Profesor: Borja Rey Seoane

TEMA 1

INTRODUCIÓN Á PROGRAMACIÓN EN CONTORNA CLIENTE

Módulo: **Desenvolvemento Web en Contorna Cliente**

Ciclo: **DAW**

Linguaxes da web

Para crear unha páxina web debemos servirnos dunha serie de linguaxes que nos permiten deseñar e programar as mesmas. A continuación expoñemos algunhas das linguaxes máis empregadas diferenciando aquelas que actúan do lado do cliente e as que o fan do lado do servidor.

Linguaxes do lado do cliente

As linguaxes do lado do cliente son aquelas que adoitan ser interpretadas integramente polo navegador web, isto é, na propia máquina cliente. Son completamente independentes do servidor, o que permite que a páxina poida ser aloxada en distintas ubicacións, en tanto que unha vez descargada o procesamento será local. O único que fai falla, polo tanto, é que o navegador dispoña dos plugins axeitados, isto é, do conxunto de ferramentas e librarías que lle permitan comprender o código co que está a tratar. O devandito código é visible polo cliente, o que pode afectar á súa seguridade.

Entre estas linguaxes destacamos:

- **HTML[1]:** é unha linguaxe de marcas que permite crear contido estático nos documentos web, definindo a súa estrutura. Esta será a linguaxe que trataremos neste tema.
- **CSS[2]:** é unha linguaxe empregada para definir a presentación dun documento web.
- **JavaScript[3]:** que trataremos en detalle máis adiante.

[1] *Hypertext Markup Language.*

[2] *Cascading Style Sheets.*

[3] En ocasións abreviado coma JS.

Linguaxes do lado do servidor

Son aquelas linguaxes que son recoñecidas, executadas e interpretadas polo propio servidor e envíanse ao cliente nun formato comprensible polo usuario, é dicir, xa completamente procesadas. Son independentes do navegador utilizado. Ademais do anterior, o código pode ocultarse ao cliente, que só verá o código HTML finalizado. Algunhas das máis populares son:

- **PHP[4]:** é unha linguaxe de *scripting* multiplataforma creada especialmente para a xeración de páxinas web dinámicas, dado que que pode ser incrustada dentro do código HTML.
- **JSP[5]:** é unha tecnoloxía Java para Web orientada ao desenvolvemento de páxinas web dinámicas. En JSP o código Java intégrase directamente no código HTML, sendo interpretado e precompilado polo servidor para incluír contido de xeito dinámico. Esta tecnoloxía, hoxe en día en retroceso, apoiábase nos *servlets*, aplicacións Java completas que intercambiaban información co navegador web do cliente e se executaban no lado do servidor para complementar a funcionalidade das páxinas JSP
- **JSF[6]:** é unha tecnoloxía e *framework* que simplifica o desenvolvemento de interfaces de usuario e aplicacións Java vía web. Emprega JSP coma tecnoloxía de base para o despregamento das páxinas, pero tamén soporta outras, tales coma XUL[7].
- **ASP.NET.:** é a tecnoloxía sucesora de ASP[8], desenvolvida e comercializada por Microsoft. Baséase na emprega dun conxunto de clases .NET co obxecto de crear páxinas web dinámicas.
- **Python:** é unha linguaxe multiplataforma e multiparadigma en tanto que permite o desenvolvemento de estilos diferentes de programación. É unha linguaxe simple, versátil e rápida de desenvolver. Incide nunha sintaxe que favoreza un código lexible, se ben a súa integración na web non é fonda coma noutras linguaxes. É moi habitual atopar o seu usado aplicado ao procesamento matemático de grandes volumes de datos.

---

[4] Orixinalmente *Personal Home Page*, hoxe en día emprégase a sigla recurrente *PHP: Hypertext Preprocessor*.

[5] *JavaServer Pages*.

[6] *JavaServer Faces.*

[7] *XML-based User-interface Language*.

[8] *Active Server Pages.*

- **Perl:** é unha linguaxe de programación moi practica para extraer información de ar-quivos de texto e xerar informes a partires do contido dos ficheiros.

JavaScript

JavaScript (ou **JS**) é unha linguaxe de programación orientada a obxectos parcial (orixinalmente non soportaba nin herdanza nin polimorfismo) que permite crear scripts do lado do cliente para, por exemplo, controlar o navegador ou alterar o contido que se amosa nos documentos web xerando contido dinámico. JS permite, polo tanto, realizar accións no ámbito dunha páxina web, sendo soportado por todos os navegadores de uso habitual.

Mediante o uso de Js podemos crear animacións e procedementos que dependan da interacción co usuario, é dicir, podemos crear páxinas dinámicas. Dado que tratamos dunha linguaxe do lado do cliente, será o navegador o encargado de interpretar as instrucións do mesmo así coma de ilas executando.

No referente ás súas capacidades relativas á orientación a obxectos, JS soporta o Modelo de Datos de Documento (**DOM**[9]), que representa un conxunto de obxectos predefinidos que permiten acceder aos distintos elementos dunha páxina web así coma a certas características específicas do navegador.

Manipulación de obxectos

A manipulación de obxectos en JS segue os mesmos principios que na maioría de linguaxes de programación pertencentes a este paradigma. Os obxectos deséñanse para un contexto funcional concreto, o que non é incompatible con que poidan ser empregados para desenvolver diversas funcións.

[9] *Document Object Model.*

Navegadores web

Afondaremos neste punto na estrutura interna e funcionamento dos **navegadores web**, en tanto serán os encargados de interpretar e executar o código no lado do cliente, coma xa introducimos no punto anterior.

O que exporemos a continuación será válido, en liñas xerais, para os cinco navegadores máis habituais no mercado actual: Google Chrome, Mozilla Firefox, Microsoft Edge, Safari e Opera.

Función

A función principal dun navegador é solicitar a un servidor web os recursos que o usuario determinara (fornecendo unha URI[10]) para amosarlle o resultado de procesalos. Eses recursos van ser, xeralmente (aínda que non obrigadamente), documentos HTML.

O xeito en que o navegador interpreta e amosa os documentos HTML ven determinado polas especificacións das linguaxes HTML e CSS, establecidas polo W3C[11].

Compoñentes

Vemos a continuación as que podemos considerar *grosso modo* como compoñentes principais comúns á maioría de navegadores típicos:

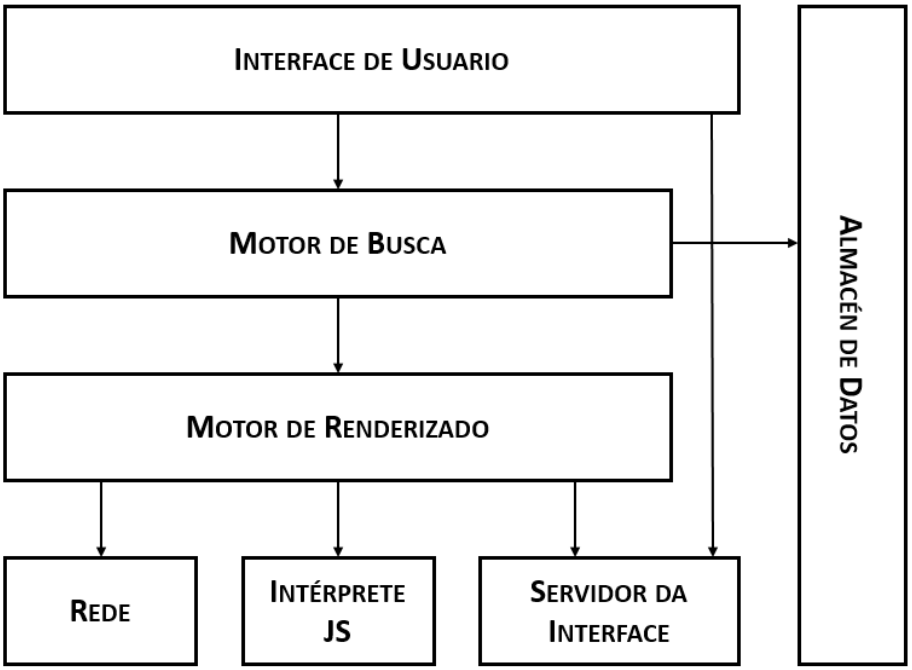
- **Interface de usuario:** que inclúe unha serie de «partes» que adoitan compartir todos os navegadores, como son:
    - Panel principal: no que se amosará o contido do documento actual.
    - Unha barra de enderezos: na que se especifica a URI/URL do documento actual.
    - Botóns de avance e retroceso: para navegar polas distintas páxinas dun mesmo sitio e do historial recente en orde cronolóxica.
    - Marcadores/favoritos: para gardar as URI/URL de documentos de interese.
    - Botón de parado: que detén a carga do documento actual.
    - Botón de refresco: que volve solicitar a carga do documento actual.

[10] *Uniform Resource Identifier.*

[11] *World Wide Web Consortium.*

    - Botón de inicio: que volve ao documento determinado como de inicio na configuración do navegador.
- **Motor de renderizado[12]:** que é responsable de traducir e amosar o contido solicitado.
- **Motor de busca/procura:** que coordina as accións entre a interface e o motor de renderización.
- **Motor/capa de rede:** que é responsable de soster as comunicacións de rede. Por exemplo: atende solicitudes HTTP, almacena a información da sesión actual, etc.
- **Servidor da interface:** que permite presentar *widgets* básicos tales coma xanelas e diálogos empregando mecanismos propios do sistema operativo en segundo plano.
- **Intérprete de JavaScript:** que analiza e executa o código JS instrución a instrución.
- **Almacén de datos/capa de persistenza:** garda a base de datos web necesaria para a operativa do navegador.

Na seguinte imaxe podemos apreciar a relación funcional entre as distintas compoñentes mencionadas:



[12] Certos navegadores, coma Google Chrome, posúen múltiples instancias do motor de renderizado funcionando a un tempo (unha por cada lapela), de xeito que cada documento en carga corresponde a un proceso/fío independente.

O motor de renderizado

Coma xa comentamos no punto anterior, o obxectivo primordial do motor de renderizado é presentar, a través da interface de usuario, o resultado de procesar o documento solicitado.

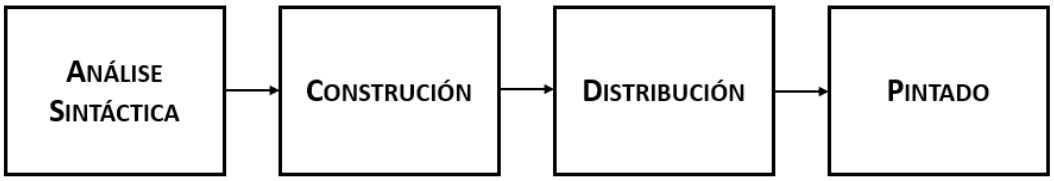
A pesares de que os navegadores modernos son quen de ler documentos de tipos non recoñecidos coma estándar pola W3C (coma os documentos PDF), centrarémonos aquí nos documentos HTML e os documentos de imaxe, en tanto estándares web recoñecidos.

Os motores de renderizado máis populares son **WebKit** (empregado por Google Chrome e Safari), **Gecko** (utilizado por Mozilla Firefox), **EdgeHTML** (usado por Microsoft Edge) e **Presto** (no navegador Opera).

Fluxo principal

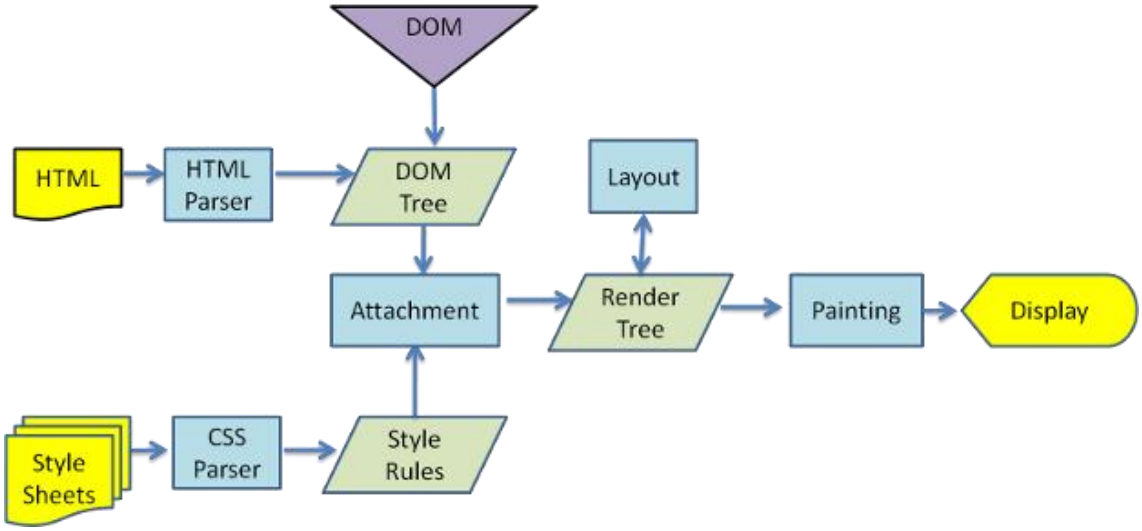
O proceso de funcionamento dun motor de renderizado (que se coñece coma **fluxo principal** do navegador) comeza coa recepción do documento solicitado dende a capa de rede (xeralmente en fragmentos de 8 KB de tamaño). Unha vez recibido o documento tómanse os seguintes pasos:

- **Análise sintáctica do documento HTML:** que ten coma obxecto construír a árbore de contidos ou árbore DOM. Con ese fin analízanse os datos de estilo (CSS) xunta coas etiquetas HTML asociadas.
- **Construción da árbore de renderizado:** partindo a información da fase anterior. Esta árbore conterá uns obxectos representados mediante rectángulos que terán cadanseus atributos visuais asociados, e que figurarán ordeados na árbore segundo a orde na que aparezan en pantalla.
- **Deseño da árbore:** asígnanse, a cada elemento da árbore de renderizado, as coordenadas do lugar no que teñen que aparecer na pantalla.
- **Pintado da árbore:** percórrese a árbore de renderizado e «píntase» cada un dos nodos empregando información fornecida polo servidor da interface de usuario.

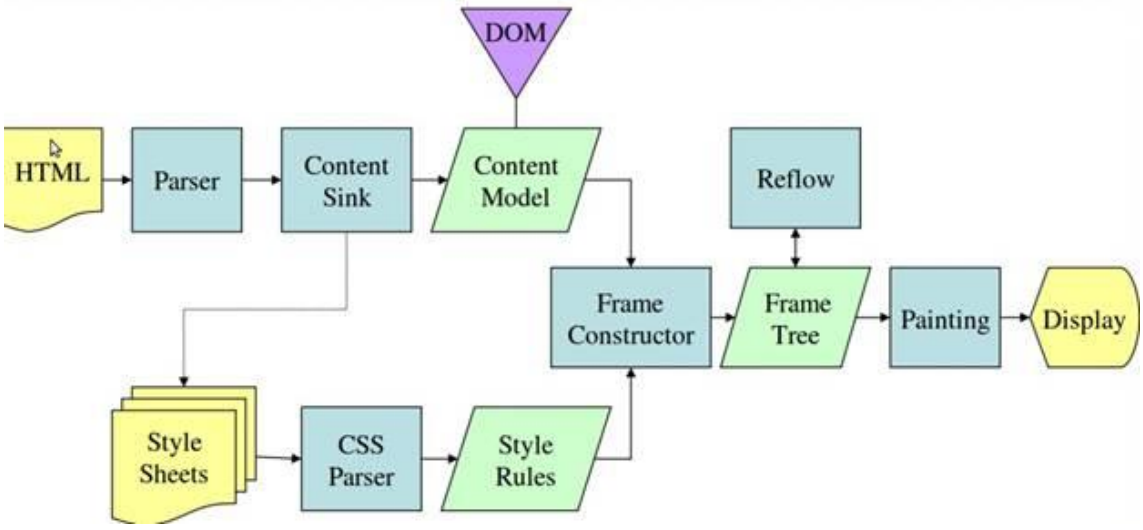


O proceso anterior é, ademais, gradual e non se espera até o seu completado para amosar os resultados ao usuario, senón que o habitual é presentar resultados parciais paso a paso até que a carga da páxina web estea finalizada.

A continuación podemos ver algúns exemplos de fluxos principais, como por exemplo o de WebKit:



Ou tamén o fluxo principal de Gecko:



Como podemos apreciar, os fluxos principais de WebKit e Gecko son practicamente análogos, con pequenas diferenzas de nomenclatura e apenas distinción entre os pasos seguidos. Unha diferenza que paga a pena salientar é a existenza dun «repositorio de

contidos» en Gecko que serve para a creación de obxectos DOM e que actúa de capa extra entre a análise do código HTML e a árbore DOM, entre as cales hai comunicación directa no caso de WebKit.

A análise

A análise sintáctica e gramatical de documentos ou código fonte (o que se denomina ás veces mediante o anglicismo *parseo*) consiste na tradución do contido a unha estrutura que teña sentido para o *software* encargado de procesar a información. Habitualmente, o resultado desa tradución será unha árbore de nodos á cal denominamos **árbore de análise** ou **árbore de sintaxe**.

Este tipo de análise é posible grazas a que os formatos informáticos empregan **gramáticas libres de contexto**, isto é, unha gramática determinista formada por un vocabulario e unhas certas regras de sintaxe (fronte ás linguas humanas, que non operan dese xeito).

O proceso de sintaxe consta de dúas fases ou subprocesos:

- **A análise léxica**, a cal consiste en descompoñer os datos de entrrada en *tokens*, elementos mínimos de vocabulario da linguaxe que actúan a xeito de bloques de construción.
- **A análise sintáctica**, que consiste na aplicación das regras sintácticas, é dicir, comproba que os *tokens* están correctamente posicionados e relacionados entre si para construír sentenzas válidas.

Así mesmo, a análise é un proceso iterativo no que, para cada novo *token* atopado, procúranse coincidencias entre o mesmo e as regras de sintaxe existentes. Chegados a ese punto existen dous posibles resultados:

- No caso de darse unha coincidencia, engádese á árbore de análise un nodo correspondente ao *token* actual, pasando o analizador a solicitar o seguinte.
- Se non hai coincidencia coas regras sintácticas, o analizador almacena o *token* internamente e continúa a solicitar novos *tokens* até atopar unha regra que coincida con todos os almacenados. No caso de que non haxa ningunha, lánzada unha excepción que indica que o documento non é válido en tanto contén erros sintácticos.

Analizador de HTML

Partindo do anterior, podemos deducir que o **analizador de HTML** ten coma labor tomar as marcas dun documento HTML para construír a árbore de análise correspondente en base á sintaxe e o vocabulario definidos nas especificacións oficiais da W3C. Esas especificacións realízanse mediante DTD (documentos de definición de tipos) e, dependendo da versión de HTML coa que esteamos a traballar, poida que non nos esteamos a enfrontar a unha gramática libre de contexto. Isto débese a que certas versións de HTML son máis «permisivas» no uso de etiquetas, permitindo asumir o engado implícito dalgunhas delas xa sexa omitindo o peche das mesmas ou admitindo variantes no texto da marca.

DTD de HTML

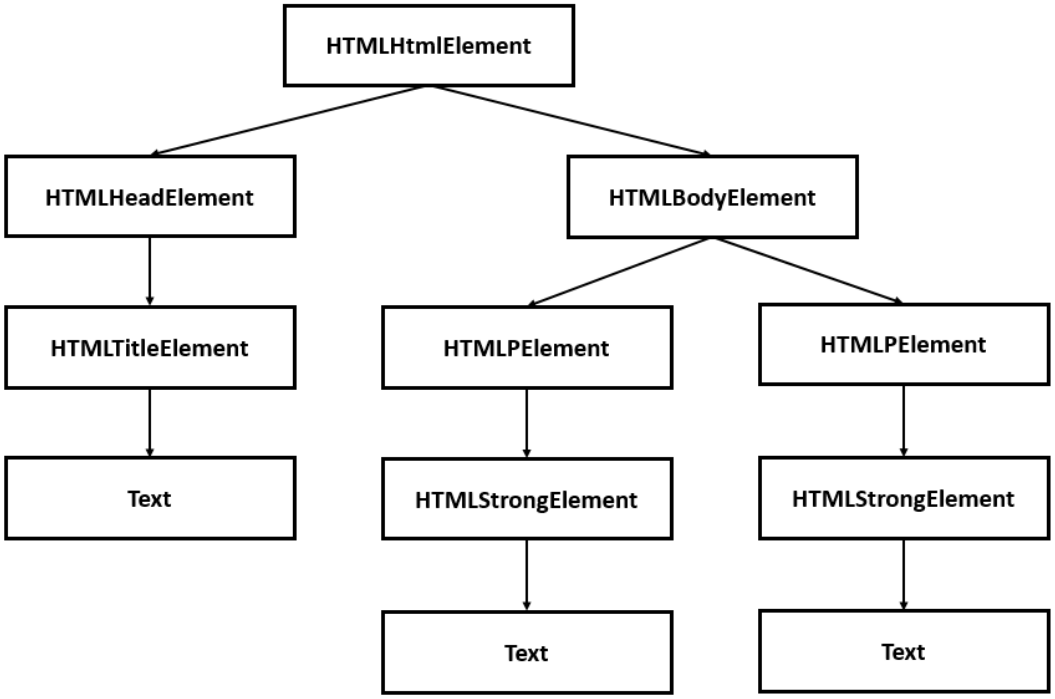
A definición de HTML, coma xa comentamos, realízase mediante os DTD liberados pola W3C. DTD permite definir regras para linguaxes da familia SGML, e os seus documentos conteñen definicións de todos os elementos permitidos, os seus atributos e a súa xerarquía.

Existen algunhas variacións da definición DTD (como o modo estricto) que forzan unha gramática libre de contexto, se ben a maioría de navegadores aínda soportan as versións máis vellas co gaio de permitir o acceso a sitios web obsoletos.

Árbore DOM

A árbore de saída dunha análise HTML está formada por elementos DOM e nodos de atributo e consiste nunha representación dos obxectos do documento HTML e a interface dos elementos HTML para o mundo exterior, como é JS.

Velaquí un exemplo de árbore DOM:



E o código HTML que correspondería á mesma:

```
<html>
  <head>
    <title>Exemplo de árbore DOM</title>
  </head>
  <body>
    <p>Este texto está <strong>remarcado</strong>.</p>
    <p>Estoutro texto está <del>riscado</del>.</p>
  </body>
</html>
```

Para coñecer mellor os distintos obxectos DOM recoñecidos no estándar é recomendable consultar a **Especificación Oficial de DOM da W3C** así coma o seu **Estándar Asociado**.

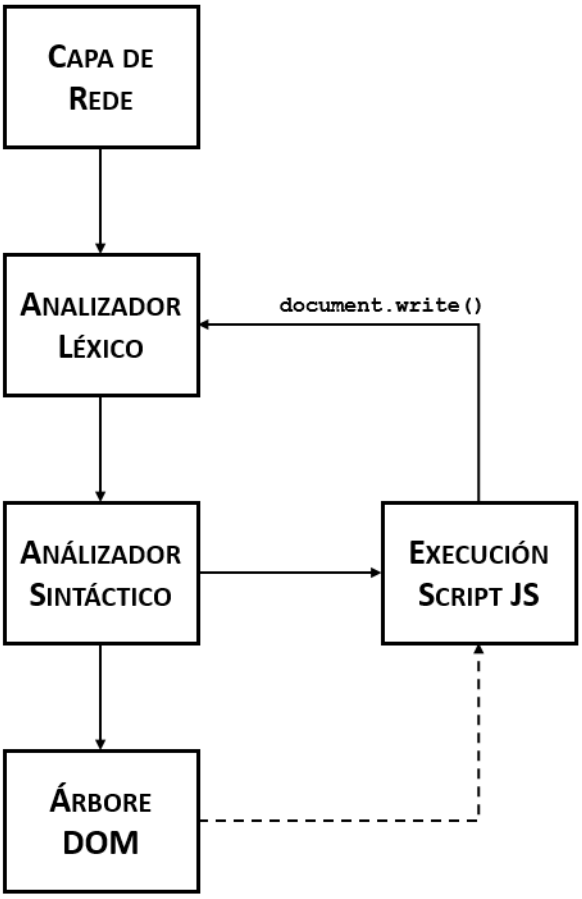
Particularidades

Coma xa vimos en puntos anteriores, a linguaxe HTML non pode ser analizada empregando os analizadores típicos xa que non sempre unha páxina HTML vai seguir unha gramática libre de contexto. O anterior débese a tres motivos principais:

- A natureza «permisiva» da linguaxe, xa mencionada.
- A tolerancia a erros gramaticais dos propios navegadores, en aras da retrocompatibilidade con versións obsoletas de HTML.

- A natureza iterativa do proceso de análise xa que, se ben idealmente os documentos HTML deberían ser estáticos, poden aparecer comandos que engadan ou eliminen *tokens* (como `document.write` en JS), o que modificaría os datos de entrada.

É por esta causa que os navegadores posúen analizadores *ad hoc*, construídos expresamente tendo en conta estas particularidades de HTML. Estes analizadores teñen a capacidade de realimentar a primeira fase da análise (a análise léxica, realizada polo «tokenizador»), tal e coma se aprecia no seguinte diagrama:



Tokenización

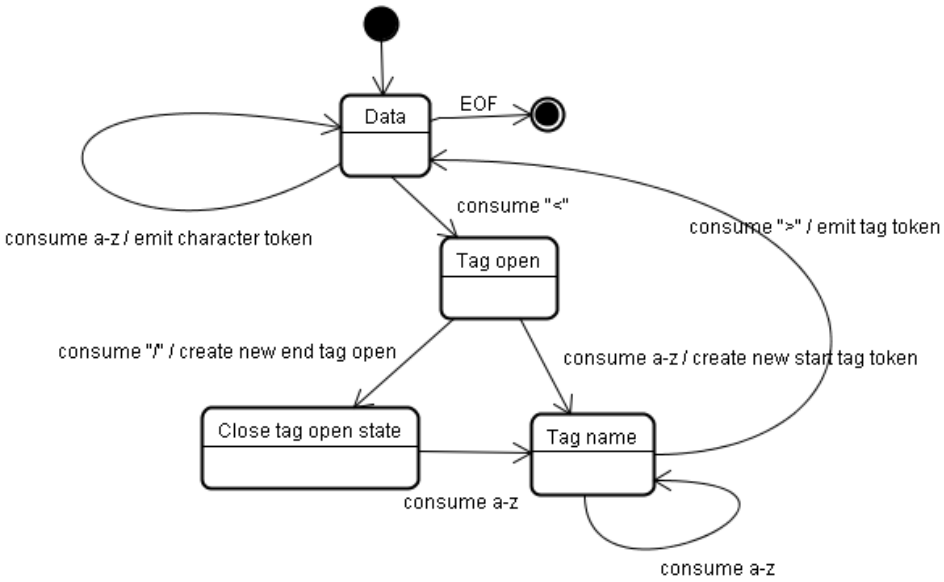
Podemos dicir que o tokenizador vai funcionar coma unha máquina de estados na que en cada paso da análise (en cada estado) producirá un *token* HTML consumindo un ou varios carácteres do fluxo de entrada e actualizando o seguinte estado dacordo cos mesmos. A decisión dependerá do estado de tokenización actual e do estado de construción da árbore. O anterior implica que un mesmo carácter consumido poderá producir resultados diferentes dependendo do estado actual do proceso.

Para entender mellor o anterior, veremos como funciona *grosso modo* a tokenización cun exemplo práctico. Partiremos do seguinte fragmento de código HTML:

```
<html>
  <body>
    Ola mundo!
  </body>
</html>
```

- O estado inicial do proceso coñécese como «estado de datos», a partires do cal detéctase o primeiro carácter «<», que indica o inicio dunha etiqueta. Nese caso, o estado pasa a ser «estado de etiqueta aberta». O tokenizador continúa recibindo os carácteres alfabéticos que forman a etiqueta, creándose o «token de etiqueta inicial». Namentres aparezan os carácteres alfabéticos da etiqueta estararase no estado de «nome de etiqueta», o cal durará até que se consuma o carácter «>». Todos os carácteres consumidos engadiranse ao nome do novo *token*, neste caso **`html`**.
- Chegados ao peche da etiqueta anterior, emítese o *token*, múdase o estado a «estado de datos» e repítense os pasos para a etiqueta **`body`**.
- Ao comezar o seguinte carácter, sendo este a «O» de «Ola mundo!», emítese un *token* de carácter e continúase así até chegar ao seguinte «<». Cada carácter terá, polo tanto, un *token* de seu.
- Voltamos ao «estado de etiqueta aberta» e consúmese o seguinte carácter, neste caso «/», creando un *token* de «etiqueta final» e pasando novamente ao estado de «nome de etiqueta». Unha vez consumido o carácter «>» emitirase o *token* de nova etiqueta e voltarase ao «estado de datos».
- Finalmente, a etiqueta **`</html>`** tratarase analogamente ao caso anterior.

A continuación apréciase un diagrama de estados que representa, en liñas xerais, o procedemento de tokenización anteriormente exposto:



Construción da árbore DOM

Cando comeza a análise dun documento HTML, e unha vez rematada a tokenización, créase o obxecto **`Document`** que actuará coma raíz da árbore DOM e ao cal engadiranse os distintos *tokens* resultantes da fase anterior. O construtor da árbore procesará cada nodo emitido polo tokenizador, especificando para cada un deles que elementos DOM son relevantes e deben ser creados en correspondencia ao *token* en cuestión. Ademais de engadirse á árbore DOM, o elemento engadirase a unha pila de elementos abertos, a cal empregarase para corrixir incidencias de aniñamento e para completar etiquetas non pechadas. O algoritmo seguido para esta segunda fase defínese tamén coma unha máquina de estados na que os mesmos reciben o nome de «modos de inserción».

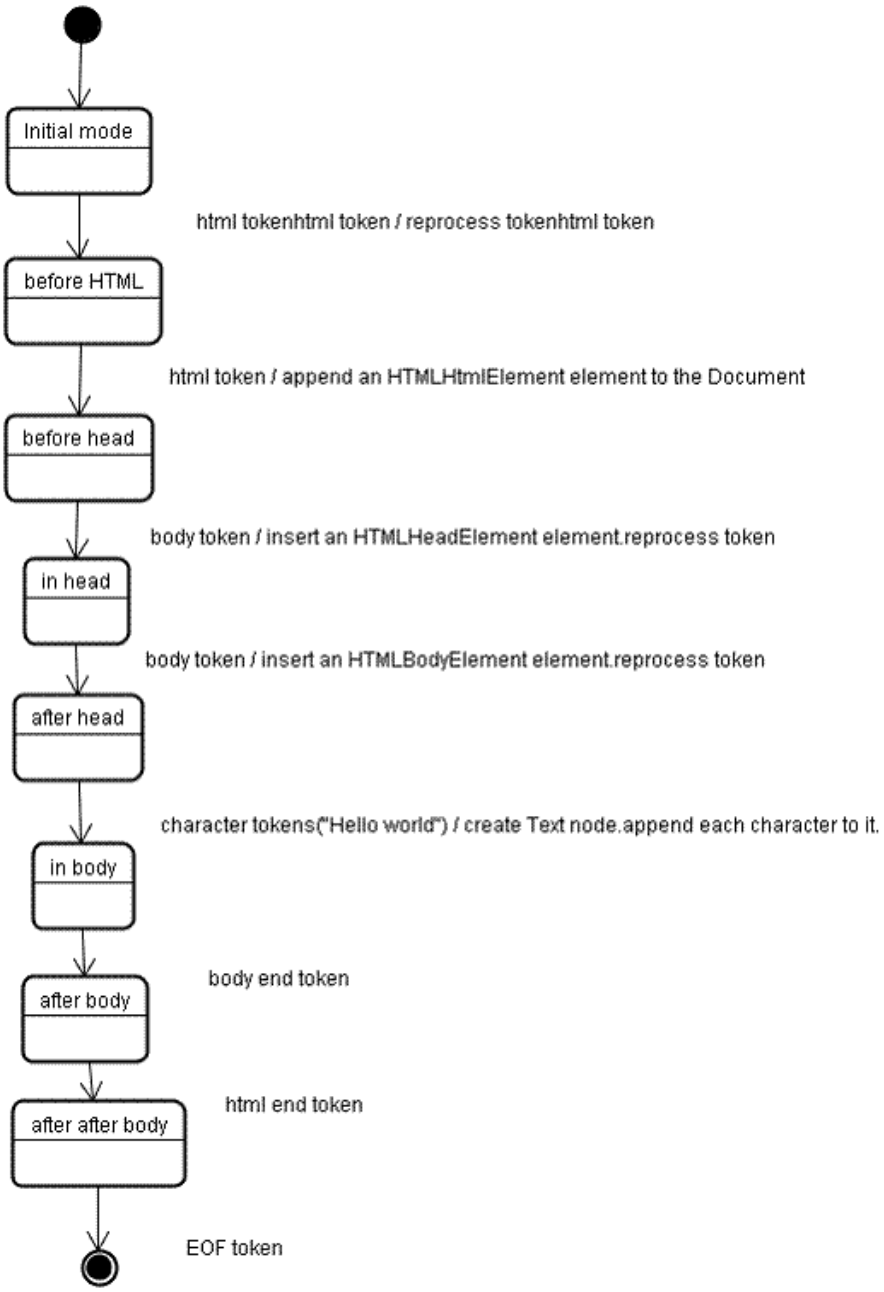
Se retomamos o exemplo do fragmento HTML do punto anterior podemos definir paso a paso o proceso de construción da árbore DOM que lle corresponde como sigue:

- O primeiro modo a procesar será o «modo inicial», representado polo token **`html`**.
- Recibido o *token* anterior pasarase ao modo «previo a html», volvendo procesar o mesmo, resultando na creación e engadido á árbore (coma fillo de **`Document`**) do elemento **`HTMLHtmlElement`**.
- O estado mudará a «antes da cabeceira». Recibiremos o *token* **`body`** e crearase de xeito implícito un elemento **`HTMLHeadElement`** (aínda que non haxa ningún

token **`head`**) o cal será engadido á árbore coma fillo do elemento **`HTMLHtmlElement`**.

- Pásase ao modo «na cabeceira» e, acto seguido, a «despois da cabeceira» (en tanto o token **`head`** foi engadido implicitamente). O *token* **`body`** é novamente procesado, creándose e inseríndose na árbore un elemento **`HTMLBodyElement`**. Pásase ao modo «no corpo».
- Posteriormente recíbese cada un dos *tokens* de carácter correspondentes á cadea «Ola Mundo!», sendo o primeiro deles o que desencadee a creación do nodo **`Text`** e a súa inserción na árbore.
- A recepción do *token* de peche do **body** producirá o cambio cara o modo de «despois do corpo».
- Finalmente recibirase o *token* de peche de html, pasándose a «despois de despois do corpo», finalizando a análise e a creación da árbore.

A continuación podemos ver un diagrama de estado ilustrativo do procedemento anterior:



Fin da análise

Unha vez rematadas as fases de tokenización e de creación da árbore DOM, o navegador marca o documento coma interactivo e procede a analizar aquelas secuencias de comandos que figuraran en modo «aprazado», isto é, aquelas que haxa que executar unha vez teñan rematado as fases iniciais da análise.

Seguidamente establecerase o estado do documento coma «completado» e lanzarase un evento de carga para amosar o resultado ao usuario.

Manipulación e tolerancia a erros

Podemos afirmar sobre os navegadores web, no relativo ás súas capacidades coma procesadores de documentos HTML, que son programas cunha alta tolerancia a erros, na que non existen verdadeiros erros por sintaxe inválida, senón que sempre vai facerse por corrixir os contidos inválidos para tentar amosar un documento completo e estable ao usuario.

A devandita xestión de erros non constitúe unha parte do estándar de procesamento HTML publicado pola W3C, senón que é resultado da experiencia acugulada ao longo de anos, especialmente daquelas épocas nas que convivían múltiples estándares HTML simultaneamente (algúns de *iure* e outros *de facto*), polo que os desenvolvedores de navegadores tiveron que se adaptar ao mercado existente.

Coma exemplo, vexamos algúns exemplos de tolerancia a erros presentados polo motor WebKit:

- **Etiqueta de salto de liña:** uso de </br> no canto de
.
- **Táboa perdida:** unha táboa incluída dentro doutra táboa, pero non dentro dunha cela será situada a continuación da táboa contedora no documento final.
- **Formulario aniñado:** se incluímos un formulario HTML dentro doutro o segundo será ignorado.
- **Xerarquía demasiado fonda:** se aniñamos máis de 20 etiquetas do mesmo tipo serán ignoradas todas.
- **Incorrecto peche do documento:** se faltan as etiquetas de peche de `body` e `html`, ou están incorrectamente colocadas, serán ignoradas e engadiranse outras implicitamente.

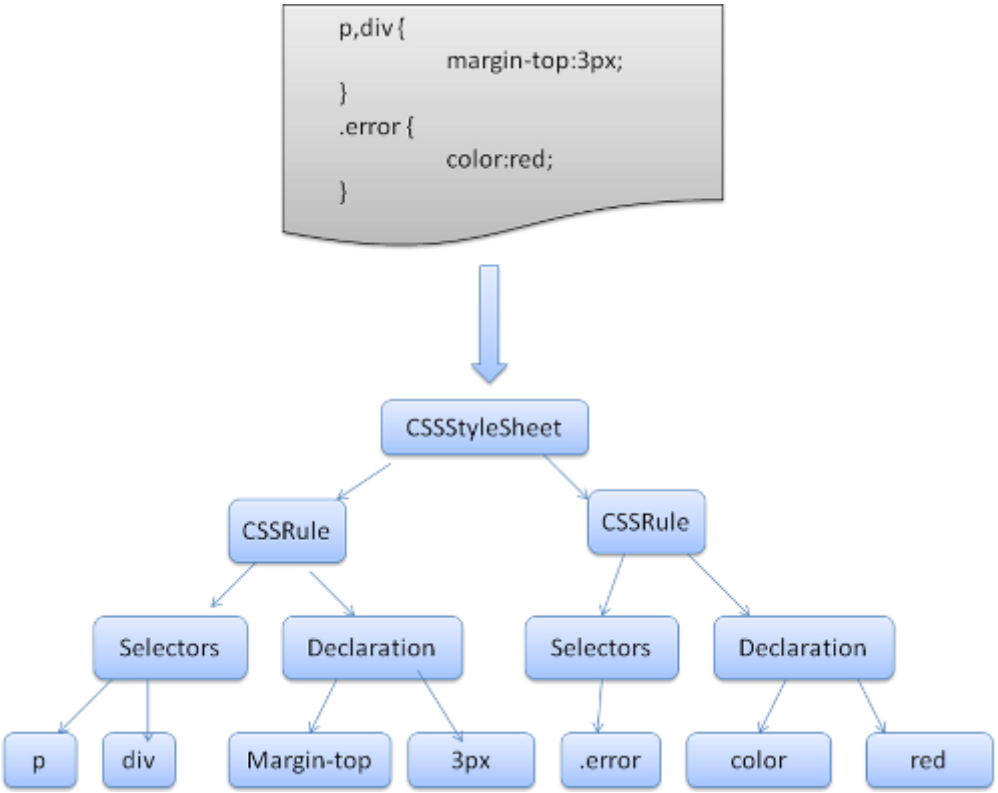
Analizador de CSS

Fronte ao que ocorría con HTML, e que implicaba as particularidades da súa análise vistas nos puntos anteriores, CSS ten unha gramática libre de contexto. Do anterior dedúcese que o código CSS pode ser procesado por analizadores tradicionais, e que calquera erro sintáctico no mesmo implicará a invalidez do conxunto.

Certos navegadores, coma Chrome, empregan analizadores ascendentes. Un analizador ascendente procesa a árbore do documento dende as follas cara á raíz. Outros

navegadores, coma Firefox, empregan analizadores descendentes, que proceden no xeito inverso.

No seguinte diagrama podemos ver a correspondencia entre un fragmento de código CSS de exemplo e a súa árbore de documento:

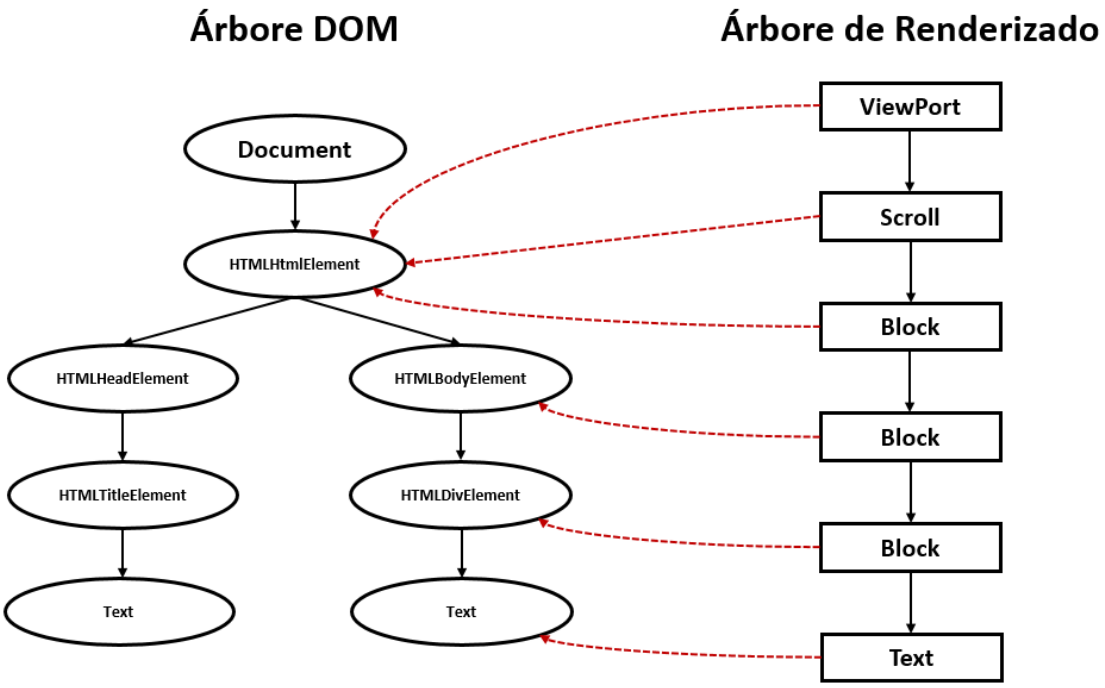


Coma xa sabemos, a árbore DOM determina os nodos que compoñen o documento HTML sobre o que se vai traballar, namentres que a árbore de renderizado determinará como se vai presentar o documento ao usuario. A análise de CSS ten que integrar, polo tanto, ambas compoñentes para unha correcta aplicación dos estilos definidos.

Se ben existe certa correspondencia estrutural entre a árbore DOM e a árbore de renderizado, esta non se produce un a un, xa que os elementos non visuais de DOM non aparecen na árbore de renderizado (como, por exemplo, o elemento `head`). Do mesmo xeito, certos elementos DOM corresponden con múltiples obxectos visuais, dado que posúen unha estructura complexa que non pode ser descrita unicamente mediante un rectángulo (por exemplo, os elementos `select` posúen tres renderizadores –un para a área de visualización, outro para a lista despregable e outro para o botón-).

Na imaxe que figura a continuación vese unha correspondencia simplificada (os elementos aboiantes e aqueles con posición absoluta fican fora do fluxo) entre unha

árbore DOM e a súa árbore de renderizado, na cal represéntanse certas características CSS que corresponderían aos elementos asociados (mediante as frechas vermellas):



Como podemos deducir da imaxe anterior, a árbore de renderizado constrúese a partires da etiqueta `html`, actuando coma nodo raíz o bloque superior que contén ao resto de bloques (cuxas dimensións coinciden coas da área de visualización da xanela do navegador –`ViewPortFrame` en Firefox e `RenderView` en navegadores WebKit-).

Nos navegadores WebKit o establecemento da relación entre un nodo DOM e os seus correspondentes na árbore de renderizado realízase mediante un proceso de **asociación** asíncrono no que o mero feito de inserir un nodo en DOM desencadea as asociacións correspondentes no renderizado.

Computación dos estilos

Para crear a árbore de renderización faise preciso calcular as propiedades visuais de cada un dos obxectos de renderización, isto é, procesar as propiedades de estilo correspondentes a cada elemento. As devanditas propiedades poden proceder de múltiples follas de estilo distintas, de estilos CSS integrados no propio documento HTML así coma de atributos visuais de HTML (coma, por exemplo, `bgcolor`).

A computación de estilos leva asociadas unha serie de dificuldades:

- Se as propiedades aplicadas a un elemento son moitas pode producirse un uso excesivo de memoria.

- É preciso optimizar o código CSS para evitar que a procura de regras coincidentes para un mesmo elemento sexa moi redundante.
- A xerarquía de regras (os selectores) pode implicar a aplicación de regras en cascada, coa complexidade de procesamento asociada.

En navegadores WebKit, os nodos da árbore de renderizado reciben o nome de obxectos de estilo (`RenderStyle`), e poden ser compartidos polos nodos DOM en certas circunstancias, se ben hai unha serie de restricións aplicables que non citaremos aquí para non entrar en demasiado detalle sobre o procedemento.

Orde de procesamento

A continuación veremos en que orde e como realizan os navegadores o procesamento dos estilos CSS e os *scripts* JS.

Así, a análise dun documento HTML detense cando detecta o inicio dunha etiqueta `script`, e non continúa até que o código JS teña rematado de executarse. Nas versións máis recentes de HTML existe a posibilidade de pospoñer a análise e execución do *script*, de xeito que fique «aprazada» até que a análise de HTML remate.

Os navegadores con motores WebKit e Gecko realizan unha **análise especulativa** nun fío separado, de xeito que se analizan os recursos externos (*scripts*, follas de estilo, imáxenes, vídeos, etc.) sen necesidade de interromper a análise principal. Este tipo de análise (a especulativa) non altera a árbore DOM.

Noutra banda, e a pesares de que o contido das follas de estilo non pode alterar a árbore DOM dun documento, si poden os estilos CSS ser modificados dende JS, polo que os navegadores bloquean a execución dos *scripts* namentres haxa follas de estilo pendentes de análise.

Para computar estilos CSS, Firefox emprega unha **árbore de regras** na que se van engadindo coma nodos da mesma os estilos a aplicar aos distintos nodos da árbore DOM; e unha **árbore de contextos** no que se establecen correspondencias entre elementos do documento HTML e as regras que lles son de aplicación. Deste xeito, cada nodo representará as regras a aplicar, tendo os niveis inferiores maior prioridade do que os superiores.

Vexámolo cun exemplo, partindo do seguinte código HTML:

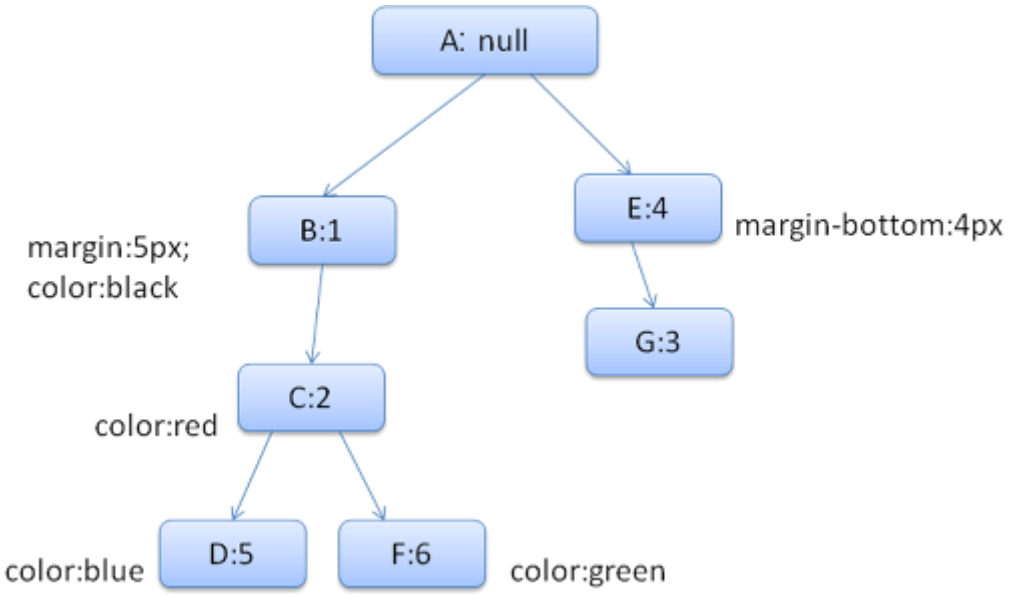
```
<html>
  <body>
    <div class="err" id="div1">
      <p>Isto é un <span class="big">erro grande </span>
        E estoutro é un
```

```
        <span class="big">erro moi grande</span>
      </p>
    </div>
    <div class="err" id="div2">Outro erro</div>
  </body>
</html>
```

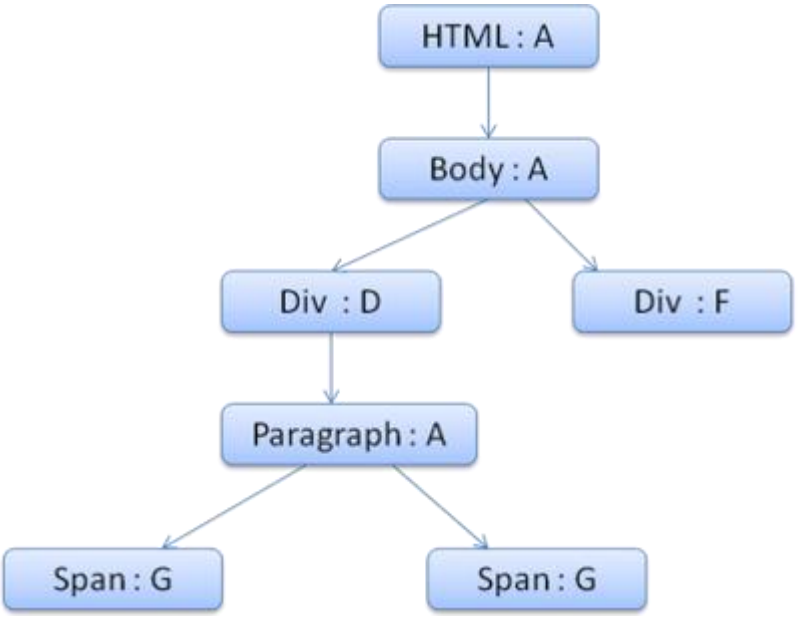
De xeito que aos estilos aniñados no código anterior correspóndenlle-las seguintes regras:

```
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}
```

Teremos polo tanto unha árbore de regras na que o nodo raíz representará a non aplicación de regras e, a partires dese punto, incluiremos as regras numeradas mencionadas:



A árbore de contextos do exemplo anterior terá a seguinte forma:



Supoñamos entón que xa temos a análise do documento HTML en marcha e nese proceso atópase a segunda etiqueta `div` do exemplo:

- Primeiro procúranse na árbore de regras aquelas regras que sexan de aplicación ao devandito `div`, neste caso as regras 1, 2 e 6.
- Engádese á árbore de regras o nodo F.
- Créase un novo contexto de estilo e engádese á árbore de contextos (Div: F) de xeito que apunte ao nodo F da árbore de regras.

Cando remate o procedemento cada nodo da árbore de contextos indicará as regras que lle son de aplicación. Así, para cada nodo da árbore de regras ascenderase até a raíz, aplicando todas as regras que aparezan no camiño, sempre respectando a prioridade dos niveis inferiores sobre os superiores. Así, para o segundo div aplicaranse as regras dos nodos A, B, C e F.

Orde en cascada

Ademais do visto no punto anterior, ao tempo de procesar os estilos CSS é preciso considerar a orde de aplicación que teran os mesmos dependendo de onde foran declarados. A isto coñéceselle coma **orde en cascada** e vai depender da orixe das follas de estilo. Así os estilos aplicaranse, de menor a maior prioridade como segue:

- Estilos con orixe *user-agent* (do navegador) e importancia *normal*.
- Estilos con orixe *user* (do propio usuario) e importancia *normal*.
- Estilos con orixe *author* (do desenvolvedor) e importancia *normal*.
- Estilos de animacións.

- Estilos con orixe *user-agent* (do navegador) e importancia *!important*.
- Estilos con orixe *user* (do propio usuario) e importancia *!important*.
- Estilos con orixe *author* (do desenvolvedor) e importancia *!important*.
- Estilos de transicións.

Bibliografía

- Garsiel, T., Irish, P. (2011). *Cómo funcionan los navegadores: lo que hay detrás de los navegadores web actuales*. [online] Dispoñible en: https://www.html5rocks.com/es/tutorials/internals/howbrowserswork [Accedido o 21 de setembro de 2021].
- MDN Web Docs. (2021). *Introducing the CSS Cascade*. [online] Dispoñible en: https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade [Accedido o 29 de setembro de 2021].

ÍNDICE