HTML

Living Standard — Last Updated 23 May 2014

← 12.2.4 Tokenization – Table of contents – 12.2.6 The end →

12.2.5 Tree construction

The input to the tree construction stage is a sequence of tokens from the tokenization stage. The tree construction stage is associated with a DOM `Document` object when a parser is created. The "output" of this stage consists of dynamically modifying or extending that document's DOM tree.

This specification does not define when an interactive user agent has to render the `Document` so that it is available to the user, or when it has to begin accepting user input.

As each token is emitted from the tokenizer, the user agent must follow the appropriate steps from the following list, known as the **tree construction dispatcher**:

- **If there is no adjusted current node**
- **If the adjusted current node is an element in the HTML namespace**
- **If the adjusted current node is a MathML text integration point and the token is a start tag whose tag name is neither "mglyph" nor "malignmark"**
- **If the adjusted current node is a MathML text integration point and the token is a character token**
- **If the adjusted current node is an `annotation-xml` element in the MathML namespace and the token is a**

↪ **If the adjusted current node is an HTML integration point and the token is a start tag**

↪ **If the adjusted current node is an HTML integration point and the token is a character token**

↪ **If the token is an end-of-file token**

Process the token according to the rules given in the section corresponding to the current insertion mode in HTML content.

↪ **Otherwise**

Process the token according to the rules given in the section for parsing tokens in foreign content.

The **next token** is the token that is about to be processed by the tree construction dispatcher (even if the token is subsequently just ignored).

A node is a **MathML text integration point** if it is one of the following elements:

- An `mi` element in the MathML namespace
- An `mo` element in the MathML namespace
- An `mn` element in the MathML namespace
- An `ms` element in the MathML namespace
- An `mtext` element in the MathML namespace

A node is an **HTML integration point** if it is one of the following elements:

- An `annotation-xml` element in the MathML namespace whose start tag token had an attribute with the name "encoding" whose value was an ASCII case-insensitive match for the string "`text/html`"
- An `annotation-xml` element in the MathML namespace whose start tag token had an attribute with the name "encoding" whose value was an ASCII case-insensitive match for the string "`application/xhtml+xml`"
- A `foreignObject` element in the SVG namespace
- A `desc` element in the SVG namespace
- A `title` element in the SVG namespace

Note *Not all of the tag names mentioned below are conformant tag names in this specification; many are included to handle legacy content. They still form part of the algorithm that implementations are required to implement to claim conformance.*

Note *The algorithm described below places no limit on the depth of the DOM tree generated, or on the length of tag names, attribute names, attribute values, `Text` nodes, etc. While implementors are encouraged to avoid arbitrary limits, it is recognised that practical concerns will likely force user agents to impose nesting depth constraints.*

12.2.5.1 Creating and inserting nodes

While the parser is processing a token, it can enable or disable **foster parenting**. This affects the following algorithm.

The **appropriate place for inserting a node**, optionally using a particular *override target*, is the position in an element returned by running the following steps:

1. If there was an *override target* specified, then let *target* be the *override target*.

   Otherwise, let *target* be the current node.

2. Determine the *adjusted insertion location* using the first matching steps from the following list:

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

   Note *Foster parenting happens when content is misnested in tables.*

   Run these substeps:

   1. Let *last template* be the last `template` element in the stack of open elements, if any.

   2. Let *last table* be the last `table` element in the stack of open elements, if any.

   3. If there is a *last template* and either there is no *last table*, or there is one, but *last template* is lower (more recently added) than *last table* in the stack of open elements, then: let *adjusted insertion location* be inside *last template*'s template contents, after its last child (if any), and

abort these substeps.

4. If there is no *last table*, then let *adjusted insertion location* be inside the first element in the stack of open elements (the `html` element), after its last child (if any), and abort these substeps. (fragment case)

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

6. Let *previous element* be the element immediately above *last table* in the stack of open elements.

7. Let *adjusted insertion location* be inside *previous element*, after its last child (if any).

Note *These steps are involved in part because it's possible for elements, the `table` element in this case in particular, to have been moved by a script around in the DOM, or indeed removed from the DOM entirely, after the element was inserted by the parser.*

↪ **Otherwise**

Let *adjusted insertion location* be inside *target*, after its last child (if any).

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

4. Return the *adjusted insertion location*.

When the steps below require the UA to **create an element for a token** in a particular *given namespace* and with a particular *intended parent*, the UA must run the following steps:

1. Create a node implementing the interface appropriate for the element type corresponding to the tag name of the token in *given namespace* (as given in the specification that defines that element, e.g. for an `a` element in the HTML namespace, this specification defines it to be the `HTMLAnchorElement` interface), with the tag name being the name of that element, with the node being in the given namespace, and with the attributes on the node being those given in the given token.

   The interface appropriate for an element in the HTML namespace that is not defined in this specification (or other applicable specifications) is `HTMLUnknownElement`. Elements in other namespaces whose interface is not defined by that namespace's specification must use the interface `Element`.

   The `ownerDocument` of the newly created element must be the same as that of the *intended parent*.

2. If the newly created element has an `xmlns` attribute *in the XMLNS namespace* whose value is not exactly the same as the element's namespace, that is a parse error. Similarly, if the newly created element has an `xmlns:xlink` attribute in the XMLNS namespace whose value is not the XLink Namespace, that is a parse error.

3. If the newly created element is a resettable element, invoke its reset algorithm. (This initializes the element's value and checkedness based on the element's attributes.)

4. If the element is a form-associated element, and the `form` element pointer is not null, and there is no `template` element on the stack of open elements, and the newly created element is either not reassociateable or doesn't have a `form` attribute, and the *intended parent* is in the same home subtree as the element pointed to by the `form` element pointer, associate the newly created element with the `form` element pointed to by the `form` element pointer, and suppress the running of the reset the form owner algorithm when the parser subsequently attempts to insert the element.

5. Return the newly created element.

When the steps below require the user agent to **insert a foreign element** for a token in a given namespace, the user agent must run these steps:

1. Let the *adjusted insertion location* be the appropriate place for inserting a node.
2. Create an element for the token in the given namespace, with the intended parent being the element in which the *adjusted insertion location* finds itself.
3. If it is possible to insert an element at the *adjusted insertion location*, then insert the newly created element at the *adjusted insertion location*.

   Note *If the* adjusted insertion location *cannot accept more elements, e.g. because it's a* `Document` *that already has an element child, then the newly created element is dropped on the floor.*
4. Push the element onto the stack of open elements so that it is the new current node.
5. Return the newly created element.

When the steps below require the user agent to **insert an HTML element** for a token, the user agent must insert a foreign element for the token, in the HTML namespace.

When the steps below require the user agent to **adjust MathML attributes** for a token, then, if the token has an attribute named `definitionurl`, change its name to `definitionURL` (note the case difference).

When the steps below require the user agent to **adjust SVG attributes** for a token, then, for each attribute on the token whose attribute name is one of the ones in the first column of the following table, change the attribute's name to the name given in the corresponding cell in the second column. (This fixes the case of SVG attributes that are not all lowercase.)

Attribute name on token	Attribute name on element
attributename	attributeName
attributetype	attributeType
basefrequency	baseFrequency
baseprofile	baseProfile
calcmode	calcMode
clippathunits	clipPathUnits
diffuseconstant	diffuseConstant
edgemode	edgeMode
externalresourcesrequired	externalResourcesRequired
filterunits	filterUnits
glyphref	glyphRef
gradienttransform	gradientTransform
gradientunits	gradientUnits
kernelmatrix	kernelMatrix
kernelunitlength	kernelUnitLength
keypoints	keyPoints
keysplines	keySplines
keytimes	keyTimes
lengthadjust	lengthAdjust
limitingconeangle	limitingConeAngle
markerheight	markerHeight
markerunits	markerUnits
markerwidth	markerWidth
maskcontentunits	maskContentUnits
maskunits	maskUnits
numoctaves	numOctaves
pathlength	pathLength
patterncontentunits	patternContentUnits
patterntransform	patternTransform
patternunits	patternUnits
pointsatx	pointsAtX

pointsaty	pointsAtY
pointsatz	pointsAtZ
preservealpha	preserveAlpha
preserveaspectratio	preserveAspectRatio
primitiveunits	primitiveUnits
refx	refX
refy	refY
repeatcount	repeatCount
repeatdur	repeatDur
requiredextensions	requiredExtensions
requiredfeatures	requiredFeatures
specularconstant	specularConstant
specularexponent	specularExponent
spreadmethod	spreadMethod
startoffset	startOffset
stddeviation	stdDeviation
stitchtiles	stitchTiles
surfacescale	surfaceScale
systemlanguage	systemLanguage
tablevalues	tableValues
targetx	targetX
targety	targetY
textlength	textLength
viewbox	viewBox
viewtarget	viewTarget
xchannelselector	xChannelSelector
ychannelselector	yChannelSelector
zoomandpan	zoomAndPan

When the steps below require the user agent to **adjust foreign attributes** for a token, then, if any of the attributes on the token match the strings given in the first column of the following table, let the attribute be a namespaced attribute, with the prefix being the string given in the corresponding cell in the second column, the local name being the string given in the corresponding cell in the third column, and the namespace being the namespace given in the corresponding cell in the fourth column. (This fixes the use of namespaced attributes, in particular `lang` attributes in the XML namespace.)

Attribute name	Prefix	Local name	Namespace
xlink:actuate	xlink	actuate	XLink namespace
xlink:arcrole	xlink	arcrole	XLink namespace
xlink:href	xlink	href	XLink namespace
xlink:role	xlink	role	XLink namespace
xlink:show	xlink	show	XLink namespace
xlink:title	xlink	title	XLink namespace
xlink:type	xlink	type	XLink namespace
xml:base	xml	base	XML namespace
xml:lang	xml	lang	XML namespace
xml:space	xml	space	XML namespace
xmlns	(none)	xmlns	XMLNS namespace
xmlns:xlink	xmlns	xlink	XMLNS namespace

When the steps below require the user agent to **insert a character** while processing a token, the user agent must run the following steps:

1. Let *data* be the characters passed to the algorithm, or, if no characters were explicitly specified, the character of

the character token being processed.

2. Let the *adjusted insertion location* be the appropriate place for inserting a node.
3. If the *adjusted insertion location* is in a `Document` node, then abort these steps.

   Note *The DOM will not let `Document` nodes have `Text` node children, so they are dropped on the floor.*

4. If there is a `Text` node immediately before the *adjusted insertion location*, then append *data* to that `Text` node's data.

   Otherwise, create a new `Text` node whose data is *data* and whose `ownerDocument` is the same as that of the element in which the *adjusted insertion location* finds itself, and insert the newly created node at the *adjusted insertion location*.

Example Here are some sample inputs to the parser and the corresponding number of `Text` nodes that they result in, assuming a user agent that executes scripts.

Input	Number of `Text` nodes

`A<script>`
`var script = document.getElementsByTagName('script')`
`[0];`
`document.body.removeChild(script);`
`</script>B`
	One `Text` node in the document, containing "AB".

`A<script>`
`var text = document.createTextNode('B');`
`document.body.appendChild(text);`
`</script>C`
	Three `Text` nodes; "A" before the script, the script's contents, and "BC" after the script (the parser appends to the `Text` node created by the script).

`A<script>`
`var text = document.getElementsByTagName('script')`
`[0].firstChild;`
`text.data = 'B';`
`document.body.appendChild(text);`
`</script>C`
	Two adjacent `Text` nodes in the document, containing "A" and "BC".

`A<table>B<tr>C</tr>D</table>`	One `Text` node before the table, containing "ABCD". (This is caused by foster parenting.)
`A<table><tr> B</tr> C</table>`	One `Text` node before the table, containing "A B C" (A-space-B-space-C). (This is caused by foster parenting.)
`A<table><tr> B</tr> </em>C</table>`	One `Text` node before the table, containing "A BC" (A-space-B-C), and one `Text` node inside the table (as a child of a `tbody`) with a single space character. (Space characters separated from non-space characters by non-character tokens are not affected by foster parenting, even if those other tokens then get ignored.)

When the steps below require the user agent to **insert a comment** while processing a comment token, optionally with an explicitly insertion position *position*, the user agent must run the following steps:

1. Let *data* be the data given in the comment token being processed.
2. If *position* was specified, then let the *adjusted insertion location* be *position*. Otherwise, let *adjusted insertion location* be the appropriate place for inserting a node.
3. Create a `Comment` node whose `data` attribute is set to *data* and whose `ownerDocument` is the same as that of the node in which the *adjusted insertion location* finds itself.
4. Insert the newly created node at the *adjusted insertion location*.

DOM mutation events must not fire for changes caused by the UA parsing the document. This includes the parsing of any content inserted using `document.write()` and `document.writeln()` calls. [DOMEVENTS]

However, mutation observers *do* fire, as required by the DOM specification.

12.2.5.2 Parsing elements that contain only text

The **generic raw text element parsing algorithm** and the **generic RCDATA element parsing algorithm** consist of the following steps. These algorithms are always invoked in response to a start tag token.

1. Insert an HTML element for the token.
2. If the algorithm that was invoked is the generic raw text element parsing algorithm, switch the tokenizer to the RAWTEXT state; otherwise the algorithm invoked was the generic RCDATA element parsing algorithm, switch the tokenizer to the RCDATA state.
3. Let the original insertion mode be the current insertion mode.
4. Then, switch the insertion mode to "text".

12.2.5.3 Closing elements that have implied end tags

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

If a step requires the UA to generate implied end tags but lists an element to exclude from the process, then the UA must perform the above steps as if that element was not in the above list.

12.2.5.4 The rules for parsing tokens in HTML content

12.2.5.4.1 The "initial" insertion mode

When the user agent is to apply the rules for the "initial" insertion mode, the user agent must handle the token as follows:

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

Ignore the token.

↪ **A comment token**

Insert a comment as the last child of the `Document` object.

↪ **A DOCTYPE token**

If the DOCTYPE token's name is not a case-sensitive match for the string "`html`", or the token's public identifier is not missing, or the token's system identifier is neither missing nor a case-sensitive match for the string "`about:legacy-compat`", and none of the sets of conditions in the following list are matched, then there is a parse error.

- The DOCTYPE token's name is a case-sensitive match for the string "`html`", the token's public identifier is the case-sensitive string "`-//W3C//DTD HTML 4.0//EN`", and the token's system identifier is either missing or the case-sensitive string "`http://www.w3.org/TR/REC-html40/strict.dtd`".
- The DOCTYPE token's name is a case-sensitive match for the string "`html`", the token's public identifier is the case-sensitive string "`-//W3C//DTD HTML 4.01//EN`", and the token's system identifier is either missing or the case-sensitive string "`http://www.w3.org/TR/html4/strict.dtd`".

- The DOCTYPE token's name is a case-sensitive match for the string "html", the token's public identifier is the case-sensitive string "-//W3C//DTD XHTML 1.0 Strict//EN", and the token's system identifier is the case-sensitive string "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd".

- The DOCTYPE token's name is a case-sensitive match for the string "html", the token's public identifier is the case-sensitive string "-//W3C//DTD XHTML 1.1//EN", and the token's system identifier is the case-sensitive string "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd".

Conformance checkers may, based on the values (including presence or lack thereof) of the DOCTYPE token's name, public identifier, or system identifier, switch to a conformance checking mode for another language (e.g. based on the DOCTYPE token a conformance checker could recognise that the document is an HTML4-era document, and defer to an HTML4 conformance checker.)

Append a DocumentType node to the Document node, with the name attribute set to the name given in the DOCTYPE token, or the empty string if the name was missing; the publicId attribute set to the public identifier given in the DOCTYPE token, or the empty string if the public identifier was missing; the systemId attribute set to the system identifier given in the DOCTYPE token, or the empty string if the system identifier was missing; and the other attributes specific to DocumentType objects set to null and empty lists as appropriate. Associate the DocumentType node with the Document object so that it is returned as the value of the doctype attribute of the Document object.

Then, if the document is *not* an iframe srcdoc document, and the DOCTYPE token matches one of the conditions in the following list, then set the Document to quirks mode:

- The *force-quirks flag* is set to *on*.
- The name is set to anything other than "html" (compared case-sensitively).
- The public identifier is set to: "-//W3O//DTD W3 HTML Strict 3.0//EN//"
- The public identifier is set to: "-/W3C/DTD HTML 4.0 Transitional/EN"
- The public identifier is set to: "HTML"
- The system identifier is set to: "http://www.ibm.com/data/dtd/v11/ibmxhtml1-transitional.dtd"
- The public identifier starts with: "+//Silmaril//dtd html Pro v0r11 19970101//"
- The public identifier starts with: "-//AS//DTD HTML 3.0 asWedit + extensions//"
- The public identifier starts with: "-//AdvaSoft Ltd//DTD HTML 3.0 asWedit + extensions//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0 Strict//"
- The public identifier starts with: "-//IETF//DTD HTML 2.0//"
- The public identifier starts with: "-//IETF//DTD HTML 2.1E//"
- The public identifier starts with: "-//IETF//DTD HTML 3.0//"
- The public identifier starts with: "-//IETF//DTD HTML 3.2 Final//"
- The public identifier starts with: "-//IETF//DTD HTML 3.2//"
- The public identifier starts with: "-//IETF//DTD HTML 3//"
- The public identifier starts with: "-//IETF//DTD HTML Level 0//"
- The public identifier starts with: "-//IETF//DTD HTML Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML Level 3//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 0//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 1//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 2//"
- The public identifier starts with: "-//IETF//DTD HTML Strict Level 3//"
- The public identifier starts with: "-//IETF//DTD HTML Strict//"
- The public identifier starts with: "-//IETF//DTD HTML//"
- The public identifier starts with: "-//Metrius//DTD Metrius Presentational//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 HTML Strict//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 HTML//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 2.0 Tables//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 HTML Strict//"
- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 HTML//"

- The public identifier starts with: "-//Microsoft//DTD Internet Explorer 3.0 Tables//"
- The public identifier starts with: "-//Netscape Comm. Corp.//DTD HTML//"
- The public identifier starts with: "-//Netscape Comm. Corp.//DTD Strict HTML//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML 2.0//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML Extended 1.0//"
- The public identifier starts with: "-//O'Reilly and Associates//DTD HTML Extended Relaxed 1.0//"
- The public identifier starts with: "-//SQ//DTD HTML 2.0 HoTMetaL + extensions//"
- The public identifier starts with: "-//SoftQuad Software//DTD HoTMetaL PRO 6.0::19990601::extensions to HTML 4.0//"
- The public identifier starts with: "-//SoftQuad//DTD HoTMetaL PRO 4.0::19971010::extensions to HTML 4.0//"
- The public identifier starts with: "-//Spyglass//DTD HTML 2.0 Extended//"
- The public identifier starts with: "-//Sun Microsystems Corp.//DTD HotJava HTML//"
- The public identifier starts with: "-//Sun Microsystems Corp.//DTD HotJava Strict HTML//"
- The public identifier starts with: "-//W3C//DTD HTML 3 1995-03-24//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2 Draft//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2 Final//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2//"
- The public identifier starts with: "-//W3C//DTD HTML 3.2S Draft//"
- The public identifier starts with: "-//W3C//DTD HTML 4.0 Frameset//"
- The public identifier starts with: "-//W3C//DTD HTML 4.0 Transitional//"
- The public identifier starts with: "-//W3C//DTD HTML Experimental 19960712//"
- The public identifier starts with: "-//W3C//DTD HTML Experimental 970421//"
- The public identifier starts with: "-//W3C//DTD W3 HTML//"
- The public identifier starts with: "-//W3O//DTD W3 HTML 3.0//"
- The public identifier starts with: "-//WebTechs//DTD Mozilla HTML 2.0//"
- The public identifier starts with: "-//WebTechs//DTD Mozilla HTML//"
- The system identifier is missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Frameset//"
- The system identifier is missing and the public identifier starts with: "-//W3C//DTD HTML 4.01 Transitional//"

Otherwise, if the document is *not* an `iframe srcdoc` document, and the DOCTYPE token matches one of the conditions in the following list, then set the `Document` to limited-quirks mode:

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

The system identifier and public identifier strings must be compared to the values given in the lists above in an ASCII case-insensitive manner. A system identifier whose value is the empty string is not considered missing for the purposes of the conditions above.

Then, switch the insertion mode to "before html".

↪ **Anything else**

If the document is *not* an `iframe srcdoc` document, then this is a parse error; set the `Document` to quirks mode.

In any case, switch the insertion mode to "before html", then reprocess the token.

12.2.5.4.2 The "before html" insertion mode

When the user agent is to apply the rules for the "before html" insertion mode, the user agent must handle the token as follows:

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A comment token**

Insert a comment as the last child of the `Document` object.

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

Ignore the token.

↳ **A start tag whose tag name is "html"**

Create an element for the token in the HTML namespace, with the `Document` as the intended parent. Append it to the `Document` object. Put this element in the stack of open elements.

If the `Document` is being loaded as part of navigation of a browsing context, then: if the newly created element has a `manifest` attribute whose value is not the empty string, then resolve the value of that attribute to an absolute URL, relative to the newly created element, and if that is successful, run the application cache selection algorithm with the result of applying the URL serializer algorithm to the resulting parsed URL with the *exclude fragment flag* set; otherwise, if there is no such attribute, or its value is the empty string, or resolving its value fails, run the application cache selection algorithm with no manifest. The algorithm must be passed the `Document` object.

Switch the insertion mode to "before head".

↳ **An end tag whose tag name is one of: "head", "body", "html", "br"**

Act as described in the "anything else" entry below.

↳ **Any other end tag**

Parse error. Ignore the token.

↳ **Anything else**

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

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

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

The root element can end up being removed from the `Document` object, e.g. by scripts; nothing in particular happens in such cases, content continues being appended to the nodes as described in the next section.

12.2.5.4.3 The "before head" insertion mode

When the user agent is to apply the rules for the "before head" insertion mode, the user agent must handle the token as follows:

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

Ignore the token.

↳ **A comment token**

Insert a comment.

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A start tag whose tag name is "html"**

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

↳ **A start tag whose tag name is "head"**

Insert an HTML element for the token.

Set the `head` element pointer to the newly created `head` element.

Switch the insertion mode to "in head".

↪ **An end tag whose tag name is one of: "head", "body", "html", "br"**

Act as described in the "anything else" entry below.

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Insert an HTML element for a "head" start tag token with no attributes.

Set the `head` element pointer to the newly created `head` element.

Switch the insertion mode to "in head".

Reprocess the current token.

12.2.5.4.4 The "in head" insertion mode

When the user agent is to apply the rules for the "in head" insertion mode, the user agent must handle the token as follows:

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

Insert the character.

↪ **A comment token**

Insert a comment.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

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

↪ **A start tag whose tag name is one of: "base", "basefont", "bgsound", "link"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "meta"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

If the element has a `charset` attribute, and getting an encoding from its value results in a supported ASCII-compatible character encoding or a UTF-16 encoding, and the confidence is currently *tentative*, then change the encoding to the resulting encoding.

Otherwise, if the element has an `http-equiv` attribute whose value is an ASCII case-insensitive match for the string "`Content-Type`", and the element has a `content` attribute, and applying the algorithm for extracting a character encoding from a `meta` element to that attribute's value returns a supported ASCII-compatible character encoding or a UTF-16 encoding, and the confidence is currently *tentative*, then change the encoding to the extracted encoding.

↪ **A start tag whose tag name is "title"**

Follow the generic RCDATA element parsing algorithm.

↪ **A start tag whose tag name is "noscript", if the scripting flag is enabled**

↪ **A start tag whose tag name is one of: "noframes", "style"**

Follow the generic raw text element parsing algorithm.

↪ **A start tag whose tag name is "noscript", if the scripting flag is disabled**

Insert an HTML element for the token.

Switch the insertion mode to "in head noscript".

↪ **A start tag whose tag name is "script"**

Run these steps:

1. Let the *adjusted insertion location* be the appropriate place for inserting a node.
2. Create an element for the token in the HTML namespace, with the intended parent being the element in which the *adjusted insertion location* finds itself.
3. Mark the element as being "parser-inserted" and unset the element's "force-async" flag.

   Note *This ensures that, if the script is external, any `document.write()` calls in the script will execute in-line, instead of blowing the document away, as would happen in most other cases. It also prevents the script from executing until the end tag is seen.*
4. If the parser was originally created for the HTML fragment parsing algorithm, then mark the `script` element as "already started". (fragment case)
5. Insert the newly created element at the *adjusted insertion location*.
6. Push the element onto the stack of open elements so that it is the new current node.
7. Switch the tokenizer to the script data state.
8. Let the original insertion mode be the current insertion mode.
9. Switch the insertion mode to "text".

↪ **An end tag whose tag name is "head"**

Pop the current node (which will be the `head` element) off the stack of open elements.

Switch the insertion mode to "after head".

↪ **An end tag whose tag name is one of: "body", "html", "br"**

Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is "template"**

Insert an HTML element for the token.

Insert a marker at the end of the list of active formatting elements.

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

Switch the insertion mode to "in template".

Push "in template" onto the stack of template insertion modes so that it is the new current template insertion mode.

↪ **An end tag whose tag name is "template"**

If there is no `template` element on the stack of open elements, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not a `template` element, then this is a parse error.

3. Pop elements from the stack of open elements until a `template` element has been popped from the stack.
4. Clear the list of active formatting elements up to the last marker.
5. Pop the current template insertion mode off the stack of template insertion modes.
6. Reset the insertion mode appropriately.

↪ **A start tag whose tag name is "head"**

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Pop the current node (which will be the `head` element) off the stack of open elements.

Switch the insertion mode to "after head".

Reprocess the token.

12.2.5.4.5 The "in head noscript" insertion mode

When the user agent is to apply the rules for the "in head noscript" insertion mode, the user agent must handle the token as follows:

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

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

↪ **An end tag whose tag name is "noscript"**

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

Switch the insertion mode to "in head".

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

↪ **A comment token**

↪ **A start tag whose tag name is one of: "basefont", "bgsound", "link", "meta", "noframes", "style"**

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

↪ **An end tag whose tag name is "br"**

Act as described in the "anything else" entry below.

↪ **A start tag whose tag name is one of: "head", "noscript"**

↪ **Any other end tag**

Parse error. Ignore the token.

↪ **Anything else**

Parse error.

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

Switch the insertion mode to "in head".

Reprocess the token.

12.2.5.4.6 The "after head" insertion mode

When the user agent is to apply the rules for the "after head" insertion mode, the user agent must handle the token as follows:

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

Insert the character.

↳ **A comment token**

Insert a comment.

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A start tag whose tag name is "html"**

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

↳ **A start tag whose tag name is "body"**

Insert an HTML element for the token.

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

Switch the insertion mode to "in body".

↳ **A start tag whose tag name is "frameset"**

Insert an HTML element for the token.

Switch the insertion mode to "in frameset".

↳ **A start tag whose tag name is one of: "base", "basefont", "bgsound", "link", "meta", "noframes", "script", "style", "template", "title"**

Parse error.

Push the node pointed to by the `head` element pointer onto the stack of open elements.

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

Remove the node pointed to by the `head` element pointer from the stack of open elements. (It might not be the current node at this point.)

Note *The `head` element pointer cannot be null at this point.*

↳ **An end tag whose tag name is "template"**

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

↳ **An end tag whose tag name is one of: "body", "html", "br"**

Act as described in the "anything else" entry below.

↳ **A start tag whose tag name is "head"**

↳ **Any other end tag**

Parse error. Ignore the token.

↳ **Anything else**

Insert an HTML element for a "body" start tag token with no attributes.

Switch the insertion mode to "in body".

Reprocess the current token.

12.2.5.4.7 The "in body" insertion mode

When the user agent is to apply the rules for the "in body" insertion mode, the user agent must handle the token as follows:

↪ **A character token that is U+0000 NULL**

Parse error. Ignore the token.

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

Reconstruct the active formatting elements, if any.

Insert the token's character.

↪ **Any other character token**

Reconstruct the active formatting elements, if any.

Insert the token's character.

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

↪ **A comment token**

Insert a comment.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

Parse error.

If there is a `template` element on the stack of open elements, then ignore the token.

Otherwise, for each attribute on the token, check to see if the attribute is already present on the top element of the stack of open elements. If it is not, add the attribute and its corresponding value to that element.

↪ **A start tag whose tag name is one of: "base", "basefont", "bgsound", "link", "meta", "noframes", "script", "style", "template", "title"**

↪ **An end tag whose tag name is "template"**

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

↪ **A start tag whose tag name is "body"**

Parse error.

If the second element on the stack of open elements is not a `body` element, if the stack of open elements has only one node on it, or if there is a `template` element on the stack of open elements, then ignore the token. (fragment case)

Otherwise, set the frameset-ok flag to "not ok"; then, for each attribute on the token, check to see if the attribute is already present on the `body` element (the second element) on the stack of open elements, and if it is not, add the attribute and its corresponding value to that element.

↪ **A start tag whose tag name is "frameset"**

Parse error.

If the stack of open elements has only one node on it, or if the second element on the stack of open elements is not a `body` element, then ignore the token. (fragment case)

If the frameset-ok flag is set to "not ok", ignore the token.

Otherwise, run the following steps:

1. Remove the second element on the stack of open elements from its parent node, if it has one.
2. Pop all the nodes from the bottom of the stack of open elements, from the current node up to, but not including, the root `html` element.
3. Insert an HTML element for the token.
4. Switch the insertion mode to "in frameset".

↳ **An end-of-file token**

If there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, a `p` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

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

Otherwise, stop parsing.

↳ **An end tag whose tag name is "body"**

If the stack of open elements does not have a `body` element in scope, this is a parse error; ignore the token.

Otherwise, if there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, an `optgroup` element, an `option` element, a `p` element, an `rp` element, an `rt` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

Switch the insertion mode to "after body".

↳ **An end tag whose tag name is "html"**

If the stack of open elements does not have a `body` element in scope, this is a parse error; ignore the token.

Otherwise, if there is a node in the stack of open elements that is not either a `dd` element, a `dt` element, an `li` element, an `optgroup` element, an `option` element, a `p` element, an `rp` element, an `rt` element, a `tbody` element, a `td` element, a `tfoot` element, a `th` element, a `thead` element, a `tr` element, the `body` element, or the `html` element, then this is a parse error.

Switch the insertion mode to "after body".

Reprocess the token.

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

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

Insert an HTML element for the token.

↳ **A start tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

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

If the current node is an HTML element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error; pop the current node off the stack of open elements.

Insert an HTML element for the token.

↳ **A start tag whose tag name is one of: "pre", "listing"**

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

Insert an HTML element for the token.

If the next token is a U+000A LINE FEED (LF) character token, then ignore that token and move on to the next one. (Newlines at the start of `pre` blocks are ignored as an authoring convenience.)

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

↪ **A start tag whose tag name is "form"**

If the `form` element pointer is not null, and there is no `template` element on the stack of open elements, then this is a parse error; ignore the token.

Otherwise:

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

Insert an HTML element for the token, and, if there is no `template` element on the stack of open elements, set the `form` element pointer to point to the element created.

↪ **A start tag whose tag name is "li"**

Run these steps:

1. Set the frameset-ok flag to "not ok".
2. Initialise *node* to be the current node (the bottommost node of the stack).
3. *Loop*: If *node* is an `li` element, then run these substeps:
    1. Generate implied end tags, except for `li` elements.
    2. If the current node is not an `li` element, then this is a parse error.
    3. Pop elements from the stack of open elements until an `li` element has been popped from the stack.
    4. Jump to the step labeled *done* below.
4. If *node* is in the special category, but is not an `address`, `div`, or `p` element, then jump to the step labeled *done* below.
5. Otherwise, set *node* to the previous entry in the stack of open elements and return to the step labeled *loop*.
6. *Done*: If the stack of open elements has a `p` element in button scope, then close a `p` element.
7. Finally, insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "dd", "dt"**

Run these steps:

1. Set the frameset-ok flag to "not ok".
2. Initialise *node* to be the current node (the bottommost node of the stack).
3. *Loop*: If *node* is a `dd` element, then run these substeps:
    1. Generate implied end tags, except for `dd` elements.
    2. If the current node is not a `dd` element, then this is a parse error.
    3. Pop elements from the stack of open elements until a `dd` element has been popped from the stack.
    4. Jump to the step labeled *done* below.
4. If *node* is a `dt` element, then run these substeps:
    1. Generate implied end tags, except for `dt` elements.
    2. If the current node is not a `dt` element, then this is a parse error.
    3. Pop elements from the stack of open elements until a `dt` element has been popped from the

stack.

4. Jump to the step labeled *done* below.

5. If *node* is in the special category, but is not an `address`, `div`, or `p` element, then jump to the step labeled *done* below.

6. Otherwise, set *node* to the previous entry in the stack of open elements and return to the step labeled *loop*.

7. *Done*: If the stack of open elements has a `p` element in button scope, then close a `p` element.

8. Finally, insert an HTML element for the token.

↪ **A start tag whose tag name is "plaintext"**

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

Insert an HTML element for the token.

Switch the tokenizer to the PLAINTEXT state.

Note *Once a start tag with the tag name "plaintext" has been seen, that will be the last token ever seen other than character tokens (and the end-of-file token), because there is no way to switch out of the PLAINTEXT state.*

↪ **A start tag whose tag name is "button"**

1. If the stack of open elements has a `button` element in scope, then run these substeps:

    1. Parse error.

    2. Generate implied end tags.

    3. Pop elements from the stack of open elements until a `button` element has been popped from the stack.

2. Reconstruct the active formatting elements, if any.

3. Insert an HTML element for the token.

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

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

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

Otherwise, run these steps:

1. Generate implied end tags.

2. If the current node is not an HTML element with the same tag name as that of the token, then this is a parse error.

3. Pop elements from the stack of open elements until an HTML element with the same tag name as the token has been popped from the stack.

↪ **An end tag whose tag name is "form"**

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

1. Let *node* be the element that the `form` element pointer is set to, or null if it is not set to an element.

2. Set the `form` element pointer to null. Otherwise, let *node* be null.

3. If *node* is null or if the stack of open elements does not have *node* in scope, then this is a parse

error; abort these steps and ignore the token.

4. Generate implied end tags.
5. If the current node is not *node*, then this is a parse error.
6. Remove *node* from the stack of open elements.

If there *is* a `template` element on the stack of open elements, then run these substeps instead:

1. If the stack of open elements does not have a `form` element in scope, then this is a parse error; abort these steps and ignore the token.
2. Generate implied end tags.
3. If the current node is not a `form` element, then this is a parse error.
4. Pop elements from the stack of open elements until a `form` element has been popped from the stack.

↪ **An end tag whose tag name is "p"**

If the stack of open elements does not have a `p` element in button scope, then this is a parse error; insert an HTML element for a "p" start tag token with no attributes.

Close a `p` element.

↪ **An end tag whose tag name is "li"**

If the stack of open elements does not have an `li` element in list item scope, then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags, except for `li` elements.
2. If the current node is not an `li` element, then this is a parse error.
3. Pop elements from the stack of open elements until an `li` element has been popped from the stack.

↪ **An end tag whose tag name is one of: "dd", "dt"**

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

Otherwise, run these steps:

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

↪ **An end tag whose tag name is one of: "h1", "h2", "h3", "h4", "h5", "h6"**

If the stack of open elements does not have an element in scope that is an HTML element and whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6", then this is a parse error; ignore the token.

Otherwise, run these steps:

1. Generate implied end tags.
2. If the current node is not an HTML element with the same tag name as that of the token, then this is a parse error.
3. Pop elements from the stack of open elements until an HTML element whose tag name is one of "h1", "h2", "h3", "h4", "h5", or "h6" has been popped from the stack.

↪ **An end tag whose tag name is "sarcasm"**

Take a deep breath, then act as described in the "any other end tag" entry below.

↪ **A start tag whose tag name is "a"**

If the list of active formatting elements contains an a element between the end of the list and the last marker on the list (or the start of the list if there is no marker on the list), then this is a parse error; run the adoption agency algorithm for the tag name "a", then remove that element from the list of active formatting elements and the stack of open elements if the adoption agency algorithm didn't already remove it (it might not have if the element is not in table scope).

Example In the non-conforming stream `<a href="a">a<table><a href="b">b</table>x`, the first a element would be closed upon seeing the second one, and the "x" character would be inside a link to "b", not to "a". This is despite the fact that the outer a element is not in table scope (meaning that a regular `</a>` end tag at the start of the table wouldn't close the outer a element). The result is that the two a elements are indirectly nested inside each other — non-conforming markup will often result in non-conforming DOMs when parsed.

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Push onto the list of active formatting elements that element.

↪ **A start tag whose tag name is one of: "b", "big", "code", "em", "font", "i", "s", "small", "strike", "strong", "tt", "u"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Push onto the list of active formatting elements that element.

↪ **A start tag whose tag name is "nobr"**

Reconstruct the active formatting elements, if any.

If the stack of open elements has a `nobr` element in scope, then this is a parse error; run the adoption agency algorithm for the tag name "nobr", then once again reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Push onto the list of active formatting elements that element.

↪ **An end tag whose tag name is one of: "a", "b", "big", "code", "em", "font", "i", "nobr", "s", "small", "strike", "strong", "tt", "u"**

Run the adoption agency algorithm for the token's tag name.

↪ **A start tag whose tag name is one of: "applet", "marquee", "object"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Insert a marker at the end of the list of active formatting elements.

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

↪ **An end tag token whose tag name is one of: "applet", "marquee", "object"**

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

Otherwise, run these steps:

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

4. Clear the list of active formatting elements up to the last marker.

↪ **A start tag whose tag name is "table"**

If the `Document` is *not* set to quirks mode, and the stack of open elements has a `p` element in button scope, then close a `p` element.

Insert an HTML element for the token.

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

Switch the insertion mode to "in table".

↪ **An end tag whose tag name is "br"**

Parse error. Drop the attributes from the token, and act as described in the next entry; i.e. act as if this was a "br" start tag token with no attributes, rather than the end tag token that it actually is.

↪ **A start tag whose tag name is one of: "area", "br", "embed", "img", "keygen", "wbr"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

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

↪ **A start tag whose tag name is "input"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

If the token does not have an attribute with the name "type", or if it does, but that attribute's value is not an ASCII case-insensitive match for the string "`hidden`", then: set the frameset-ok flag to "not ok".

↪ **A start tag whose tag name is one of: "menuitem", "param", "source", "track"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "hr"**

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

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

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

↪ **A start tag whose tag name is "image"**

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

↪ **A start tag whose tag name is "isindex"**

Parse error.

If there is no `template` element on the stack of open elements and the `form` element pointer is not null, then ignore the token.

Otherwise:

Acknowledge the token's *self-closing flag*, if it is set.

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

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

Insert an HTML element for a "form" start tag token with no attributes, and, if there is no `template` element on the stack of open elements, set the `form` element pointer to point to the element created.

If the token has an attribute called "action", set the `action` attribute on the resulting `form` element to the value of the "action" attribute of the token.

Insert an HTML element for an "hr" start tag token with no attributes. Immediately pop the current node off the stack of open elements.

Reconstruct the active formatting elements, if any.

Insert an HTML element for a "label" start tag token with no attributes.

Insert characters (see below for what they should say).

Insert an HTML element for an "input" start tag token with all the attributes from the "isindex" token except "name", "action", and "prompt", and with an attribute named "name" with the value "isindex". (This creates an `input` element with the `name` attribute set to the magic value "`isindex`".) Immediately pop the current node off the stack of open elements.

Insert more characters (see below for what they should say).

Pop the current node (which will be the `label` element created earlier) off the stack of open elements.

Insert an HTML element for an "hr" start tag token with no attributes. Immediately pop the current node off the stack of open elements.

Pop the current node (which will be the `form` element created earlier) off the stack of open elements, and, if there is no `template` element on the stack of open elements, set the `form` element pointer back to null.

Prompt: If the token has an attribute with the name "prompt", then the first stream of characters must be the same string as given in that attribute, and the second stream of characters must be empty. Otherwise, the two streams of character tokens together should, together with the `input` element, express the equivalent of "This is a searchable index. Enter search keywords: (input field)" in the user's preferred language.

↪ **A start tag whose tag name is "textarea"**

Run these steps:

1. Insert an HTML element for the token.
2. If the next token is a U+000A LINE FEED (LF) character token, then ignore that token and move on to the next one. (Newlines at the start of `textarea` elements are ignored as an authoring convenience.)
3. Switch the tokenizer to the RCDATA state.
4. Let the original insertion mode be the current insertion mode.
5. Set the frameset-ok flag to "not ok".
6. Switch the insertion mode to "text".

↪ **A start tag whose tag name is "xmp"**

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

Reconstruct the active formatting elements, if any.

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

Follow the generic raw text element parsing algorithm.

↪ **A start tag whose tag name is "iframe"**

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

Follow the generic raw text element parsing algorithm.

↪ **A start tag whose tag name is "noembed"**

↪ **A start tag whose tag name is "noscript", if the scripting flag is enabled**

Follow the generic raw text element parsing algorithm.

↪ **A start tag whose tag name is "select"**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

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

If the insertion mode is one of "in table", "in caption", "in table body", "in row", or "in cell", then switch the insertion mode to "in select in table". Otherwise, switch the insertion mode to "in select".

↪ **A start tag whose tag name is one of: "optgroup", "option"**

If the current node is an `option` element, then pop the current node off the stack of open elements.

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

↪ **A start tag whose tag name is one of: "rp", "rt"**

If the stack of open elements has a `ruby` element in scope, then generate implied end tags. If the current node is not then a `ruby` element, this is a parse error.

Insert an HTML element for the token.

↪ **A start tag whose tag name is "math"**

Reconstruct the active formatting elements, if any.

Adjust MathML attributes for the token. (This fixes the case of MathML attributes that are not all lowercase.)

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink.)

Insert a foreign element for the token, in the MathML namespace.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

↪ **A start tag whose tag name is "svg"**

Reconstruct the active formatting elements, if any.

Adjust SVG attributes for the token. (This fixes the case of SVG attributes that are not all lowercase.)

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink in SVG.)

Insert a foreign element for the token, in the SVG namespace.

If the token has its *self-closing flag* set, pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "frame", "head", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **Any other start tag**

Reconstruct the active formatting elements, if any.

Insert an HTML element for the token.

Note *This element will be an ordinary element.*

↪ **Any other end tag**

Run these steps:

1. Initialise *node* to be the current node (the bottommost node of the stack).
2. *Loop*: If *node* is an HTML element with the same tag name as the token, then:
    1. Generate implied end tags, except for HTML elements with the same tag name as the token.
    2. If *node* is not the current node, then this is a parse error.
    3. Pop all the nodes from the current node up to *node*, including *node*, then stop these steps.
3. Otherwise, if *node* is in the special category, then this is a parse error; ignore the token, and abort these steps.
4. Set *node* to the previous entry in the stack of open elements.
5. Return to the step labeled *loop*.

When the steps above say the user agent is to **close a `p` element**, it means that the user agent must run the following steps:

1. Generate implied end tags, except for p elements.
2. If the current node is not a p element, then this is a parse error.
3. Pop elements from the stack of open elements until a p element has been popped from the stack.

The **adoption agency algorithm**, which takes as its only argument a tag name *subject* for which the algorithm is being run, consists of the following steps:

1. If the current node is an HTML element whose tag name is *subject*, and the current node is not in the list of active formatting elements, then pop the current node off the stack of open elements, and abort these steps.
2. Let *outer loop counter* be zero.
3. *Outer loop*: If *outer loop counter* is greater than or equal to eight, then abort these steps.
4. Increment *outer loop counter* by one.
5. Let *formatting element* be the last element in the list of active formatting elements that:
    - is between the end of the list and the last scope marker in the list, if any, or the start of the list otherwise, and
    - has the tag name *subject*.

    If there is no such element, then abort these steps and instead act as described in the "any other end tag" entry above.
6. If *formatting element* is not in the stack of open elements, then this is a parse error; remove the element from the list, and abort these steps.
7. If *formatting element* is in the stack of open elements, but the element is not in scope, then this is a parse error; abort these steps.
8. If *formatting element* is not the current node, this is a parse error. (But do not abort these steps.)
9. Let *furthest block* be the topmost node in the stack of open elements that is lower in the stack than *formatting*

element, and is an element in the special category. There might not be one.

10. If there is no *furthest block*, then the UA must first pop all the nodes from the bottom of the stack of open elements, from the current node up to and including *formatting element*, then remove *formatting element* from the list of active formatting elements, and finally abort these steps.

11. Let *common ancestor* be the element immediately above *formatting element* in the stack of open elements.

12. Let a bookmark note the position of *formatting element* in the list of active formatting elements relative to the elements on either side of it in the list.

13. Let *node* and *last node* be *furthest block*. Follow these steps:

    1. Let *inner loop counter* be zero.

    2. *Inner loop*: Increment *inner loop counter* by one.

    3. Let *node* be the element immediately above *node* in the stack of open elements, or if *node* is no longer in the stack of open elements (e.g. because it got removed by this algorithm), the element that was immediately above *node* in the stack of open elements before *node* was removed.

    4. If *node* is *formatting element*, then go to the next step in the overall algorithm.

    5. If *inner loop counter* is greater than three and *node* is in the list of active formatting elements, then remove *node* from the list of active formatting elements.

    6. If *node* is not in the list of active formatting elements, then remove *node* from the stack of open elements and then go back to the step labeled *inner loop*.

    7. Create an element for the token for which the element *node* was created, in the HTML namespace, with *common ancestor* as the intended parent; replace the entry for *node* in the list of active formatting elements with an entry for the new element, replace the entry for *node* in the stack of open elements with an entry for the new element, and let *node* be the new element.

    8. If *last node* is *furthest block*, then move the aforementioned bookmark to be immediately after the new *node* in the list of active formatting elements.

    9. Insert *last node* into *node*, first removing it from its previous parent node if any.

    10. Let *last node* be *node*.

    11. Return to the step labeled *inner loop*.

14. Insert whatever *last node* ended up being in the previous step at the appropriate place for inserting a node, but using *common ancestor* as the *override target*.

15. Create an element for the token for which *formatting element* was created, in the HTML namespace, with *furthest block* as the intended parent.

16. Take all of the child nodes of *furthest block* and append them to the element created in the last step.

17. Append that new element to *furthest block*.

18. Remove *formatting element* from the list of active formatting elements, and insert the new element into the list of active formatting elements at the position of the aforementioned bookmark.

19. Remove *formatting element* from the stack of open elements, and insert the new element into the stack of open elements immediately below the position of *furthest block* in that stack.

20. Jump back to the step labeled *outer loop*.

Note *This algorithm's name, the "adoption agency algorithm", comes from the way it causes elements to change parents, and is in contrast with other possible algorithms for dealing with misnested content, which included the "incest algorithm", the "secret affair algorithm", and the "Heisenberg algorithm".*

12.2.5.4.8 The "text" insertion mode

When the user agent is to apply the rules for the "text" insertion mode, the user agent must handle the token as follows:

↳ **A character token**

Insert the token's character.

Note *This can never be a U+0000 NULL character; the tokenizer converts those to U+FFFD REPLACEMENT CHARACTER characters.*

↳ **An end-of-file token**

Parse error.

If the current node is a `script` element, mark the `script` element as "already started".

Pop the current node off the stack of open elements.

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

↳ **An end tag whose tag name is "script"**

If the stack of script settings objects is empty, perform a microtask checkpoint.

Let *script* be the current node (which will be a `script` element).

Pop the current node off the stack of open elements.

Switch the insertion mode to the original insertion mode.

Let the *old insertion point* have the same value as the current insertion point. Let the insertion point be just before the next input character.

Increment the parser's script nesting level by one.

Prepare the *script*. This might cause some script to execute, which might cause new characters to be inserted into the tokenizer, and might cause the tokenizer to output more tokens, resulting in a reentrant invocation of the parser.

Decrement the parser's script nesting level by one. If the parser's script nesting level is zero, then set the parser pause flag to false.

Let the insertion point have the value of the *old insertion point*. (In other words, restore the insertion point to its previous value. This value might be the "undefined" value.)

At this stage, if there is a pending parsing-blocking script, then:

↳ **If the script nesting level is not zero:**

Set the parser pause flag to true, and abort the processing of any nested invocations of the tokenizer, yielding control back to the caller. (Tokenization will resume when the caller returns to the "outer" tree construction stage.)

Note *The tree construction stage of this particular parser is being called reentrantly, say from a call to `document.write()`.*

↳ **Otherwise:**

Run these steps:

1. Let *the script* be the pending parsing-blocking script. There is no longer a pending parsing-blocking script.
2. Block the tokenizer for this instance of the HTML parser, such that the event loop will not run tasks that invoke the tokenizer.
3. If the parser's `Document` has a style sheet that is blocking scripts or *the script*'s

"ready to be parser-executed" flag is not set: spin the event loop until the parser's `Document` has no style sheet that is blocking scripts and *the script*'s "ready to be parser-executed" flag is set.

4. If this parser has been aborted in the meantime, abort these steps.

   Note *This could happen if, e.g., while the spin the event loop algorithm is running, the browsing context gets closed, or the `document.open()` method gets invoked on the `Document`.*

5. Unblock the tokenizer for this instance of the HTML parser, such that tasks that invoke the tokenizer can again be run.

6. Let the insertion point be just before the next input character.

7. Increment the parser's script nesting level by one (it should be zero before this step, so this sets it to one).

8. Execute *the script*.

9. Decrement the parser's script nesting level by one. If the parser's script nesting level is zero (which it always should be at this point), then set the parser pause flag to false.

10. Let the insertion point be undefined again.

11. If there is once again a pending parsing-blocking script, then repeat these steps from step 1.

↪ **Any other end tag**

Pop the current node off the stack of open elements.

Switch the insertion mode to the original insertion mode.

12.2.5.4.9 The "in table" insertion mode

When the user agent is to apply the rules for the "in table" insertion mode, the user agent must handle the token as follows:

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

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

Let the original insertion mode be the current insertion mode.

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

↪ **A comment token**

Insert a comment.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "caption"**

Clear the stack back to a table context. (See below.)

Insert a marker at the end of the list of active formatting elements.

Insert an HTML element for the token, then switch the insertion mode to "in caption".

↪ **A start tag whose tag name is "colgroup"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in column group".

↳ **A start tag whose tag name is "col"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for a "colgroup" start tag token with no attributes, then switch the insertion mode to "in column group".

Reprocess the current token.

↳ **A start tag whose tag name is one of: "tbody", "tfoot", "thead"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in table body".

↳ **A start tag whose tag name is one of: "td", "th", "tr"**

Clear the stack back to a table context. (See below.)

Insert an HTML element for a "tbody" start tag token with no attributes, then switch the insertion mode to "in table body".

Reprocess the current token.

↳ **A start tag whose tag name is "table"**

Parse error.

If the stack of open elements does not have a `table` element in table scope, ignore the token.

Otherwise:

Pop elements from this stack until a `table` element has been popped from the stack.

Reset the insertion mode appropriately.

Reprocess the token.

↳ **An end tag whose tag name is "table"**

If the stack of open elements does not have a `table` element in table scope, this is a parse error; ignore the token.

Otherwise:

Pop elements from this stack until a `table` element has been popped from the stack.

Reset the insertion mode appropriately.

↳ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↳ **A start tag whose tag name is one of: "style", "script", "template"**

↳ **An end tag whose tag name is "template"**

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

↳ **A start tag whose tag name is "input"**

If the token does not have an attribute with the name "type", or if it does, but that attribute's value is not an ASCII case-insensitive match for the string "`hidden`", then: act as described in the "anything else" entry below.

Otherwise:

Parse error.

Insert an HTML element for the token.

Pop that `input` element off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↪ **A start tag whose tag name is "form"**

Parse error.

If there is a `template` element on the stack of open elements, or if the `form` element pointer is not null, ignore the token.

Otherwise:

Insert an HTML element for the token, and set the `form` element pointer to point to the element created.

Pop that `form` element off the stack of open elements.

↪ **An end-of-file token**

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

↪ **Anything else**

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

When the steps above require the UA to **clear the stack back to a table context**, it means that the UA must, while the current node is not a `table`, `template`, or `html` element, pop elements from the stack of open elements.

Note *The current node being an `html` element after this process is a fragment case.*

12.2.5.4.10 The "in table text" insertion mode

When the user agent is to apply the rules for the "in table text" insertion mode, the user agent must handle the token as follows:

↪ **A character token that is U+0000 NULL**

Parse error. Ignore the token.

↪ **Any other character token**

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

↪ **Anything else**

If any of the tokens in the *pending table character tokens* list are character tokens that are not space characters, then reprocess the character tokens in the *pending table character tokens* list using the rules given in the "anything else" entry in the "in table" insertion mode.

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

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

12.2.5.4.11 The "in caption" insertion mode

When the user agent is to apply the rules for the "in caption" insertion mode, the user agent must handle the token as follows:

↪ **An end tag whose tag name is "caption"**

If the stack of open elements does not have a `caption` element in table scope, this is a parse error; ignore the token. (fragment case)

Otherwise:

Generate implied end tags.

Now, if the current node is not a `caption` element, then this is a parse error.

Pop elements from this stack until a `caption` element has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in table".

↪ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

↪ **An end tag whose tag name is "table"**

If the stack of open elements does not have a `caption` element in table scope, this is a parse error; ignore the token. (fragment case)

Otherwise:

Generate implied end tags.

Now, if the current node is not a `caption` element, then this is a parse error.

Pop elements from this stack until a `caption` element has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in table".

Reprocess the token.

↪ **An end tag whose tag name is one of: "body", "col", "colgroup", "html", "tbody", "td", "tfoot", "th", "thead", "tr"**

Parse error. Ignore the token.

↪ **Anything else**

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

12.2.5.4.12 The "in column group" insertion mode

When the user agent is to apply the rules for the "in column group" insertion mode, the user agent must handle the token as follows:

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

Insert the character.

↪ **A comment token**

Insert a comment.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

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

↪ **A start tag whose tag name is "col"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↳ **An end tag whose tag name is "colgroup"**

If the current node is not a `colgroup` element, then this is a parse error; ignore the token.

Otherwise, pop the current node from the stack of open elements. Switch the insertion mode to "in table".

↳ **An end tag whose tag name is "col"**

Parse error. Ignore the token.

↳ **A start tag whose tag name is "template"**

↳ **An end tag whose tag name is "template"**

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

↳ **An end-of-file token**

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

↳ **Anything else**

If the current node is not a `colgroup` element, then this is a parse error; ignore the token.

Otherwise, pop the current node from the stack of open elements.

Switch the insertion mode to "in table".

Reprocess the token.

12.2.5.4.13 The "in table body" insertion mode

When the user agent is to apply the rules for the "in table body" insertion mode, the user agent must handle the token as follows:

↳ **A start tag whose tag name is "tr"**

Clear the stack back to a table body context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in row".

↳ **A start tag whose tag name is one of: "th", "td"**

Parse error.

Clear the stack back to a table body context. (See below.)

Insert an HTML element for a "tr" start tag token with no attributes, then switch the insertion mode to "in row".

Reprocess the current token.

↳ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

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

Otherwise:

Clear the stack back to a table body context. (See below.)

Pop the current node from the stack of open elements. Switch the insertion mode to "in table".

↳ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead"**

↳ **An end tag whose tag name is "table"**

If the stack of open elements does not have a `tbody`, `thead`, or `tfoot` element in table scope, this is a parse error; ignore the token.

Otherwise:

Clear the stack back to a table body context. (See below.)

Pop the current node from the stack of open elements. Switch the insertion mode to "in table".

Reprocess the token.

↳ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "td", "th", "tr"**

Parse error. Ignore the token.

↳ **Anything else**

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

When the steps above require the UA to **clear the stack back to a table body context**, it means that the UA must, while the current node is not a `tbody`, `tfoot`, `thead`, `template`, or `html` element, pop elements from the stack of open elements.

Note *The current node being an `html` element after this process is a fragment case.*

12.2.5.4.14 The "in row" insertion mode

When the user agent is to apply the rules for the "in row" insertion mode, the user agent must handle the token as follows:

↳ **A start tag whose tag name is one of: "th", "td"**

Clear the stack back to a table row context. (See below.)

Insert an HTML element for the token, then switch the insertion mode to "in cell".

Insert a marker at the end of the list of active formatting elements.

↳ **An end tag whose tag name is "tr"**

If the stack of open elements does not have a `tr` element in table scope, this is a parse error; ignore the token.

Otherwise:

Clear the stack back to a table row context. (See below.)

Pop the current node (which will be a `tr` element) from the stack of open elements. Switch the insertion mode to "in table body".

↳ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "tfoot", "thead", "tr"**

↳ **An end tag whose tag name is "table"**

If the stack of open elements does not have a `tr` element in table scope, this is a parse error; ignore the token.

Otherwise:

Clear the stack back to a table row context. (See below.)

Pop the current node (which will be a `tr` element) from the stack of open elements. Switch the insertion mode to "in table body".

Reprocess the token.

↳ **An end tag whose tag name is one of: "tbody", "tfoot", "thead"**

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

If the stack of open elements does not have a `tr` element in table scope, ignore the token.

Otherwise:

Clear the stack back to a table row context. (See below.)

Pop the current node (which will be a `tr` element) from the stack of open elements. Switch the insertion mode to "in table body".

Reprocess the token.

↳ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html", "td", "th"**

Parse error. Ignore the token.

↳ **Anything else**

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

When the steps above require the UA to **clear the stack back to a table row context**, it means that the UA must, while the current node is not a `tr`, `template`, or `html` element, pop elements from the stack of open elements.

Note *The current node being an `html` element after this process is a fragment case.*

12.2.5.4.15 The "in cell" insertion mode

When the user agent is to apply the rules for the "in cell" insertion mode, the user agent must handle the token as follows:

↳ **An end tag whose tag name is one of: "td", "th"**

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

Otherwise:

Generate implied end tags.

Now, if the current node is not an HTML element with the same tag name as the token, then this is a parse error.

Pop elements from the stack of open elements stack until an HTML element with the same tag name as the token has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Switch the insertion mode to "in row".

↳ **A start tag whose tag name is one of: "caption", "col", "colgroup", "tbody", "td", "tfoot", "th", "thead", "tr"**

If the stack of open elements does *not* have a `td` or `th` element in table scope, then this is a parse error; ignore the token. (fragment case)

Otherwise, close the cell (see below) and reprocess the token.

↳ **An end tag whose tag name is one of: "body", "caption", "col", "colgroup", "html"**

Parse error. Ignore the token.

↳ **An end tag whose tag name is one of: "table", "tbody", "tfoot", "thead", "tr"**

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

Otherwise, close the cell (see below) and reprocess the token.

↳ **Anything else**

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

Where the steps above say to **close the cell**, they mean to run the following algorithm:

1. Generate implied end tags.
2. If the current node is not now a `td` element or a `th` element, then this is a parse error.
3. Pop elements from the stack of open elements stack until a `td` element or a `th` element has been popped from the stack.
4. Clear the list of active formatting elements up to the last marker.
5. Switch the insertion mode to "in row".

Note *The stack of open elements cannot have both a `td` and a `th` element in table scope at the same time, nor can it have neither when the close the cell algorithm is invoked.*

12.2.5.4.16 The "in select" insertion mode

When the user agent is to apply the rules for the "in select" insertion mode, the user agent must handle the token as follows:

↪ **A character token that is U+0000 NULL**

Parse error. Ignore the token.

↪ **Any other character token**

Insert the token's character.

↪ **A comment token**

Insert a comment.

↪ **A DOCTYPE token**

Parse error. Ignore the token.

↪ **A start tag whose tag name is "html"**

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

↪ **A start tag whose tag name is "option"**

If the current node is an `option` element, pop that node from the stack of open elements.

Insert an HTML element for the token.

↪ **A start tag whose tag name is "optgroup"**

If the current node is an `option` element, pop that node from the stack of open elements.

If the current node is an `optgroup` element, pop that node from the stack of open elements.

Insert an HTML element for the token.

↪ **An end tag whose tag name is "optgroup"**

First, if the current node is an `option` element, and the node immediately before it in the stack of open elements is an `optgroup` element, then pop the current node from the stack of open elements.

If the current node is an `optgroup` element, then pop that node from the stack of open elements. Otherwise, this is a parse error; ignore the token.

↪ **An end tag whose tag name is "option"**

If the current node is an `option` element, then pop that node from the stack of open elements. Otherwise, this is a parse error; ignore the token.

↪ **An end tag whose tag name is "select"**

If the stack of open elements does not have a `select` element in select scope, this is a parse error; ignore the token. (fragment case)

Otherwise:

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

↪ **A start tag whose tag name is "select"**

Parse error.

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

Note *It just gets treated like an end tag.*

↪ **A start tag whose tag name is one of: "input", "keygen", "textarea"**

Parse error.

If the stack of open elements does not have a `select` element in select scope, ignore the token. (fragment case)

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

Reprocess the token.

↪ **A start tag whose tag name is one of: "script", "template"**

↪ **An end tag whose tag name is "template"**

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

↪ **An end-of-file token**

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

↪ **Anything else**

Parse error. Ignore the token.

12.2.5.4.17 The "in select in table" insertion mode

When the user agent is to apply the rules for the "in select in table" insertion mode, the user agent must handle the token as follows:

↪ **A start tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**

Parse error.

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

Reprocess the token.

↪ **An end tag whose tag name is one of: "caption", "table", "tbody", "tfoot", "thead", "tr", "td", "th"**

Parse error.

If the stack of open elements does not have an element in table scope that is an HTML element and with the same tag name as that of the token, then ignore the token.

Otherwise:

Pop elements from the stack of open elements until a `select` element has been popped from the stack.

Reset the insertion mode appropriately.

Reprocess the token.

↳ **Anything else**

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

12.2.5.4.18 The "in template" insertion mode

When the user agent is to apply the rules for the "in template" insertion mode, the user agent must handle the token as follows:

↳ **A character token**

↳ **A comment token**

↳ **A DOCTYPE token**

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

↳ **A start tag whose tag name is one of: "base", "basefont", "bgsound", "link", "meta", "noframes", "script", "style", "template", "title"**

↳ **An end tag whose tag name is "template"**

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

↳ **A start tag whose tag name is one of: "caption", "colgroup", "tbody", "tfoot", "thead"**

Pop the current template insertion mode off the stack of template insertion modes.

Push "in table" onto the stack of template insertion modes so that it is the new current template insertion mode.

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

↳ **A start tag whose tag name is "col"**

Pop the current template insertion mode off the stack of template insertion modes.

Push "in column group" onto the stack of template insertion modes so that it is the new current template insertion mode.

Switch the insertion mode to "in column group", and reprocess the token.

↳ **A start tag whose tag name is "tr"**

Pop the current template insertion mode off the stack of template insertion modes.

Push "in table body" onto the stack of template insertion modes so that it is the new current template insertion mode.

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

↳ **A start tag whose tag name is one of: "td", "th"**

Pop the current template insertion mode off the stack of template insertion modes.

Push "in row" onto the stack of template insertion modes so that it is the new current template insertion mode.

Switch the insertion mode to "in row", and reprocess the token.

↳ **Any other start tag**

Pop the current template insertion mode off the stack of template insertion modes.

Push "in body" onto the stack of template insertion modes so that it is the new current template insertion mode.

Switch the insertion mode to "in body", and reprocess the token.

↳ **Any other end tag**

Parse error. Ignore the token.

↳ **An end-of-file token**

If there is no `template` element on the stack of open elements, then stop parsing. (fragment case)

Otherwise, this is a parse error.

Pop elements from the stack of open elements until a `template` element has been popped from the stack.

Clear the list of active formatting elements up to the last marker.

Pop the current template insertion mode off the stack of template insertion modes.

Reset the insertion mode appropriately.

Reprocess the token.

12.2.5.4.19 The "after body" insertion mode

When the user agent is to apply the rules for the "after body" insertion mode, the user agent must handle the token as follows:

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

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

↳ **A comment token**

Insert a comment as the last child of the first element in the stack of open elements (the `html` element).

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A start tag whose tag name is "html"**

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

↳ **An end tag whose tag name is "html"**

If the parser was originally created as part of the HTML fragment parsing algorithm, this is a parse error; ignore the token. (fragment case)

Otherwise, switch the insertion mode to "after after body".

↳ **An end-of-file token**

Stop parsing.

↳ **Anything else**

Parse error. Switch the insertion mode to "in body" and reprocess the token.

12.2.5.4.20 The "in frameset" insertion mode

When the user agent is to apply the rules for the "in frameset" insertion mode, the user agent must handle the token as follows:

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

Insert the character.

↳ **A comment token**

Insert a comment.

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A start tag whose tag name is "html"**

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

↳ **A start tag whose tag name is "frameset"**

Insert an HTML element for the token.

↳ **An end tag whose tag name is "frameset"**

If the current node is the root `html` element, then this is a parse error; ignore the token. (fragment case)

Otherwise, pop the current node from the stack of open elements.

If the parser was *not* originally created as part of the HTML fragment parsing algorithm (fragment case), and the current node is no longer a `frameset` element, then switch the insertion mode to "after frameset".

↳ **A start tag whose tag name is "frame"**

Insert an HTML element for the token. Immediately pop the current node off the stack of open elements.

Acknowledge the token's *self-closing flag*, if it is set.

↳ **A start tag whose tag name is "noframes"**

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

↳ **An end-of-file token**

If the current node is not the root `html` element, then this is a parse error.

Note *The current node can only be the root `html` element in the fragment case.*

Stop parsing.

↳ **Anything else**

Parse error. Ignore the token.

12.2.5.4.21 The "after frameset" insertion mode

When the user agent is to apply the rules for the "after frameset" insertion mode, the user agent must handle the token as follows:

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

Insert the character.

↳ **A comment token**

Insert a comment.

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A start tag whose tag name is "html"**

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

↳ **An end tag whose tag name is "html"**

Switch the insertion mode to "after after frameset".

↳ **A start tag whose tag name is "noframes"**

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

↳ **An end-of-file token**

Stop parsing.

↳ **Anything else**

Parse error. Ignore the token.

12.2.5.4.22 The "after after body" insertion mode

When the user agent is to apply the rules for the "after after body" insertion mode, the user agent must handle the token as follows:

↳ **A comment token**

Insert a comment as the last child of the `Document` object.

↳ **A DOCTYPE token**

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

↳ **A start tag whose tag name is "html"**

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

↳ **An end-of-file token**

Stop parsing.

↳ **Anything else**

Parse error. Switch the insertion mode to "in body" and reprocess the token.

12.2.5.4.23 The "after after frameset" insertion mode

When the user agent is to apply the rules for the "after after frameset" insertion mode, the user agent must handle the token as follows:

↳ **A comment token**

Insert a comment as the last child of the `Document` object.

↳ **A DOCTYPE token**

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

↳ **A start tag whose tag name is "html"**

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

↳ **An end-of-file token**

Stop parsing.

↳ **A start tag whose tag name is "noframes"**

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

↳ **Anything else**

Parse error. Ignore the token.

12.2.5.5 The rules for parsing tokens in foreign content

When the user agent is to apply the rules for parsing tokens in foreign content, the user agent must handle the token as follows:

↳ **A character token that is U+0000 NULL**

Parse error. Insert a U+FFFD REPLACEMENT CHARACTER character.

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

Insert the token's character.

↳ **Any other character token**

Insert the token's character.

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

↳ **A comment token**

Insert a comment.

↳ **A DOCTYPE token**

Parse error. Ignore the token.

↳ **A start tag whose tag name is one of: "b", "big", "blockquote", "body", "br", "center", "code", "dd", "div", "dl", "dt", "em", "embed", "h1", "h2", "h3", "h4", "h5", "h6", "head", "hr", "i", "img", "li", "listing", "menu", "meta", "nobr", "ol", "p", "pre", "ruby", "s", "small", "span", "strong", "strike", "sub", "sup", "table", "tt", "u", "ul", "var"**

↳ **A start tag whose tag name is "font", if the token has any attributes named "color", "face", or "size"**

Parse error.

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

Otherwise:

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

Then, reprocess the token.

↳ **Any other start tag**

If the adjusted current node is an element in the MathML namespace, adjust MathML attributes for the token. (This fixes the case of MathML attributes that are not all lowercase.)

If the adjusted current node is an element in the SVG namespace, and the token's tag name is one of the ones in the first column of the following table, change the tag name to the name given in the corresponding cell in the second column. (This fixes the case of SVG elements that are not all lowercase.)

Tag name	Element name
altglyph	altGlyph
altglyphdef	altGlyphDef
altglyphitem	altGlyphItem
animatecolor	animateColor

animatemotion	animateMotion
animatetransform	animateTransform
clippath	clipPath
feblend	feBlend
fecolormatrix	feColorMatrix
fecomponenttransfer	feComponentTransfer
fecomposite	feComposite
feconvolvematrix	feConvolveMatrix
fediffuselighting	feDiffuseLighting
fedisplacementmap	feDisplacementMap
fedistantlight	feDistantLight
fedropshadow	feDropShadow
feflood	feFlood
fefunca	feFuncA
fefuncb	feFuncB
fefuncg	feFuncG
fefuncr	feFuncR
fegaussianblur	feGaussianBlur
feimage	feImage
femerge	feMerge
femergenode	feMergeNode
femorphology	feMorphology
feoffset	feOffset
fepointlight	fePointLight
fespecularlighting	feSpecularLighting
fespotlight	feSpotLight
fetile	feTile
feturbulence	feTurbulence
foreignobject	foreignObject
glyphref	glyphRef
lineargradient	linearGradient
radialgradient	radialGradient
textpath	textPath

If the adjusted current node is an element in the SVG namespace, adjust SVG attributes for the token. (This fixes the case of SVG attributes that are not all lowercase.)

Adjust foreign attributes for the token. (This fixes the use of namespaced attributes, in particular XLink in SVG.)

Insert a foreign element for the token, in the same namespace as the adjusted current node.

If the token has its *self-closing flag* set, then run the appropriate steps from the following list:

↪ **If the token's tag name is "script"**

Acknowledge the token's *self-closing flag*, and then act as described in the steps for a "script" end tag below.

↪ **Otherwise**

Pop the current node off the stack of open elements and acknowledge the token's *self-closing flag*.

↪ **An end tag whose tag name is "script", if the current node is a `script` element in the SVG namespace**

Pop the current node off the stack of open elements.

Let the *old insertion point* have the same value as the current insertion point. Let the insertion point be just

before the next input character.

Increment the parser's script nesting level by one. Set the parser pause flag to true.

Process the `script` element according to the SVG rules, if the user agent supports SVG. [SVG]

Note *Even if this causes new characters to be inserted into the tokenizer, the parser will not be executed reentrantly, since the parser pause flag is true.*

Decrement the parser's script nesting level by one. If the parser's script nesting level is zero, then set the parser pause flag to false.

Let the insertion point have the value of the *old insertion point*. (In other words, restore the insertion point to its previous value. This value might be the "undefined" value.)

↪ **Any other end tag**

Run these steps:

1. Initialise *node* to be the current node (the bottommost node of the stack).
2. If *node*'s tag name, converted to ASCII lowercase, is not the same as the tag name of the token, then this is a parse error.
3. *Loop*: If *node* is the topmost element in the stack of open elements, abort these steps. (fragment case)
4. If *node*'s tag name, converted to ASCII lowercase, is the same as the tag name of the token, pop elements from the stack of open elements until *node* has been popped from the stack, and then abort these steps.
5. Set *node* to the previous entry in the stack of open elements.
6. If *node* is not an element in the HTML namespace, return to the step labeled *loop*.
7. Otherwise, process the token according to the rules given in the section corresponding to the current insertion mode in HTML content.