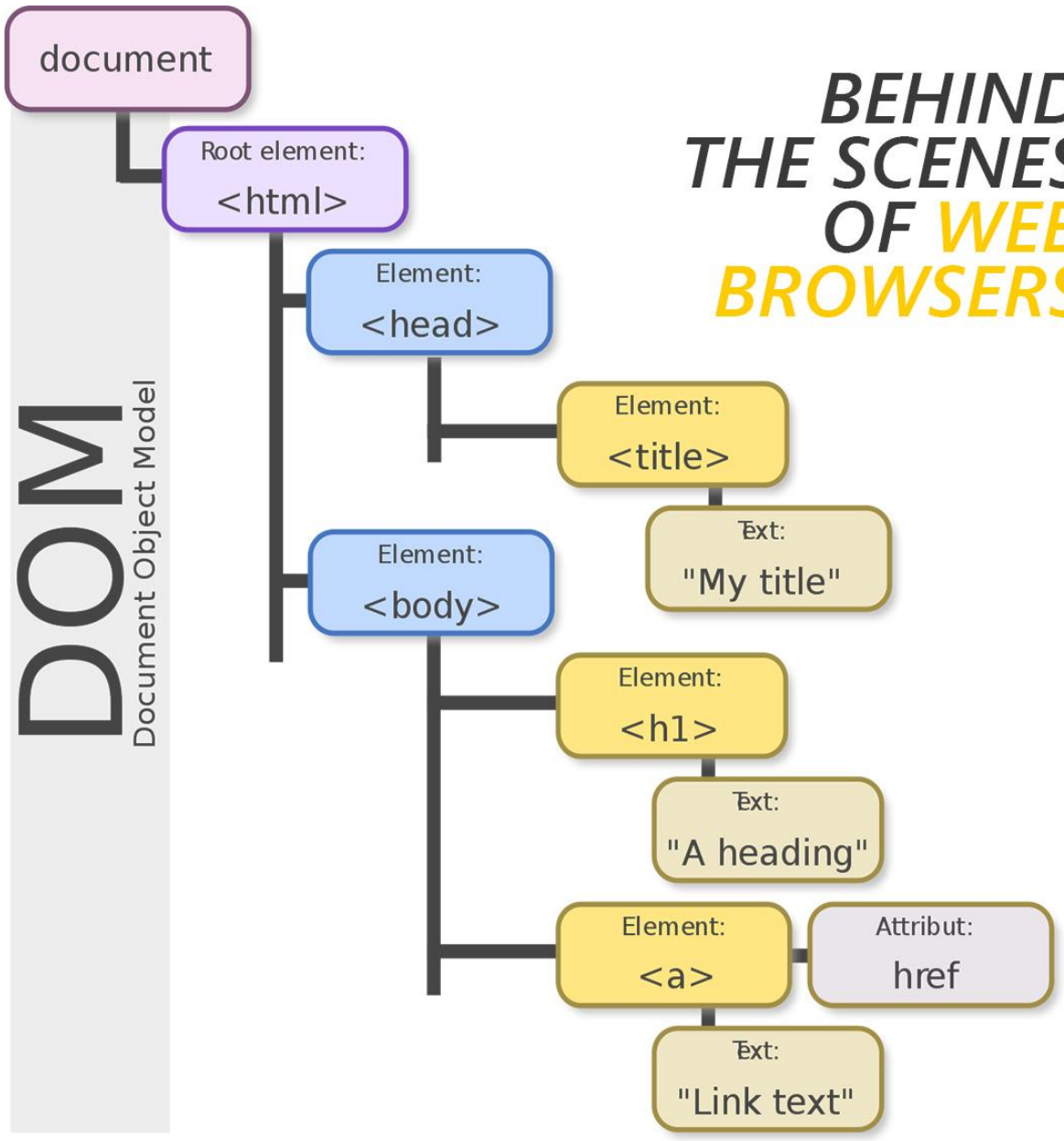
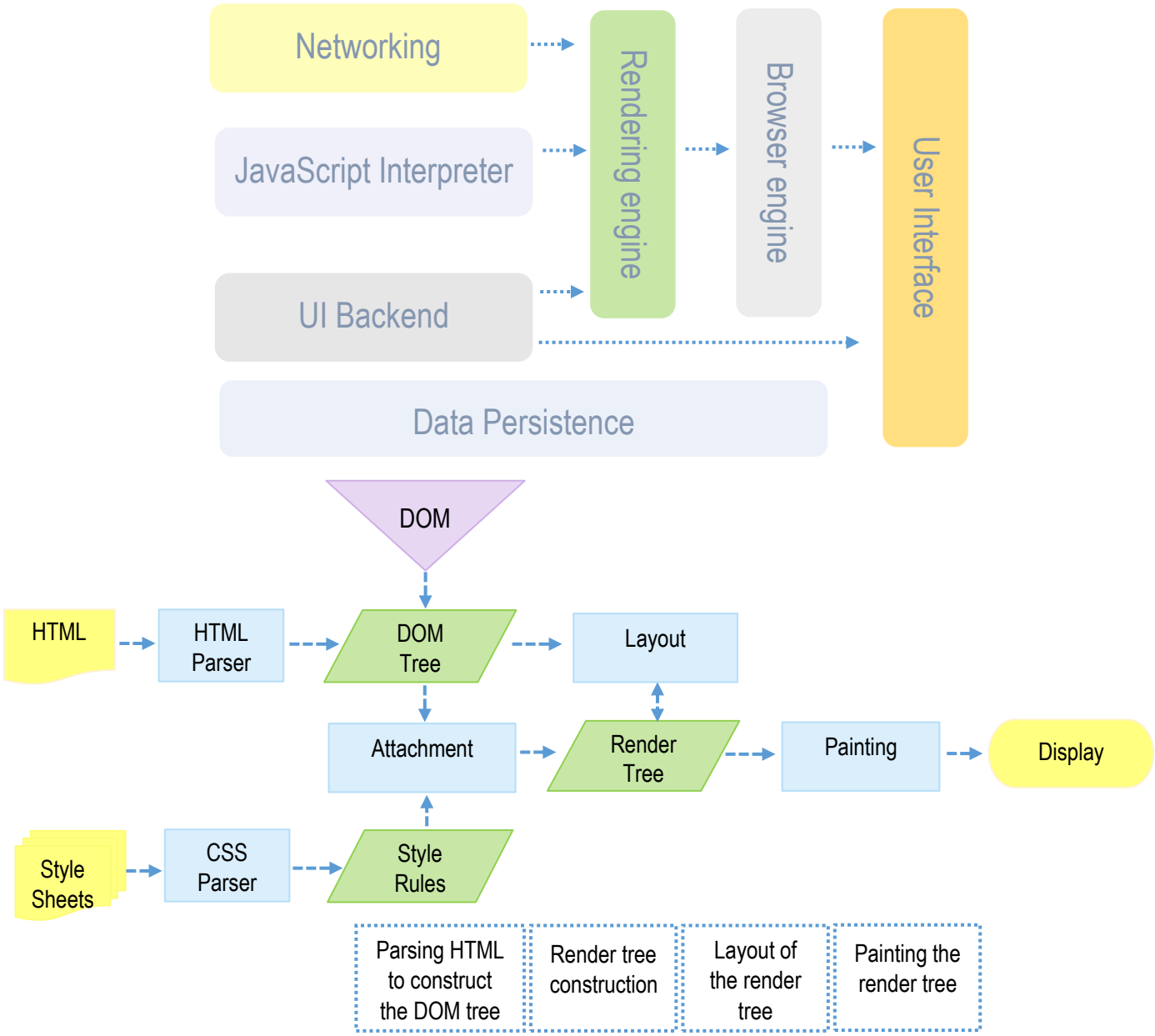


There are five major browsers used today - Safari, Chrome, Firefox, Internet Explorer and Opera. Before starting to work with different JavaScript frameworks, planninng the project structure and just simply start coding, development part starts from user experience relying on process starting from url typing in the address bar till reaching the page on the browser screen.

Table of Contents

1. The main stages



Webkit main flow

1. The user interface - this includes the address bar, back/forward button, bookmarking menu etc.;
2. The browser engine - the interface for querying and manipulating the rendering engine.

The rendering engine will start getting the contents of the requested document from the networking layer. The rendering engine will start parsing the HTML document and turn the tags to DOM nodes in a tree called the "content tree". It will parse the style data, both in external CSS files and in style

elements. The 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.

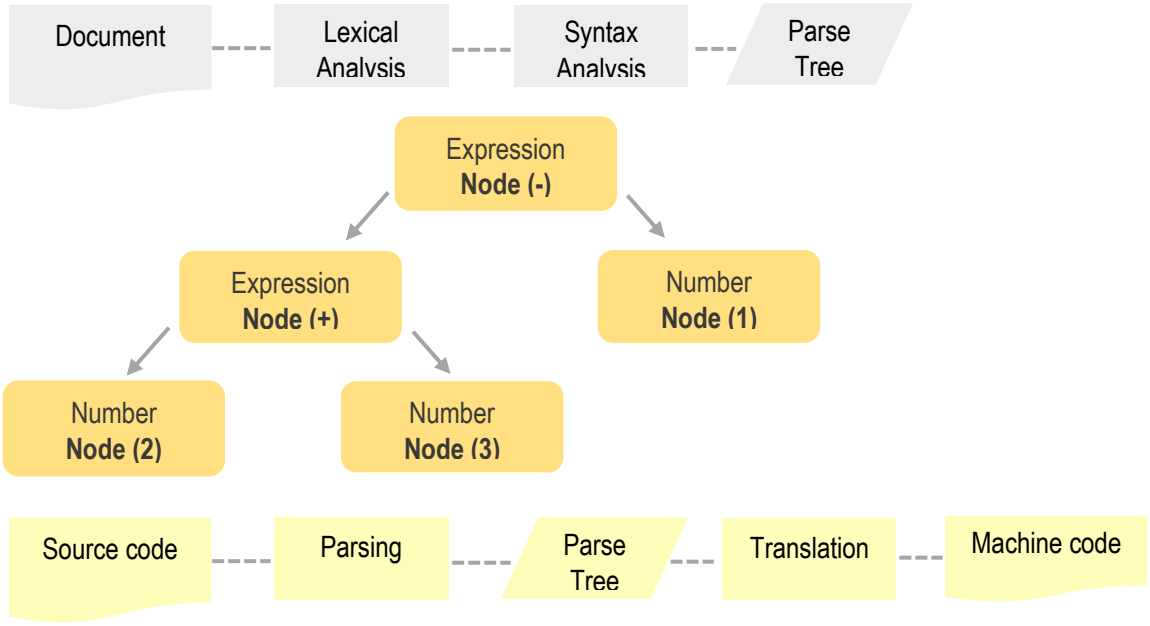
After the construction of the render tree it goes through a "layout" process. This means giving each node the exact coordinates where it should appear on the screen. The next stage is painting - the render tree will be traversed and each node will be painted using the UI backend layer.

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.

2. Parsing

Parsing a document means translating it to some structure that makes sense - something the code can understand and use. The result of parsing is usually a tree of nodes that represent the structure of the document. It is called a parse tree or a syntax tree.

Example - parsing the expression "2 + 3 - 1" could return this tree:



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.

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.

Many times 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 a source code into machine code first parses it into a parse tree and then translates the tree into a machine code document.

2.2. 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:

- The language syntax building blocks are expressions, terms and operations.

- Our language can include any number of expressions.

- A expression is defined as a "term" followed by an "operation" followed by another term

- An operation is a plus token or a minus token

- 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 second 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.

2.3. Types of parsers

There are two basic types of parsers - top down parsers and bottom up parsers. An intuitive explanation is that top down parsers see the high level structure of the syntax and try to match one of them. 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:

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 parsers stack.

3. 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.

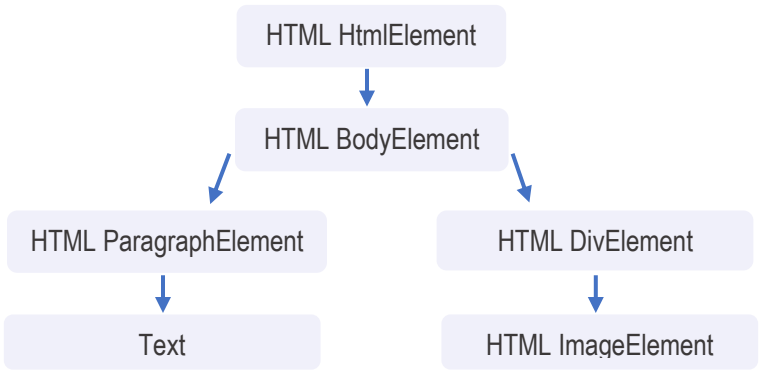
As we saw in the previous sections, HTML cannot be parsed using the regular top down or bottom up parsers.The reasons are:

- The forgiving nature of the language.

-The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.

The parsing process in reentrant. Usually the source doesn't change during parsing, but in HTML, script tags containing "document.write" can add extra tokens, so the parsing process actually modifies the input.

```
<html>
    <body>
        <p>
            Hello World
        </p>
    </body>
</html>
```

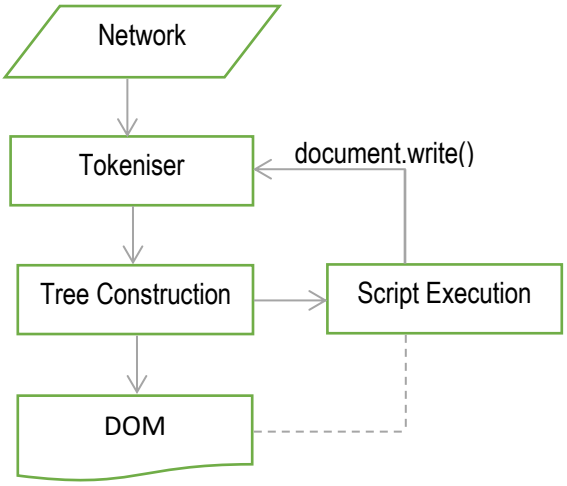


4. Tokenization and tree construction

Unable to use the regular parsing techniques, browsers create custom parsers for parsing HTML. The parsing algorithm is described in details 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.



4.1. 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 bring fully, so let's see a simple example that will help us understand the principal.

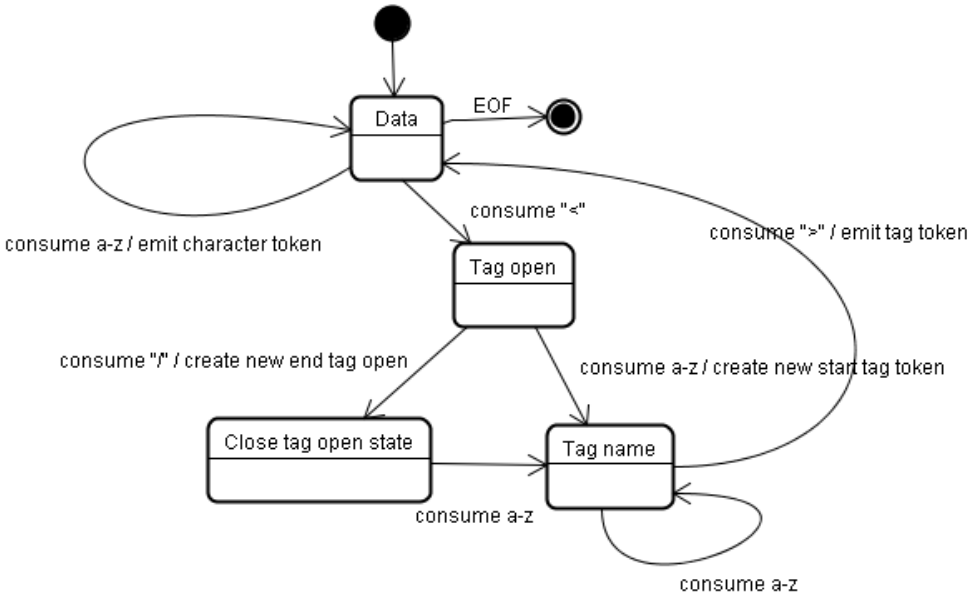
Basic example - tokenizing the following HTML:

```
<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 change 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.


Data
EOF
consume a-z / emit character token
consume "<"
consume ">" / emit tag token
Tag open
consume "/" / create new end tag open
consume a-z / create new start tag token
Close tag open state
Tag name
consume a-z
consume a-z

4.2. 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. Except of adding the element to the DOM tree it is also added to a 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>
```

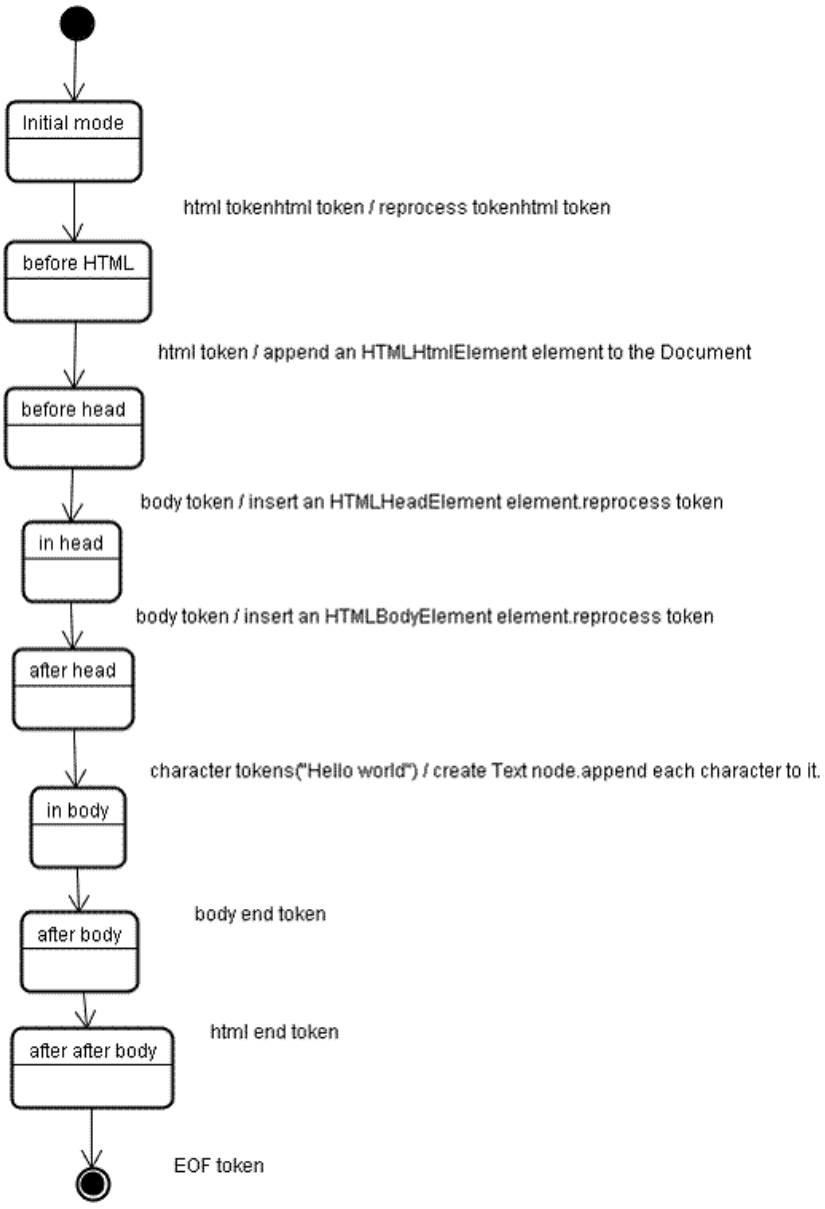
The input to the tree construction stage is a sequence of tokens from the tokenization stage The first mode is the "initial mode". Receiving the html token will cause a move to the "before html" mode and a reprocessing of the token in that mode. This will cause a creation of the HTMLHtmlElement element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

The state will be changed to "before head". We receive the "body" token. 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.



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 who 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 HTML5 specification - http://www.w3.org/TR/html5/syntax.html#html-parser

5. 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 was executed. If the script is external then the resource must be first fetched from the network - this is also done synchronously, the parsing halts until the resource is fetched. This was the model for many years and is also specified in HTML 4 and 5 specifications. Authors could mark the script as "defer" and thus it will not halt the document parsing and will execute after it is parsed. HTML5 adds an option to mark the script as asynchronous so it will be parsed and executed by a different thread.

5.1. 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. These way resources can be loaded on parallel connections and the overall speed is better. Note - the speculative parser doesn't modify the DOM tree and leaves that to the main parser, it only parses references to external resources like external scripts, style sheets and images.

5.2. 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 for certain style properties that may be effected by unloaded style sheets.

6. Render tree construction

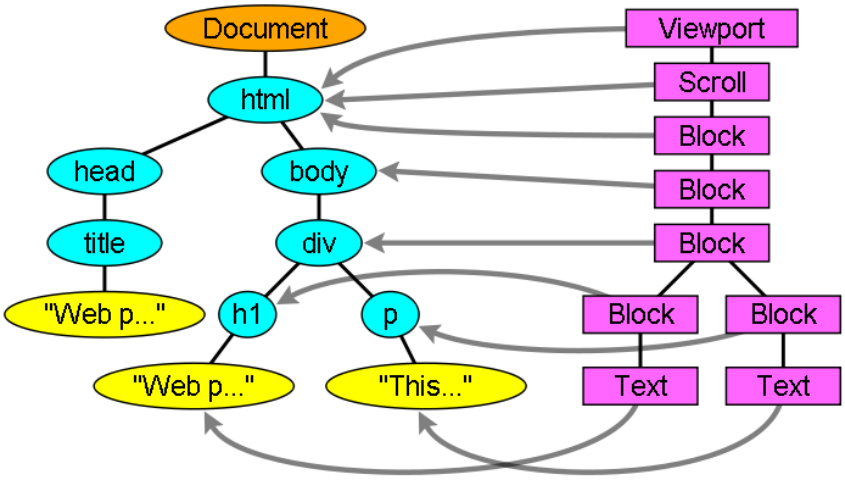
While the DOM tree is being constructed, the browser constructs another tree, the render tree. This tree is of visual elements in the order in which they will be displayed. It is the visual representation of the document. The purpose of this tree is to enable painting the contents in their correct order.

Firefox calls the elements in the render tree "frames". Webkit uses the term renderer or render object.

A renderer knows how to layout and paint itself and it's children.

The render tree relation to the DOM tree

The renderers correspond to the 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 attribute was assigned to "none" will not appear in the tree (elements with "hidden" visibility attribute will appear in 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.

7. 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 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 is 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.

7.1. Dirty bit system

In order not to do a full layout for every small change, browser 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". Children are dirty means that although the renderer itself may be ok, it has at least one child that needs a layout.

7.2. Global and incremental layout

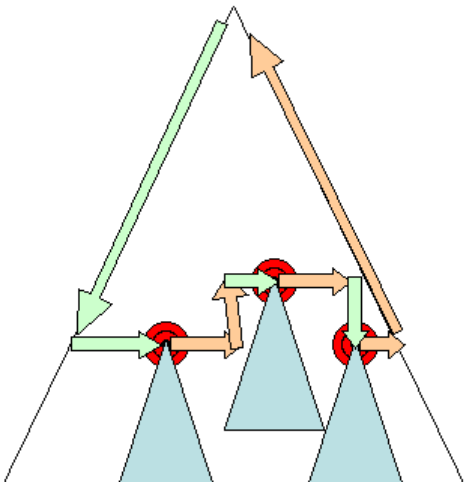
Layout can be triggered on the entire render tree - this is "global" layout. This can happen as a result of:

A global style change that affects all renderers, like a font size change.

As a result of a screen being resized

Layout can be incremental, only the dirty renderers will be layed 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.



7.3. 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 "offsightHeight" 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.

7.4. The layout process

The layout usually has the following pattern:

- Parent renderer determines its own width.

- Parent goes over children and:

- Place the child renderer (sets its x and y).

- Calls child layout if needed(they are dirty or we are in a global layout or some other reason) - this calculates the child's height.

- Parent uses children accumulative heights and the heights of the margins and paddings to set it own height - this will be used by the parent renderer's parent.

- Sets its dirty bit to false.

8. Painting

In the painting stage, the render tree is traversed and the renderers "paint" method is called to display their content on the screen. Painting uses the UI infrastructure component. More on that in the chapter about the UI.

8.1. 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 it's 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).

8.2. The painting order

CSS2 defines the order of the painting process - http://www.w3.org/TR/CSS21/zindex.html. 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:

- background color

- background image

- border

- children

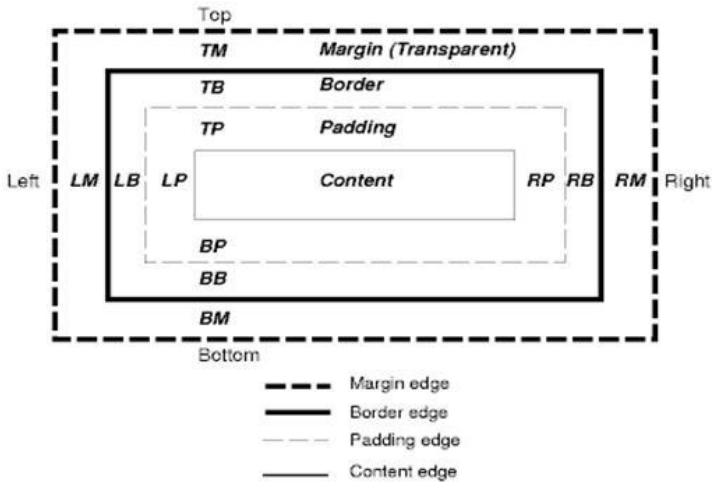
- outline

According to CCS2 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.

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.



All the content is summary from http://taligarsiel.com/Projects/howbrowserswork1.htm for learning purposes captured by Sintija Birgele