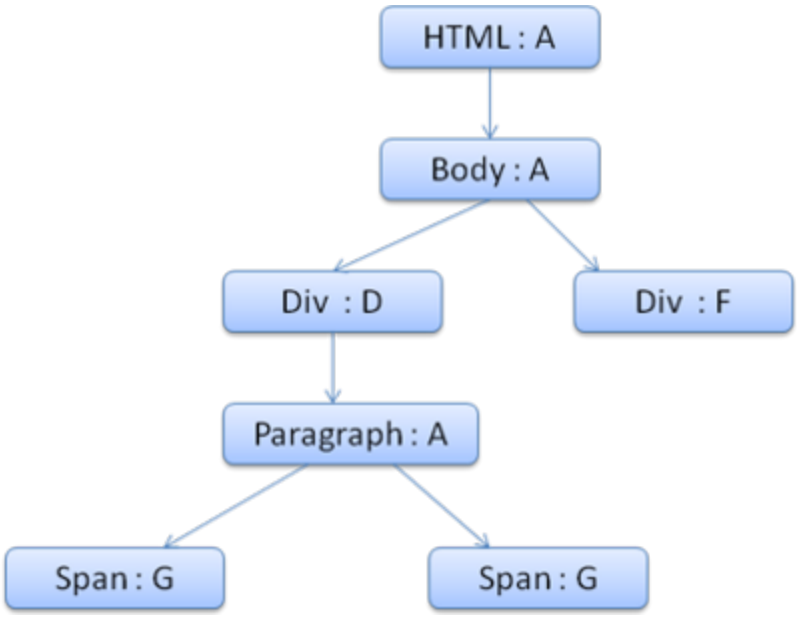


Figure : The context tree

Suppose we parse the HTML and get to the second tag. We need to create a style context for this node and fill its style structs.
We will match the rules and discover that the matching rules for the are 1, 2 and 6. This means there is already an existing path in the tree that our element can use and we just need to add another node to it for rule 6 (node F in the rule tree).
We will create a style context and put it in the context tree. The new style context will point to node F in the rule tree.

We now need to fill the style structs. We will begin by filling out the margin struct. Since the last rule node (F) doesn't add to the margin struct, we can go up the tree until we find a cached struct computed in a previous node insertion and use it. We will find it on node B, which is the uppermost node that specified margin rules.

We do have a definition for the color struct, so we can't use a cached struct. Since color has one attribute we don't need to go up the tree to fill other attributes. We will compute the end value (convert string to RGB etc) and cache the computed struct on this node.

The work on the second element is even easier. We will match the

rules and come to the conclusion that it points to rule G, like the previous span. Since we have siblings that point to the same node, we can share the entire style context and just point to the context of the previous span.

For structs that contain rules that are inherited from the parent, caching is done on the context tree (the color property is actually inherited, but Firefox treats it as reset and caches it on the rule tree).
For instance if we added rules for fonts in a paragraph:

```
p {font-family: Verdana; font size: 10px; font-weight: bold}
```

Then the paragraph element, which is a child of the div in the context tree, could have shared the same font struct as his parent. This is if no font rules where specified for the paragraph.

In WebKit, who does not have a rule tree, the matched declarations are traversed four times. First non-important high priority properties are applied (properties that should be applied first because others depend on them, such as display), then high priority important, then normal priority non-important, then normal priority important rules. This means that properties that appear multiple times will be resolved according to the correct cascade order. The last wins.

So to summarize: sharing the style objects (entirely or some of the structs inside them) solves issues 1 and 3. The Firefox rule tree also helps in applying the properties in the correct order.

Manipulating the rules for an easy match

There are several sources for style rules:

- CSS rules, either in external style sheets or in style elements.

    ```
    p {color: blue}
    ```

- Inline style attributes like

```
<p style="color: blue" />
```

- HTML visual attributes (which are mapped to relevant style rules)

```
<p bgcolor="blue" />
```

The last two are easily matched to the element since he owns the style attributes and HTML attributes can be mapped using the element as the key.

As noted previously in issue #2, the CSS rule matching can be trickier. To solve the difficulty, the rules are manipulated for easier access.

After parsing the style sheet, the rules are added to one of several hash maps, according to the selector. There are maps by id, by class name, by tag name and a general map for anything that doesn't fit into those categories. If the selector is an id, the rule will be added to the id map, if it's a class it will be added to the class map etc.
This manipulation makes it much easier to match rules. There is no need to look in every declaration: we can extract the relevant rules for an element from the maps. This optimization eliminates 95+% of the rules, so that they need not even be considered during the matching process(4.1).

Let's see for example the following style rules:

```
p.error {color: red}
#messageDiv {height: 50px}
div {margin: 5px}
```

The first rule will be inserted into the class map. The second into the id map and the third into the tag map.
For the following HTML fragment;

```
<p class="error">an error occurred </p>
<div id=" messageDiv">this is a message</div>
```

We will first try to find rules for the p element. The class map will contain an "error" key under which the rule for "p.error" is found. The div element will have relevant rules in the id map (the key is the id) and the tag map. So the only work left is finding out which of the rules that were extracted by the keys really match.
For example if the rule for the div was

```
table div {margin: 5px}
```

it will still be extracted from the tag map, because the key is the rightmost selector, but it would not match our div element, who does not have a table ancestor.

Both WebKit and Firefox do this manipulation.

Applying the rules in the correct cascade order

The style object has properties corresponding to every visual attribute (all CSS attributes but more generic). If the property is not defined by any of the matched rules, then some properties can be inherited by the parent element style object. Other properties have default values.

The problem begins when there is more than one definition–here comes the cascade order to solve the issue.

Style sheet cascade order

A declaration for a style property can appear in several style sheets, and several times inside a style sheet. This means the order of applying the rules is very important. This is called the "cascade" order. According to CSS2 spec, the cascade order is (from low to high):

1. Browser declarations
2. User normal declarations
3. Author normal declarations
4. Author important declarations
5. User important declarations

The browser declarations are least important and the user overrides the author only if the declaration was marked as important. Declarations with the same order will be sorted by specificity and then the order they are specified. The HTML visual attributes are translated to matching CSS declarations . They are treated as author rules with low priority.

Specificity

The selector specificity is defined by the CSS2 specification as follows:

- count 1 if the declaration it is from is a 'style' attribute rather than a rule with a selector, 0 otherwise (= a)
- count the number of ID attributes in the selector (= b)
- count the number of other attributes and pseudo-classes in the selector (= c)
- count the number of element names and pseudo-elements in the selector (= d)

Concatenating the four numbers a-b-c-d (in a number system with a large base) gives the specificity.

The number base you need to use is defined by the highest count you have in one of the categories.
For example, if a=14 you can use hexadecimal base. In the unlikely case where a=17 you will need a 17 digits number base. The later situation can happen with a selector like this: html body div div p ... (17 tags in your selector.. not very likely).

Some examples:

```
 *              {}  /* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0
*/
 li             {}  /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1
*/
 li:first-line {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2
*/
 ul li          {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2
*/
 ul ol+li       {}  /* a=0 b=0 c=0 d=3 -> specificity = 0,0,0,3
*/
 h1 + *[rel=up]{}  /* a=0 b=0 c=1 d=1 -> specificity = 0,0,1,1
*/
 ul ol li.red   {}  /* a=0 b=0 c=1 d=3 -> specificity = 0,0,1,3
*/
 li.red.level   {}  /* a=0 b=0 c=2 d=1 -> specificity = 0,0,2,1
*/
 #x34y          {}  /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0
*/
 style=""           /* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0
*/
```

Sorting the rules

After the rules are matched, they are sorted according to the cascade rules. WebKit uses bubble sort for small lists and merge sort for big ones. WebKit implements sorting by overriding the ">" operator for the rules:

```
static bool operator >(CSSRuleData& r1, CSSRuleData& r2)
{
    int spec1 = r1.selector()->specificity();
    int spec2 = r2.selector()->specificity();
    return (spec1 == spec2) : r1.position() > r2.position() :
spec1 > spec2;
}
```

Gradual process

WebKit uses a flag that marks if all top level style sheets (including @imports) have been loaded. If the style is not fully loaded when attaching, place

holders are used and it is marked in the document, and they will be recalculated once the style sheets were loaded.

Layout

When the renderer is created and added to the tree, it does not have a position and size. Calculating these values is called layout or reflow.

HTML uses a flow based layout model, meaning that most of the time it is possible to compute the geometry in a single pass. Elements later ``in the flow'' typically do not affect the geometry of elements that are earlier ``in the flow'', so layout can proceed left-to-right, top-to-bottom through the document. There are exceptions: for example, HTML tables may require more than one pass (3.5).

The coordinate system is relative to the root frame. Top and left coordinates are used.

Layout is a recursive process. It begins at the root renderer, which corresponds to the `<html>` element of the HTML document. Layout continues recursively through some or all of the frame hierarchy, computing geometric information for each renderer that requires it.

The position of the root renderer is 0,0 and its dimensions are the viewport– the visible part of the browser window.

All renderers have a "layout" or "reflow" method, each renderer invokes the layout method of its children that need layout.

Dirty bit system

In order not to do a full layout for every small change, browsers use a "dirty bit" system. A renderer that is changed or added marks itself and its children as "dirty": needing layout.

There are two flags: "dirty", and "children are dirty" which means that although the renderer itself may be OK, it has at least one child that needs a

layout.

Global and incremental layout

Layout can be triggered on the entire render tree–this is "global" layout. This can happen as a result of:

1. A global style change that affects all renderers, like a font size change.
2. As a result of a screen being resized

Layout can be incremental, only the dirty renderers will be laid out (this can cause some damage which will require extra layouts).
Incremental layout is triggered (asynchronously) when renderers are dirty. For example when new renderers are appended to the render tree after extra content came from the network and was added to the DOM tree.

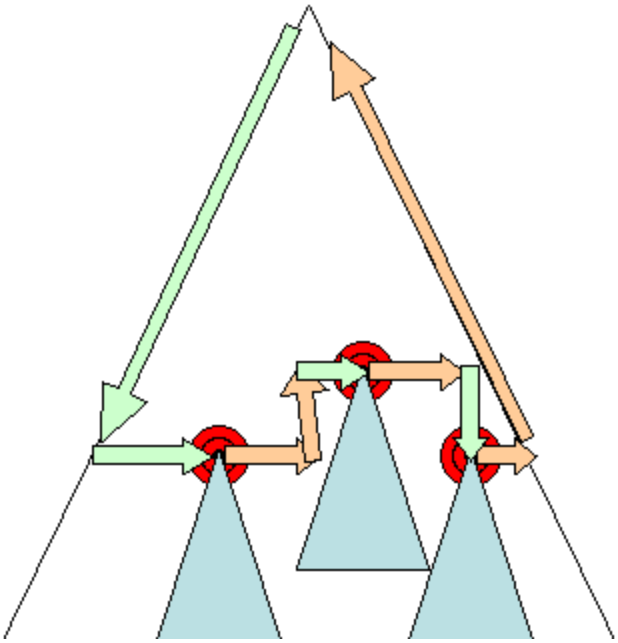


Figure : Incremental layout–only dirty renderers and their children are laid out (3.6)

Asynchronous and Synchronous layout

Incremental layout is done asynchronously. Firefox queues "reflow

commands" for incremental layouts and a scheduler triggers batch execution of these commands. WebKit also has a timer that executes an incremental layout–the tree is traversed and "dirty" renderers are layout out.
Scripts asking for style information, like "offsetHeight" can trigger incremental layout synchronously.
Global layout will usually be triggered synchronously.
Sometimes layout is triggered as a callback after an initial layout because some attributes, like the scrolling position changed.

Optimizations

When a layout is triggered by a "resize" or a change in the renderer position(and not size), the renders sizes are taken from a cache and not recalculated..
In some cases only a sub tree is modified and layout does not start from the root. This can happen in cases where the change is local and does not affect its surroundings–like text inserted into text fields (otherwise every keystroke would trigger a layout starting from the root).

The layout process

The layout usually has the following pattern:

1. Parent renderer determines its own width.
2. Parent goes over children and:
    1. Place the child renderer (sets its x and y).
    2. Calls child layout if needed–they are dirty or we are in a global layout, or for some other reason–which calculates the child's height.
3. Parent uses children's accumulative heights and the heights of margins and padding to set its own height–this will be used by the parent renderer's parent.
4. Sets its dirty bit to false.

Firefox uses a "state" object(nsHTMLReflowState) as a parameter to layout (termed "reflow"). Among others the state includes the parents width.
The output of the Firefox layout is a "metrics" object(nsHTMLReflowMetrics). It will contain the renderer computed height.

Width calculation

The renderer's width is calculated using the container block's width, the renderer's style "width" property, the margins and borders.
For example the width of the following div:

```
<div style="width: 30%"/>
```

Would be calculated by WebKit as the following(class RenderBox method calcWidth):

- The container width is the maximum of the containers availableWidth and 0. The availableWidth in this case is the contentWidth which is calculated as:

  ```
  clientWidth() - paddingLeft() - paddingRight()
  ```

  clientWidth and clientHeight represent the interior of an object excluding border and scrollbar.
- The elements width is the "width" style attribute. It will be calculated as an absolute value by computing the percentage of the container width.
- The horizontal borders and paddings are now added.

So far this was the calculation of the "preferred width". Now the minimum and maximum widths will be calculated.
If the preferred width is greater then the maximum width, the maximum width is used. If it is less then the minimum width (the smallest unbreakable unit) then the minimum width is used.

The values are cached in case a layout is needed, but the width does not change.

Line Breaking

When a renderer in the middle of a layout decides that it needs to break, the renderer stops and propagates to the layout's parent that it needs to be broken. The parent creates the extra renderers and calls layout on them.

Painting

In the painting stage, the render tree is traversed and the renderer's "paint()" method is called to display content on the screen. Painting uses the UI infrastructure component.

Global and Incremental

Like layout, painting can also be global–the entire tree is painted–or incremental. In incremental painting, some of the renderers change in a way that does not affect the entire tree. The changed renderer invalidates its rectangle on the screen. This causes the OS to see it as a "dirty region" and generate a "paint" event. The OS does it cleverly and coalesces several regions into one. In Chrome it is more complicated because the renderer is in a different process then the main process. Chrome simulates the OS behavior to some extent. The presentation listens to these events and delegates the message to the render root. The tree is traversed until the relevant renderer is reached. It will repaint itself (and usually its children).

The painting order

CSS2 defines the order of the painting process. This is actually the order in which the elements are stacked in the stacking contexts. This order affects painting since the stacks are painted from back to front. The stacking order of a block renderer is:

1. background color
2. background image
3. border
4. children
5. outline

Firefox display list

Firefox goes over the render tree and builds a display list for the painted rectangular. It contains the renderers relevant for the rectangular, in the right painting order (backgrounds of the renderers, then borders etc). That way the tree needs to be traversed only once for a repaint instead of several times–painting all backgrounds, then all images, then all borders etc.

Firefox optimizes the process by not adding elements that will be hidden, like elements completely beneath other opaque elements.

WebKit rectangle storage

Before repainting, WebKit saves the old rectangle as a bitmap. It then paints only the delta between the new and old rectangles.

Dynamic changes

The browsers try to do the minimal possible actions in response to a change. So changes to an elements color will cause only repaint of the element. Changes to the element position will cause layout and repaint of the element, its children and possibly siblings. Adding a DOM node will cause layout and repaint of the node. Major changes, like increasing font size of the "html" element, will cause invalidation of caches, relayout and repaint of the entire tree.

The rendering engine's threads

The rendering engine is single threaded. Almost everything, except network operations, happens in a single thread. In Firefox and Safari this is the main

thread of the browser. In Chrome it's the tab process main thread. Network operations can be performed by several parallel threads. The number of parallel connections is limited (usually 2–6 connections).

Event loop

The browser main thread is an event loop. It's an infinite loop that keeps the process alive. It waits for events (like layout and paint events) and processes them. This is Firefox code for the main event loop:

```
while (!mExiting)
    NS_ProcessNextEvent(thread);
```

CSS2 visual model

The canvas

According to the CSS2 specification, the term canvas describes "the space where the formatting structure is rendered": where the browser paints the content. The canvas is infinite for each dimension of the space but browsers choose an initial width based on the dimensions of the viewport.

According to www.w3.org/TR/CSS2/zindex.html, the canvas is transparent if contained within another, and given a browser defined color if it is not.

CSS Box model

The CSS box model describes the rectangular boxes that are generated for elements in the document tree and laid out according to the visual formatting model.
Each box has a content area (e.g. text, an image, etc.) and optional surrounding padding, border, and margin areas.

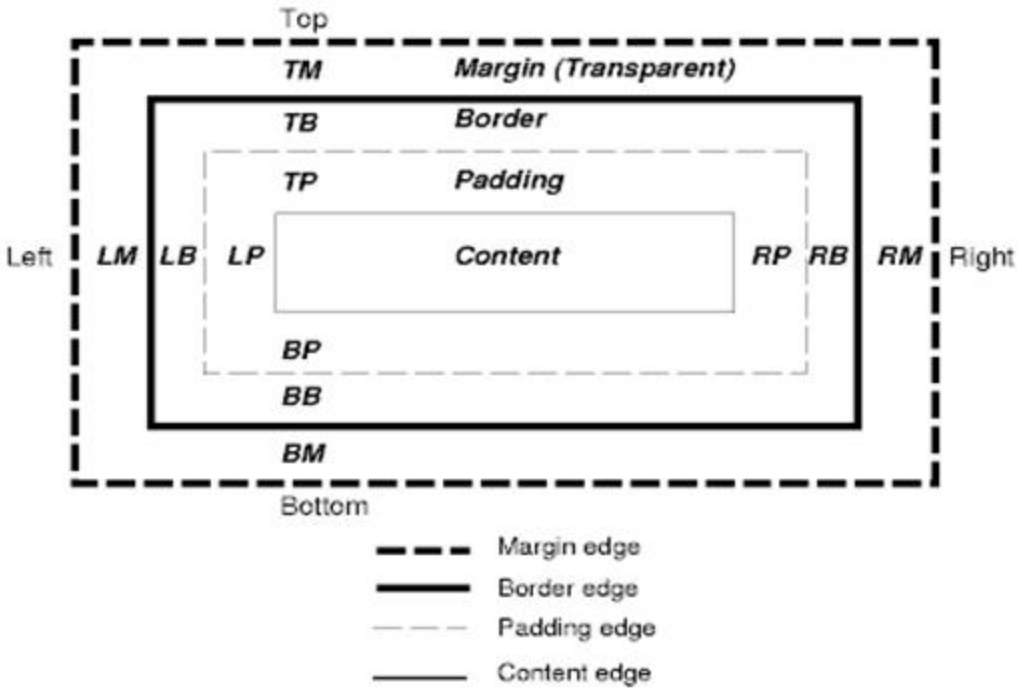


Figure : CSS2 box model

Each node generates 0..n such boxes.
All elements have a "display" property that determines the type of box that will be generated. Examples:

```
block: generates a block box.
inline: generates one or more inline boxes.
none: no box is generated.
```

The default is inline but the browser style sheet may set other defaults. For example: the default display for the "div" element is block.
You can find a default style sheet example here: www.w3.org/TR/CSS2/sample.html

Positioning scheme

There are three schemes:

1. Normal: the object is positioned according to its place in the document.

This means its place in the render tree is like its place in the DOM tree and laid out according to its box type and dimensions
2. Float: the object is first laid out like normal flow, then moved as far left or right as possible
3. Absolute: the object is put in the render tree in a different place than in the DOM tree

The positioning scheme is set by the "position" property and the "float" attribute.

- static and relative cause a normal flow
- absolute and fixed cause absolute positioning

In static positioning no position is defined and the default positioning is used. In the other schemes, the author specifies the position: top, bottom, left, right.

The way the box is laid out is determined by:

- Box type
- Box dimensions
- Positioning scheme
- External information such as image size and the size of the screen

Box types

Block box: forms a block–has its own rectangle in the browser window.

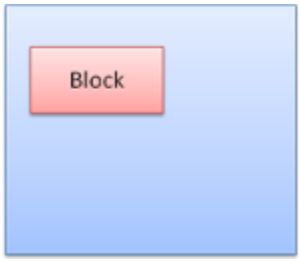


Figure : Block box

Inline box: does not have its own block, but is inside a containing block.

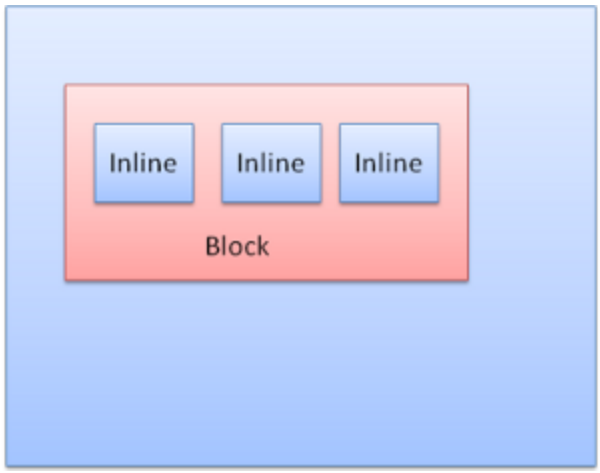


Figure : Inline boxes

Blocks are formatted vertically one after the other. Inlines are formatted horizontally.

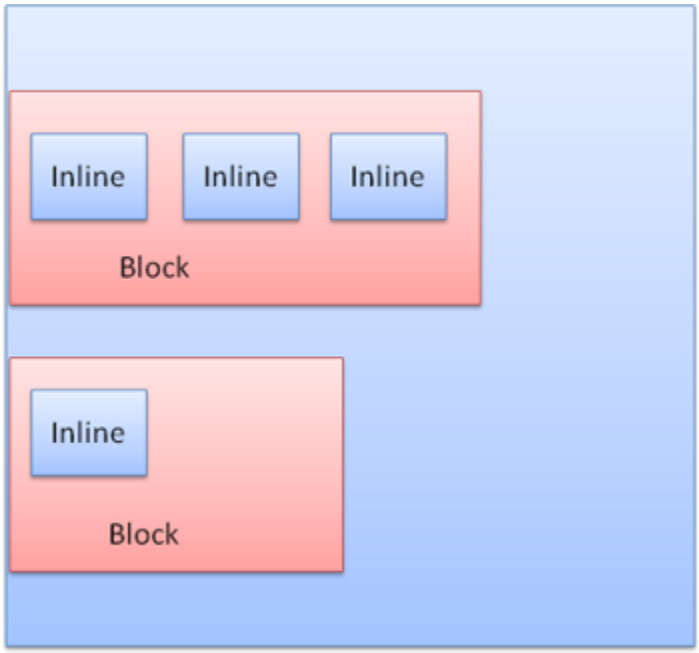


Figure : Block and Inline formatting

Inline boxes are put inside lines or "line boxes". The lines are at least as tall as the tallest box but can be taller, when the boxes are aligned "baseline"– meaning the bottom part of an element is aligned at a point of another box other then the bottom. If the container width is not enough, the inlines will be

put on several lines. This is usually what happens in a paragraph.

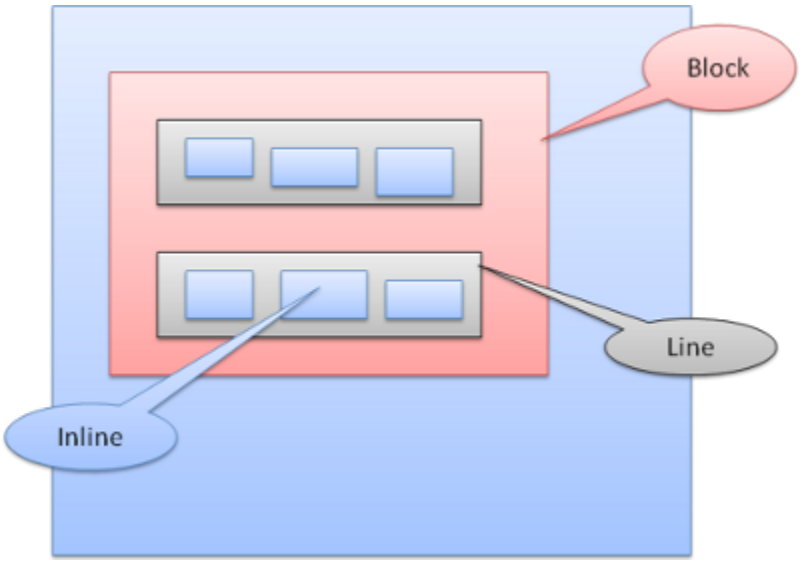


Figure: Lines

Positioning

Relative

Relative positioning–positioned like usual and then moved by the required delta.

```
<html>
        <div>
                <span>1</span>
                <span>2</span>
                <span style="position:relative;left:5px">3</span>
        </div>
</html>
```

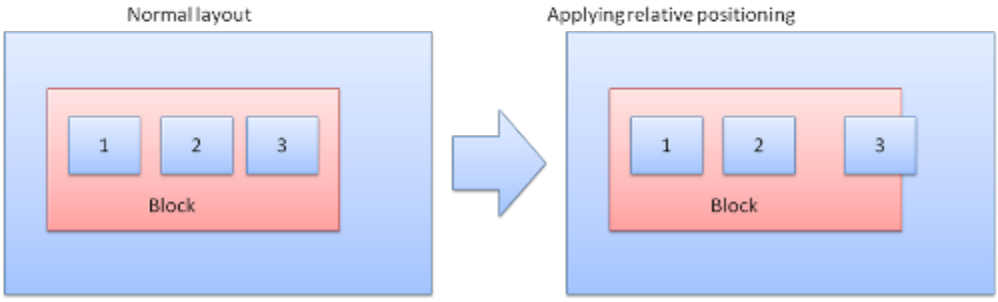


Figure: Relative positioning

Floats

A float box is shifted to the left or right of a line. The interesting feature is that the other boxes flow around it The HTML:

```
<p>
  <img style="float: right" src="images/image.gif" width="100"
height="100">
  Lorem ipsum dolor sit amet, consectetuer...
</p>
```

Will look like:

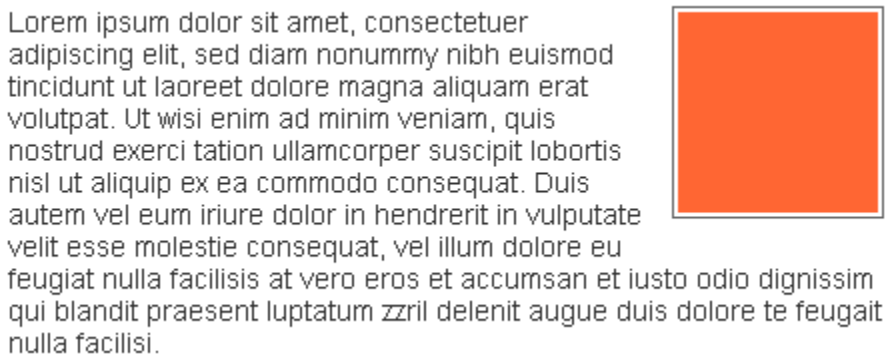


Figure : Float

Absolute and fixed

The layout is defined exactly regardless of the normal flow. The element does not participate in the normal flow. The dimensions are relative to the container. In fixed, the container is the viewport.

```
<html>
        <div>
                <span>1</span>
                 <span>2</span>
                 <span style="position:fixed;top:5px,left:5px">3</span>
        </div>
</html>
```

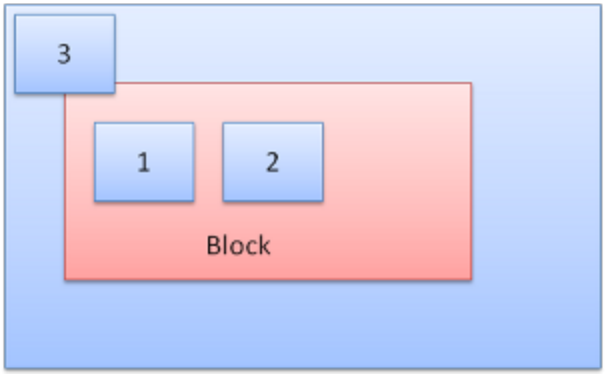


Figure : Fixed positioning

Note: the fixed box will not move even when the document is scrolled!

Layered representation

This is specified by the z-index CSS property. It represents the third dimension of the box: its position along the "z axis".

The boxes are divided into stacks (called stacking contexts). In each stack the back elements will be painted first and the forward elements on top, closer to the user. In case of overlap the foremost element will hide the former element.
The stacks are ordered according to the z-index property. Boxes with "z-index" property form a local stack. The viewport has the outer stack.

Example:

```
<style type="text/css">
      div {
```

```
        position: absolute;
        left: 2in;
        top: 2in;
      }
</style>

<p>
    <div
         style="z-index: 3;background-color:red; width: 1in;
height: 1in; ">
    </div>
    <div
         style="z-index: 1;background-color:green;width: 2in;
height: 2in;">
    </div>
 </p>
```

The result will be this:

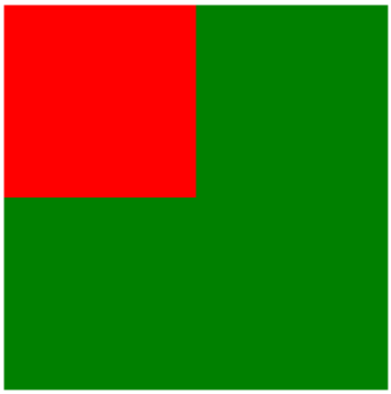


Figure : Fixed positioning

Although the red div precedes the green one in the markup, and would have been painted before in the regular flow, the z-index property is higher, so it is more forward in the stack held by the root box.

Resources

1. Browser architecture

1. Grosskurth, Alan. A Reference Architecture for Web Browsers (pdf)
2. Gupta, Vineet. How Browsers Work–Part 1–Architecture

2. Parsing

1. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools (aka the "Dragon book"), Addison-Wesley, 1986
2. Rick Jelliffe. The Bold and the Beautiful: two new drafts for HTML 5.

3. Firefox

1. L. David Baron, Faster HTML and CSS: Layout Engine Internals for Web Developers.
2. L. David Baron, Faster HTML and CSS: Layout Engine Internals for Web Developers (Google tech talk video)
3. L. David Baron, Mozilla's Layout Engine
4. L. David Baron, Mozilla Style System Documentation
5. Chris Waterson, Notes on HTML Reflow
6. Chris Waterson, Gecko Overview
7. Alexander Larsson, The life of an HTML HTTP request

4. WebKit

1. David Hyatt, Implementing CSS(part 1)
2. David Hyatt, An Overview of WebCore
3. David Hyatt, WebCore Rendering
4. David Hyatt, The FOUC Problem

5. W3C Specifications

1. HTML 4.01 Specification
2. W3C HTML5 Specification
3. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification

6. Browsers build instructions

1. Firefox. https://developer.mozilla.org/en/Build_Documentation
2. WebKit. http://webkit.org/building/build.html

Tali Garsiel is a developer in Israel. She started as a web developer in 2000, and became aquainted with Netscape's "evil" layer model. Just like Richard Feynmann, she had a fascination for figuring out how things work so she began digging into browser internals and documenting what she found. Tali also has published a short guide on client-side performance.

Translations

This page has been translated into Japanese, twice! How Browsers Work–Behind the Scenes of Modern Web Browsers (ja) by @_kosei_ and also ブラウザってどうやって動いてるの？(モダンWEBブラウザシーンの裏側 by @ikeike443 and @kiyoto01. Thanks everyone!