How to cope with incorrect HTML

Cand.Scient thesis
by
Dagfinn Parnas

Department of Informatics
University of Bergen

4th december 2001

Abstract

This thesis considers problems related to incorrect HTML. We will propose a method of parsing both valid and invalid HTML documents. The major emphasis will be on incorrect HTML and how we in a systematic manner can correct invalid HTML documents while still being compliant with SGML.

These issues are of prime importance when developing and maintaining browser software.

We therefore also describe work on a automatic regression testing system for IceStorm Browser, a Java browser originally developed by IceSoft AS. This work provides a method of testing new versions of the browser against former versions.

Acknowledgements

I would like to thank my supervisor, Professor Petter Bjørstad, for giving me a lot of freedom, but still being there when I needed advice. The stay in Denver really helped my motivation.

My regards goes out to all the people at IceSoft AS. It is a shame that IceSoft AS will no longer exist.

I would also like to thank all my friends, especially the ones in Hero and on the "bridge", without you Bergen would not be the same. Warm regards to my parents for always being there and supporting my choices.

Most of all I would like thank Frances, for all her love and support. Without you, life would be so much less.

Contents

1. Introduction

Ever since the birth of the first browser, error-correcting has been an essential feature of it. Unlike most other markup- and programming-languages, HTML, as interpreted by browsers, is incredibly flexible and has a loose syntax. Browsers are infamous for attempting to do what the author of an HTML document wants to do, instead of doing what the author actually specified in the HTML document. The result of this is that most of the HTML documents on the Internet are invalid according to the HTML standard published by the World Wide Web Consortium.

Since throughout the history of HTML most HTML documents have been invalid, new browsers had to, and still must, simulate the error-correcting methods of the major browsers of the past.

The trend the last few years have been to stabilize the HTML standard on the web as it is now, and not add any new error-correcting features. The latest version of the two leading browsers, Internet Explorer and Netscape Navigator, are in fact rather similar in this area.

This thesis will attempt to find out how big a problem invalid HTML is. We will identify the methods the leading browsers use for error-correcting, and show that the same process can be simulated in an orderly fashion which is valid according to SGML rules.

We will also look closer on an automatic regression testing system developed for Icesoft AS, which tests the functionality of new versions of the IceStorm Browser.

2. HTML

2.1 HTML, a markup language

As you might already know, HTML is an abbreviation for Hypertext Markup Language. In our context the word 'markup' comes from the publishing and printing business, where it means the instructions for the typesetter on a manuscript to control the alignment and size of the printed text.

2.1.1 Markup languages

A markup language is a set of markup conventions used together for encoding text. A document encoded in a markup language contains both markup and the text to be encoded.

Almost all electronic documents are written in a markup language of some form. This thesis, for example, is written in LaTeX, but documents written in Microsoft Word or using RTF(Rich Text Format) are also written in a markup language.

The most common example of a document not using a specific markup language is a plain text file, although one might argue that the placement of punctuation marks and line feeds provide a markup language of their own, dividing the text into lines and paragraphs.

There are two basic reasons why we format text using a markup language:

1. To provide an exact method of displaying a document

2. To provide the logical structure of the document.

The first is called procedural markup, and the second is called descriptive or declarative markup.

Procedural markup is categorized by the existence of markup conventions telling exactly how to display the document. The names of procedural markup element types are usually verbs; for example :

- **changeFont** : Change the font to a specific size, face and colour

- **doIndent** : Make an indent with a width specified in some form of measurement.

Declarative markup, on the other hand, contains markup element types that provide structure to the document. The names of these element types are usually nouns, such as:

- **paragraph** : A paragraph in the text
- **title** : The title of the document.

Procedural markup is more efficient because it allows the computer to follow the supplied instructions without any additional interpretation. Declarative markup on the other hand needs an extra step in order for a document to be displayed.

The strength of declarative markup is that it allows the document to be stored in a single form, independently of how it is going to be presented. This means that a document might be displayed in many different ways and on many different platforms. It is also easier to change or analyse a document because structural information is present, and the extra complexity of how to present the document is not part of the document.

At the end of the 1960s an increasing number of documents were transferred into electronic form. At this time, most of the electronic file formats were procedural, but to allow easier manipulation and cross-platform compatibility the need for a declarative markup language was evident. A single markup language could not contain the logical structure of all types of data that may exist, so instead a language for creating markup languages was developed.

Such a meta-language was developed in 1969 by Charles Goldfarb, Edward Mosher and Raymond Lori during an IBM research project. This language was called first Generalized Markup Language(GML), but was published in 1980 as a working draft by ANSI[1] under the name Standard Generalized Markup Language (SGML). It became ISO standard 8879[ISO86] in 1986.

SGML gives the constructs to design markup languages for most purposes. The most central part of a markup language defined through SGML is the Document Type Definition (DTD). In essence the DTD contains a definition of which markup element types exist and how these element types relate to each other.

There often seems to be a bit confusion about the difference between the terms tag, element and element type. Let us clarify how SGML defines these terms before we continue.

A tag is a physical "marker" present in a document which indicates the presence of an element. An element is a logical representation of everything from a start-tag to a matching end-tag. An element may have attributes (which are defined in the start-tag), and can contain other elements (delimited by their own tags) or text content. An element type, on the other hand, is the overall class to which a particular element belongs to. Therefore an element is an

[1]American National Standards Institute.

instance of an element type. To find out which element type an element is an instance of, we look at its generic identifier, which is the first word after the start of the start- or end-tag.

If we in this thesis write a generic identifier followed by the word element, we mean an element which is an instance of the element type which the generic identifier corresponds to. For example: CAPTION element refers to an element which is an instance of the CAPTION element type.

A tag does not always have to be present in a document in order for an element to occur or end. SGML allows a feature called markup minimization, which allows us to omit a start- and/or end-tag of an element if the tag is unambiguously implied from the context.

The relationship between the elements in an SGML document is a parent-child relationship. This allows us to view an SGML document through the tree representation of it. A text node is a node in this tree which only contains text and cannot have any child element. It is therefore always a leaf in the tree representation of an SGML document. We also know that if an element B is between the actual or implied start- and end-tag of element A, we are guaranteed that B is in A's subtree.

2.1.2 Hypertext

"Hypertext is the presentation of information as a linked network of nodes which readers are free to navigate in a non-linear fashion. It allows for multiple authors, a blurring of the author and reader functions, extended works with diffuse boundaries and multiple reading paths."

— *Ted Nelson*

The most important part in the definition of hypertext above is the word non-linear. A book is not hypertext, in the sense that it is expected to be read from front to back. Hypertext has no fixed reading path, and the user can discontinue reading one document at certain points in order to consult other related material.

2.1.3 The Internet

The Internet started in the 1960s when the ARPAnet[2] was developed in the USA. The goal of the ARPAnet was to provide a robust network of computers, that would sustain stability even if many computers lost their network connection. Until the early 1990s this network consisted of mostly American defense contractors and international academic institutions.

The ARPAnet grew gradually, but it was not until 1991 it really became huge and became what we today know as the Internet. There were four major prerequisites to the evolvement of the Internet.

[2]Advanced Research and Projects Agency network of the Department of Defense.

1. A telecommunication network with possibilities of sending data
2. The development of protocols and standards which allowed machines at remotely different locations and with different operating systems to communicate with each other. The most influential of these are TCP/IP, http and HTML
3. Software which use the protocols and standards developed to communicate through the telecommunication network. In our case this would be the first HTML browser
4. A critical mass of users that had the possibility of and interest in using such software as described in point 3.

It was CERN[3] which started the Word-Wide Web (WWW) in 1991, as well as developed the HTML data format and browser software. The WWW provides hypertext linking which gives access to documents located on another server connected to the Internet at the click of a mouse.

2.1.4 Browsers and user-agents

There are two main types of browser, visual(text-only and graphical) and non-visual(audio, Braille). When we use the word browser in this thesis, we refer to visual browsers. If we wish to refer to non-visual browsers, we will do this explicitly.

In our context, when we refer to a user-agent we mean any device that interprets HTML documents. The most common user-agents are browsers, search robots, proxies and HTML editors.

2.2 The HTML Standard

The first proposition of HTML came in 1991 and contained only a handful of element types. Although the standard was loosely based on SGML, this version was not formalized through an SGML DTD. It differed from SGML in the interpretation of some tags. For example the P start-tag indicated the end of a paragraph to a user-agent in the first HTML version, instead of the beginning of a P element as it would have in SGML.

In version 2.0 these differences were corrected, and HTML was formalized through an SGML DTD.

Since then new versions have been proposed and recommended by the World Wide Web Consortium(W3C), the last being HTML 4.01[4].

HTML defines three different DTDs; the loose(also called the transitional), the strict and the frameset DTD. The difference between the strict and the

[3] Conseil European pour la Recherché Nucleaire (European Laboratory for Particle Physics; Geneva, Switzerland).

[4] XHTML has not been taken into account here. See 7.4 for more information about XHTML.

loose DTD, is that the strict DTD does not contain certain deprecated element types present in the loose DTD, whose semantics specify how text is formatted, or how the layout of a document should be. These element types have been removed in order to make the language more declarative. The frameset DTD is identical to the loose DTD except that the BODY element type is replaced with the FRAMESET element type to allow usage of frames.

HTML has also been published as ISO standard 15445[PA00]. This version of HTML is stricter than the W3C's HTML 4.01 using the strict DTD. Future references to HTML will always refer to W3C's HTML 4.01 if not otherwise specified.

2.2.1 Understanding DTDs

To truly understand how we can parse HTML documents we need to take a closer look at the HTML DTD. The most important part of this section is to understand how SGML constructs define HTML as a markup language. Therefore it is not especially important which DTD we choose to look at. I have chosen the strict DTD as the basis of the examples because it is the one recommended by W3C, and the element types which it contains will be used in future versions of HTML.

The strict DTD for HTML is available through the WWW at http://www.w3.org/TR/html4/strict.dtd.

Let us first look at the important constructs SGML defines that are used in the HTML DTD.

Entity: `<!ENTITY % nameofentity ''value of entity''>`
or
`<!ENTITY & nameofentity ''value of entity''>`

```
<!ENTITY % heading ''H1 | H2 | H3 | H4 | H5 | H6''>
<!ENTITY & lt ''<''>
```

Example 2.1: ENTITY examples from the HTML strict DTD.

The Example 2.1 shows the heading entity and the entity for using the character <. These entities are in the strict DTD.

An entity is used to make shortcuts to a text that will appear several places in the DTD or in a document. Using the % sign will make the entity available only in the DTD, while using the & sign will make it available in documents using this DTD as well. To use the entity we normally use the notation %nameofentity; or &nameofentity; depending on the entity type. When processed the entity will be treated as the value of the entity.

Element: `<!ELEMENT myelement - O (mysubelement) >`

```
<!ELEMENT HTML O O (HEAD,BODY)>
```

Example 2.2: ELEMENT example from the HTML strict DTD.

The Example 2.2 shows how the HTML element type is defined in the strict DTD.

All definitions of an element type start with <!ELEMENT and end with character >.Between the start and end delimiter we have, in the following order:

1. The name of the element type
2. Two characters with a space between them. This defines whether the start- and end-tags are required for an instance of this element type[5]. If these characters are - -, both the start- and end-tag are necessary. If they are - O[6], the end-tag is optional[7].
3. Which types of elements are allowed to be nested inside an element of this element type; often referred to as the element type's content model[8]. We use a special syntax to describe the content model. In this syntax the A and B characters represent an A element and a B element respectively. This syntax is:

A	A must occur, one time only
A+	A must occur, one or more times
A?	A may occur, zero or one time
A*	A may occur zero or more times
+(A)	A may occur under elements of this element type. This property is inherited by all elements in the subtree of an element of this element type
-(A)	A must not occur under elements of this element type. This property is inherited by all elements in the subtree of an element of this element type
A|B	Either A or B must occur, not both
A,B	Both A and B must occur, in that order
A&B	Both A and B must occur, in any order
EMPTY	No nested elements allowed.

If we have a finite set of symbols V, the content model E can be described formally as a language L(E) of words over the symbols in V. This language can be defined inductively by using the following rules[BK93]:

[5]This is only used if tag minimization is turned on in SGML, as it is in HTML.

[6]The letter O, and not the number 0.

[7]And if the content model is empty, the end-tag is forbidden.

[8]Each element has also got its own content model which specify which elements are contained within this element, and which elements can be contained.

Inductive definition of content models

$L(a) = a\ for\ a \in V$
$L(F?) = L(F) \cup \epsilon$, where ϵ denotes the empty word
$L(F|G) = L(F) \cup L(G)$
$L(F,G) = L(F)L(G) = \{uv|u \in L(F), v \in L(G)\}$
$L(F\&G) = L(FG|GF) = \{uv|u \in L(F), v \in L(G)$
$\quad or\ u \in L(G)\ and\ v \in L(F)\}$
$L(F^*) = \{v_1, \ldots, v_n | n \geq 0, v_1, \ldots, v_n \in L(F)\}$
$L(F+) = \{v_1, \ldots, v_n | n \geq 1, v_1, \ldots, v_n \in L(F)\}$

Let us now analyse one of the more complex content models in the HTML DTD. The element type we will be looking at is TABLE. The content model for the element type TABLE is :

```
(CAPTION?, (COL*|COLGROUP*), THEAD?, TFOOT?, TBODY+)
```

The informal interpretation of this would be: You will have, in the following order, an optional CAPTION element, either zero or more COL elements or zero or more COLGROUP elements, an optional THEAD element, an optional TFOOT element and one or more TBODY elements.

For A formal method we use the inductive definition of content models in order to produce a reduction system.

Reduction of the element type TABLE's content model. Symbols written in uppercase.

$$L(TABLE) = L(CAPTION?, (COL^*|COLGROUP^*), THEAD?, TFOOT?, TBODY+)$$
$$= L(CAPTION?)L(COL^*|COLGROUP^*)L(THEAD?)L(TFOOT?)L(TBODY+)$$
$$= L(CAPTION?)L(COL^*) \cup L(COLGROUP^*)L(THEAD?)L(TFOOT?)L(TBODY+)$$
$$= (CAPTION \cup \epsilon)((COL_1, \ldots, COL_n) \cup (COLGROUP_1, \ldots, COLGROUP_m))$$
$$(THEAD \cup \epsilon)(TFOOT \cup \epsilon)(TBODY_1, \ldots, TBODY_o)$$
where $n \geq 0, m \geq 0, o \geq 1$

By combining the different terms we get the following 16 valid combinations(words).

$CAPTION, COL_1, \ldots, COL_n, THEAD, TFOOT, TBODY_1, \ldots, TBODY_o$
$CAPTION, COL_1, \ldots, COL_n, \epsilon, TFOOT, TBODY_1, \ldots, TBODY_o$
$CAPTION, COL_1, \ldots, COL_n, THEAD, \epsilon, TBODY_1, \ldots, TBODY_o$
$CAPTION, COL_1, \ldots, COL_n, \epsilon, \epsilon, TBODY_1, \ldots, TBODY_o$
$CAPTION, COLGROUP_1, \ldots, COLGROUP_m, THEAD, TFOOT, TBODY_1, \ldots, TBODY_o$
$CAPTION, COLGROUP_1, \ldots, COLGROUP_m, \epsilon, TFOOT, TBODY_1, \ldots, TBODY_o$
$CAPTION, COLGROUP_1, \ldots, COLGROUP_m, THEAD, \epsilon, TBODY_1, \ldots, TBODY_o$
$CAPTION, COLGROUP_1, \ldots, COLGROUP_m, \epsilon, \epsilon, TBODY_1, \ldots, TBODY_o$

$\epsilon, COL_1, \ldots, COL_n, THEAD, TFOOT, TBODY_1, \ldots, TBODY_o$
$\epsilon, COL_1, \ldots, COL_n, \epsilon, TFOOT, TBODY_1, \ldots, TBODY_o$
$\epsilon, COL_1, \ldots, COL_n, THEAD, \epsilon, TBODY_1, \ldots, TBODY_o$
$\epsilon, COL_1, \ldots, COL_n, \epsilon, \epsilon, TBODY_1, \ldots, TBODY_o$
$\epsilon, COLGROUP_1, \ldots, COLGROUP_m, THEAD, TFOOT, TBODY_1, \ldots, TBODY_o$
$\epsilon, COLGROUP_1, \ldots, COLGROUP_m, \epsilon, TFOOT, TBODY_1, \ldots, TBODY_o$
$\epsilon, COLGROUP_1, \ldots, COLGROUP_m, THEAD, \epsilon, TBODY_1, \ldots, TBODY_o$
$\epsilon, COLGROUP_1, \ldots, COLGROUP_m, \epsilon, \epsilon, TBODY_1, \ldots, TBODY_o$
where $n \geq 0, m \geq 0, o \geq 1$

From the valid words you might draw the conclusion that you have to have either a col or colgroup element, but since $n \geq 0, m \geq 0$ and the fact that $COL_1, \ldots, COL_0 = \epsilon$ you may choose not to include any of these elements.

The previous example shows that content models can be very complex and difficult for an application to handle. Fortunately, most element types in HTML have defined their content model in such a manner that it is not dependent on a particular order of element types, instead allowing all possible combinations of sequences of element types.

The definition of attributes for the element types is made through the Attlist construct. It uses the following notation:

Attlist: `<!ATTLIST nameof element`
```
    attribute1 attribute-type default-value
    attribute2 attribute-type default-value
    >
```

```
<!ATTLIST IMG
src CDATA #REQUIRED
alt CDATA #REQUIRED
name CDATA #IMPLIED
height CDATA #IMPLIED
width CDATA #IMPLIED
usemap CDATA #IMPLIED
ismap (ismap) #IMPLIED
>
```
Example 2.3: ATTLIST construct from the HTML strict DTD.

In Example 2.3 we see that the IMG element type (representing an image) has 7 attributes[9]. The attributes with a trailing #REQUIRED must be entered in a start-tag of an IMG element, while the attributes with a trailing #IMPLIED means that the user-agent shall supply a default value if no value is present. Most of the attributes are of the type CDATA. This means that they can consist

[9]Some attributes have been stripped from the DTD to reduce the amount of space used.

of valid SGML characters as well as entities[10]. The ismap attribute on the other hand is a boolean attribute that has either the value ismap or none.

2.2.2 Summarizing the HTML DTD

The most interesting feature in the HTML DTD for parsing an HTML document correctly, is each element type's content model and whether or not the start- and/or end-tag can be omitted.

In the Table 2.1 we have summarized the content model and tag properties for each element type in the HTML 4.01 strict DTD.

In this table we have used two entities with element types, which are also defined in the HTML DTD, in order to keep the size down and the readability up. These are the block entity and the inline entity. It is easiest to describe the difference between block element types and inline element types in terms of presentation of instances.

Block elements are usually presented with a break before and after them. Most of them can contain other block elements, and many can contain character data or inline elements. The block entity is defined as:

```
<!ENTITY % block ``P|H1|H2|H3|H4|H5|H6|UL|OL|PRE|DL|DIV|NOSCRIPT
|BLOCKQUOTE|FORM|HR|TABLE|FIELDSET|ADDRESS''>
```

Inline elements are generally presented without any breaks. The most common effect inline elements have is a font change, but there may be no visual distinction at all. Inline elements can contain character data and other inline elements, but never any block elements.

The inline entity is defined as:

```
<!ENTITY % inline
``#PCDATA|TT|I|B|BIG|SMALL|EM|STRONG|DFN|CODE|SAMP|KBD|VAR|CITE
|ABBR|ACRONYM|A|IMG|OBJECT|BR|SCRIPT|MAP|Q|SUB|SUP|SPAN|BDO
|INPUT|SELECT|TEXTAREA|LABEL|BUTTON'' >
```

The symbols used for the start and end-tag properties are

- O: Optional
- -: Required
- F: Forbidden

[10]However, if CDATA is used in the content model, entities should not be replaced. In HTML this never occurs.

Element type	Start tag	End tag	**Content model**
HTML	O	O	HEAD,BODY
HEAD	O	O	(TITLE & BASE?) +(SCRIPT|STYLE|META|LINK|OBJECT)
TITLE	-	-	(#PCDATA)
BASE	-	F	EMPTY
META	-	F	EMPTY
STYLE	-	-	CDATA
SCRIPT	-	-	CDATA
NOSCRIPT	-	-	(%block;)+
BODY	O	O	(%block;|SCRIPT)+ +(INS|DEL)
ADDRESS	-	-	(%inline;)*
DIV	-	-	(%block;|%inline;)*
A	-	-	(%inline;)* - A
MAP	-	-	(%block;) | AREA)+
AREA	-	F	EMPTY
LINK	-	F	EMPTY
IMG	-	F	EMPTY
OBJECT	-	-	(PARAM|%block;|inline;)*
HR	-	F	EMPTY
P	-	F	(%inline;)*
H1,...,H6	-	-	(%inline;)*
PRE	-	F	(%inline;)* -(IMG|OBJECT|SMALL|SUB|SUP)
Q	-	-	(%inline;)*
BLOCKQUOTE	-	-	(%block;|SCRIPT)+
INS	-	-	(%block;| %inline;)*
DEL	-	-	(%block;|%inline;)*
DL	-	-	(DT|DD)+
DT	-	O	(%inline;)*
DD	-	O	(%block;|%inline;)*
OL	-	-	(LI)+
UL	-	-	(LI)+
LI	-	O	(%block;|%inline;)*
FORM	-	-	(%block;|SCRIPT) -(FORM)
LABEL	-	-	(%inline;)* -(LABEL)
INPUT	-	F	EMPTY
SELECT	-	-	(OPTGROUP|OPTION)
OPTGROUP	-	-	(OPTION)+
OPTION	-	O	(#PCDATA)
TEXTAREA	-	-	(#PCDATA)
FIELDSET	-	-	(#PCDATA,LEGEND,(%block;|%inline;)*)
LEGEND	-	-	(%inline;)*

Element type	Start tag	End tag	**Content model**
BUTTON	-	-	(%block;|%inline;)* -(A|INPUT|SELECT|TEXTAREA| LABEL|BUTTON|FORM|FIELDSET)
TABLE	-	-	(CAPTION?,(COL*|COLGROUP*), THEAD?,TFOOT?,TBODY+)
CAPTION	-	-	(%inline;)*
THEAD	-	O	(TR)+
TFOOT	-	O	(TR)+
TBODY	O	O	(TR)+
COLGROUP	-	O	(COL*)
COL	-	F	EMPTY
TR	-	O	(TH|TD)+
TH	-	O	(%block;|%inline;)*
TD	-	O	(%block;|%inline;)*
TT|I|B|BIG|SMALL	-	-	(%inline;)*
EM|STRONG|DFN |CODE|SAMP|KBD|VAR |CITE|ABBR|ACRONYM	-	-	(%inline;)*
SUB	-	-	(%inline;)*
SUP	-	-	(%inline;)*
SPAN	-	-	(%inline;)*
BDO	-	-	(%inline;)*
BR	-	F	EMPTY

Table 2.1: Properties of the element types in HTML 4.01 strict. Shows omittable start- and end-tags, and content model for each element type.

Table 2.1 can also be illustrated using a graph. The graph in Figure 2.1 shows the relationships between the different element types. It does not properly illustrate exclusions, such as an A element can not contain any other A element. All dotted lines shows which element types are part of which entities.

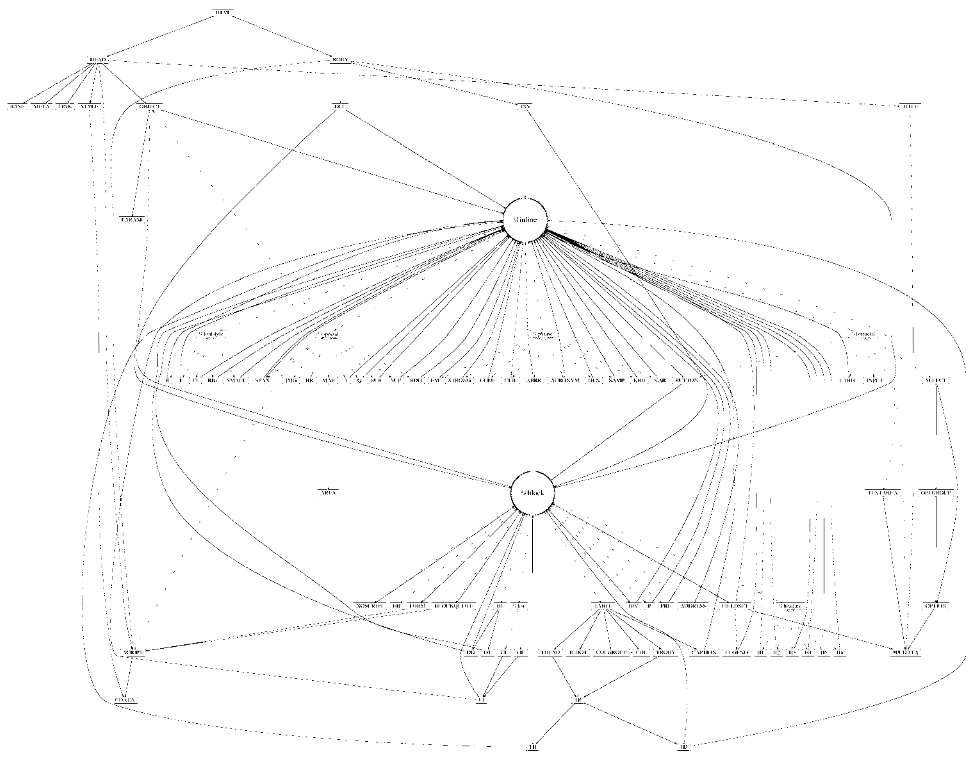


Figure 2.1: Complex graph showing the relationships between the HTML element types.

2.3 Data structure

We need to use an appropriate data structure to contain the HTML document. It is natural to use the W3C proposition Document Object Model(DOM). This model can be used for any structured document, but is specifically intended for use on HTML and XML documents.

The Node interface is the primary data type for the entire DOM. The DOM structure uses parent-child relationships between the nodes and contains a single root node, hence creating a tree structure. Although we access the DOM structure as if it was a tree, the underlying structure might be completly different. When we refer to the DOM structure's tree form, we use the term DOM tree instead of DOM structure.

The Node class represents a single node in the tree representing a document. Several classes extend[11] the Node interface, among them Element, Comment,

[11]The Element and Document class inherit directly from the Node interface, while the Comment and Text class inherit it indirectly.

Text and Document. This allows us to use polymorphism to access all the nodes in the DOM structure in a uniform way.

Before we look at our first HTML example, we need to define what an open element is, and what a valid child is.

We define an open element to be an element where the start-tag has been processed correctly, while the corresponding end-tag has not, directly or indirectly, been processed. An end-tag can be processed indirectly if the end-tag of the element type is optional, or the content model is empty[12]. All the ancestors to an open element are also open.

In the rest of the text we will write start- and end-tag, even though the tags could be optional and thereby not a part of the original HTML code. Since elements using optional tags in SGML must be unambiguous, this statement will not affect the correctness of my later statements.

A node N is a valid child of an element M if and only if:

- N is situated between the start- and end-tag of M
- There are no other elements between the start- and end-tag of M that are open at the time N is processed, which can contain node N. If these open elements' element types does not allow omittance of the end-tag, N is not a valid child of element M.
- The content model of M allows N to be added at the time when it is encountered(it does not break the order or count of the content model at the current time).

2.3.1 Our first HTML example

Now we are ready to look at our first HTML example.

[12]The end-tag is forbidden for all elements with an empty content model in HTML.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                 "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
 <HEAD>
  <TITLE>Valid HTML</TITLE>
 </HEAD>
 <BODY>
  <TABLE>
   <TBODY>
    <TR>
     <TD>This is valid HTML</TD>
     <TD>This is my second column</TD>
    </TR>
   </TBODY>
  </TABLE>
 </BODY>
</HTML>
```

Example 2.4: Our first valid HTML example

This HTML document in Example 2.4 will display two columns, one with the text:"This is valid HTML" and one with "This is my second column".

By using the Table 2.1 we can verify that this is a valid HTML document. The parent-child relationships which exist are shown in Table 2.2.

Parent	**Child/Children**
HTML	HEAD, BODY
HEAD	TITLE
BODY	TABLE
TABLE	TBODY
TBODY	TR
TR	TD#1, TD#2

Table 2.2: Which parent-child relationships exist in Example 2.4.

Using the parent-child relationships shown in Table 2.2, we can create a DOM tree. This DOM tree is presented in Figure 2.2.

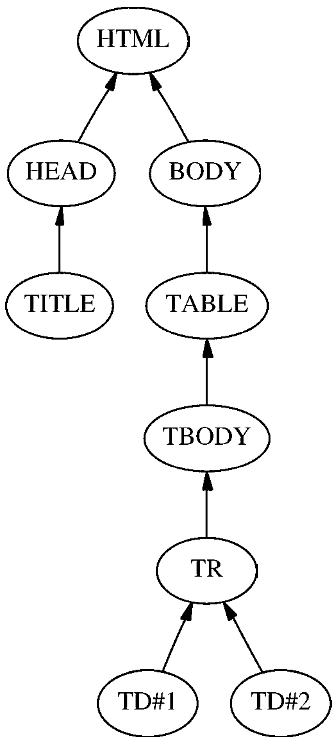


Figure 2.2: Representation of the HTML document in Example 2.4 as a DOM tree.

Note that the following two examples, Examples 2.5 and 2.6, are equivalent to Example 2.4.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                "http://www.w3.org/TR/html4/strict.dtd">
<HTML><HEAD><TITLE>Valid HTML</TITLE></HEAD><BODY>
<TABLE><TBODY><TR><TD>This is valid HTML</TD><TD>
This is my second column</TD></TR></TBODY></TABLE>
</BODY></HTML>
```

Example 2.5: The first HTML document equivalent to Example 2.4

In Example 2.5 we have removed most whitespace characters from the HTML code.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                 "http://www.w3.org/TR/html4/strict.dtd">
<TITLE>Valid HTML</TITLE>
<TABLE>
  <TR>
    <TD>This is valid HTML
    <TD>This is my second column
</TABLE>
```

Example 2.6: The second HTML document equivalent to Example 2.4

In Example 2.6 we have omitted the start-tag for the elements HTML, HEAD, BODY and TBODY, and the end-tag for the elements HTML, HEAD, TBODY, TR and TD. According to the DTD used, all these tags can be omitted.

While we are looking at HTML documents we may also look at our first example of invalid HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                 "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
 <HEAD>
  <TITLE>Valid HTML</TITLE>
 </HEAD>
 <BODY>
  <TABLE>
   <TBODY>
     <TD>This is valid HTML</TD>
     <TD>This is my second column</TD>
   </TBODY>
  </TABLE>
 </BODY>
</HTML>
```

Example 2.7: Our first invalid HTML document

The HTML document in Example 2.7 is the same as Examples 2.4, 2.5 and 2.6, except that the TR element have been removed. TD is not allowed to be a child of TBODY, since it is not contained at all in TBODY's content model, which is (TR)+. Since TR is the only element type which has TD in its content model, we cannot add any elements with optional start- and end-tags to solve the inconsistency.

Even though this example contains invalid HTML you can still view it in both Netscape Navigator and Internet Explorer. We will get back to invalid HTML in Chapter 4.

3. Parsing valid HTML

A parser is an algorithm or program which determines the syntactic structure of a sentence or string of symbols in a language[How]. In our context it means the transition from an HTML document as a set of consecutive ASCII characters, into the HTML document structure as a data structure.

The parser is dependent on a lexical analyser which reads the character data and converts it into tokens. In HTML, tokens are typically generated for start-tags, end-tags, comments, text data, document type declarations and a few more.

The parser works on a higher level of abstraction than the lexical analyser, and is responsible for processing the tokens and creating the data structure representing the HTML document. It uses one of the HTML DTDs to enforce rules on the relationship between elements, noticing and possibly fixing errors in the HTML document.

We will now take a closer look at how a lexical analyser for HTML should work, before we turn our attention to the parser.

3.1 A lexical analyser for HTML

HTML is an SGML application, and user-agents should therefore comply with the lexical syntax defined in [ISO86]. Unfortunately, they generally do not.

The reasons why they do not comply with the [ISO86] could be:

- SGML is a complex technology. Although HTML only uses a small subset of the features of SGML, these features are intervened into the specification of SGML and are difficult to extract
- The authors of most user-agents have used the KISS[1] principle and precedence of earlier browsers, when implementing their user-agents
- The features omitted in user-agents provide more complexity to HTML with little gain of expressiveness. Some of the features also decrease readability.

[1]Keep it simple, stupid.

Features that have been excluded from user-agents are for instance :

- Tag minimization
  - Empty start- and end-tags: <>text</>
  - Omittance of TAGC[2] when two tags are succeeding: <HTML<HEAD> </BODY</HTML> .
- Marked sections: <![CDATA [<This >Will not be interpreted as <markup>]]>
- Defining attribute with a unique value from a name token group: <DIV center> instead of <DIV align="center"> .

Some of the features that have been included are not implemented in accordance with SGML standards. Comment syntax is one of these. An HTML document[3] made by Ian Hickson tests if a browser is handling comments correctly. A test of Netscape 4.78, Netscape 6.01, Internet Explorer 4, Internet Explorer 5 and Opera 5.0 shows that none of the browsers parsed comments correctly. According to [Gol90] a comment starts with the com delimiter(which is -- for HTML) and ends with the com delimiter. A comment can only occur in markup declarations. A comment declaration starts with the mod delimiter (<! in HTML) followed by one or more comments (with possible whitespace characters between them) and ends with the mdc delimiter(>). The important thing to notice is that a comment is not stopped if a mdc(>) delimiter is encountered, only if a com delimiter (--) occurs. In the Example 3.1 we will see how comments are specified.

```
<!-- This is the first comment, we will now end it and start another--
-->
<p>This text should be ignored totally as it is in the second comment
<! -->
```

Example 3.1: Two comments encapsulating all text and markup.

In Example 3.1 there are two comments encapsulating all other markup and text.

Most browsers on the other hand, treat "<!--" as the start of the comment and the next occurrence of "-->" as the end of the comment. Some browsers even treat the next occurrence of "->" as the end of the comment.

As long as HTML is an SGML application, lexical analysers for HTML should correctly tokenize an HTML document according to SGML specifications.

[2]This is defined to be > in HTML.

[3]The test is available at http://lxr.mozilla.org/mozilla/source/htmlparser/tests/logparse/comments.html.

A description and implementation[4] of a lexical analyser for HTML, and SGML application similar to HTML, is given in [Con97]. The sgml-lex program Dan Connolly describes uses callback functions every time it has found a token. It uses three different callback functions:

1. Primary callback: Used for start-tags with attributes, end-tags and character data
2. Secondary callback: Used for other markup declarations such as document type declarations, comments and marked sections. We also use this callback function to pass a special token called document end, which marks the end of the document[5]
3. Third callback: Used for syntax errors.

In the next section we will assume we have a lexical analyser, like sgml-lex, which correctly tokenizes HTML documents according to SGML rules.

3.2 The Parser

The goals of the parser described in this section are:

- To build a valid HTML structure according to the DTD used in the document
- In the event of an error; discard the token, report the error and continue parsing the remaining tokens.

We will use the DOM level 2 interface defined in [W3C] as the data structure. To thoroughly understand the following part it is important to understand the hierarchy of the classes used in the DOM level 2 interface.

[4]The implementation available does not include all SGML features available in HTML.

[5]This token should only be used when we have read all the data there is. It should not be used when the end-tag corresponding to the element type HTML is found, since we might have more data after it.

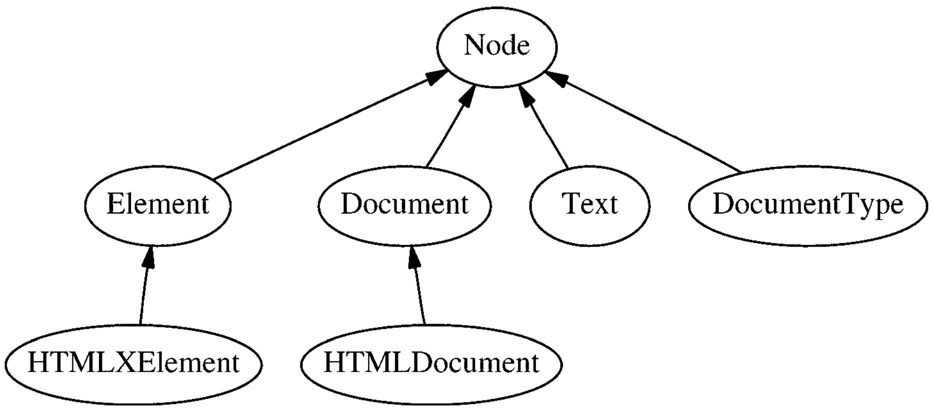


Figure 3.1: Inheritance between DOM classes.

As we can see from Figure 3.1, all classes used are also subclasses of Node. Therefore, all nodes in the Document tree can be treated as Node instances.

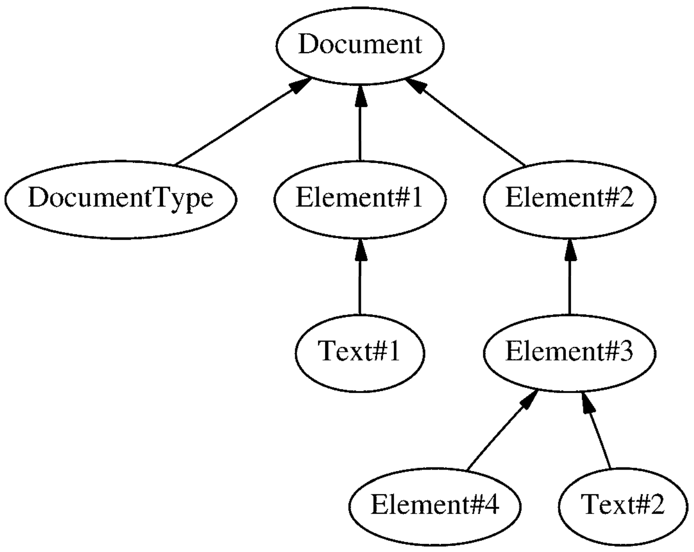


Figure 3.2: A typical DOM tree.

Figure 3.2 shows an example of which classes the different Nodes in the tree are typically also an instance of.

We will now describe a step-by-step method of parsing valid HTML documents. We will assume we always have a reference to the current node. This is generally the node which was last added to the DOM structure. We always try to add new nodes to the current node first.

In the DOM structure we have no methods for opening and closing elements. Instead, we use the current node to specify which elements are open and which elements are closed. The current node and all elements which have the current node in its subtree are open(these elements are the current nodes ancestors), while the rest of the elements are closed.

The first step when parsing an HTML document is to create a Document instance, which will be the root of the DOM tree, with a DocumentType declaration as a child. The document type declaration should be the first token created (except comments which we will ignore in this context) by the lexical analyser. The document type declaration token will be passed to the secondary callback function and a Document object with the specified document type declaration will be instantiated at that moment[6]. We make the current node point to the Document Element which we acquire from the newly created Document. If a callback to the primary callback function is executed before the Document has been instantiated, we should discard the token and report the error.

For the rest of the description on how the parser should work, we assume the Document has been correctly instantiated.

3.2.1 Handling a start-tag

If a token is passed to the primary callback function and it is a start-tag, we propose the following algorithm:

1. If the start-tag represents a valid element type, meaning it is present in the DTD defined by the document type declaration and specifies all required attributes for this element type, we create an Element instance representing the element started by the start-tag. If there is an invalid attribute specification, we discard the element and the token, and report the error[7]. The attributes need to be post-processed here according to Section 7.9.3 of the SGML standard. This is done by:

    - Replacing character and entity references, for example replacing &lt; with $<$ and &#960; with π
    - Deleting character 10(ASCII Line Feed)

[6]We cannot create the Document before since DOM level 2 does not allow changing the Document type of a Document once it has been instantiated.

[7]We could have ignored the invalid attributes, but then we would do error-correcting.

- Replacing character 9 (ACSII Horizontal Feed) and character 13 (ASCII Carriage Return) with character 32 (ASCII Space).

   We call the newly created Element elementA and proceed to the next step.

   If the start-tag does not represent a valid element, we discard the token, report the error and await the next callback.

2. Make a new pointer to the current node called currentOld. This will help us backtrack if an error occurs. Proceed to the next step.

3. If the current node in its current state[8] accepts elementA as a child, add elementA as a child node to the current node. Change the current node to point at elementA. The token has now been processed correctly, and we await the next callback.

   If the current node does not accept elementA as a child, go to the next step.

4. Check if an element type with an omittable start-tag exists, which we can add an instance of as a valid child to the current node, and which allows elementA to be added to it. If such an element type exists[9], create an instance of it, add it as a child to the current node and add elementA as a child to it [10]. Change the current node to point at elementA.

   If such an element type does not exist, go to the next step.

5. If the current node has an omittable end-tag and is not the Document node(the root of the DOM tree), make the current node the parent of the current node and go to step 3. If the current node does not have an omittable end-tag, go to the next step.

6. If we come to this step, we know the document is invalid because we failed to insert a new element to the DOM tree. We will report this error and backtrack to the position we were before the callback occurred. This is done by changing the current node to point to currentOld.

Let us look at some figures illustrating how a start-tag is processed. The most simple case is when the element it represents the start of, can be added to the current node. This is shown in Figure 3.3.

[8]The current state refers to the state of the content model.

[9]More than one element will never exist due to SGML rules on ambiguities regarding tag minimization.

[10]Although it is sufficient in valid HTML documents to attempt to find a parent with an omittable start-tag, documents from other SGML applications might require a great-grandparent, a grandparent and a parent all with omittable start-tags. We should look for a sequence $a_1, \ldots, a_n$ where a_1 can be added as a child to the current node, a_i can be added as a child to a_{i-1} for $i \geq 2$ and elementA can be added as a child to a_n.

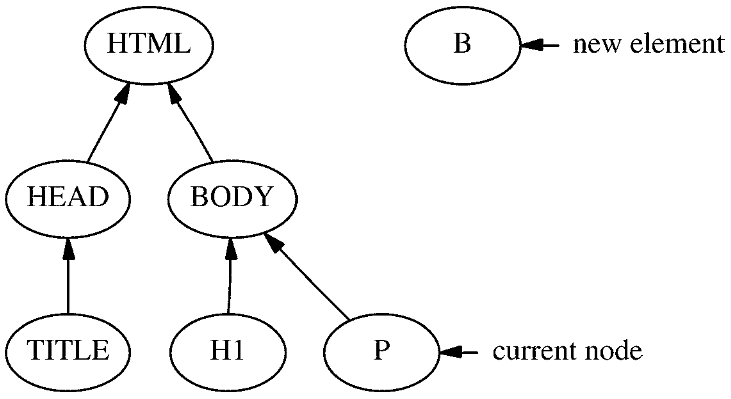


Figure 3.3: The DOM tree before the addition, the new element is B and the current node is P.

In Figure 3.3 the current node's element type, P, allows the element type B as a child. Therefore, the B element will be added to the current node. The current node is then change to point at the B element. The resulting DOM tree is shown in Figure 3.4.

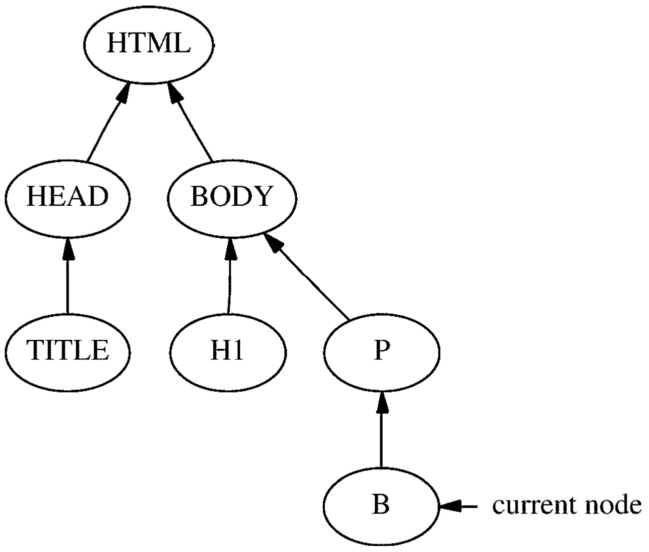


Figure 3.4: The DOM tree after the insertion of B.

If the new element cannot be trivially added to the current node, we have two choices:

1. Add an element with omittable start-tag which the new element can be a child of
2. Change the current node to the parent of the current node if the current node has an omittable end-tag. Then attempt to add the new element as a child to the new current node.

An example of the first choice is the addition of a TR element, when the current node is a TABLE element. This is shown in Figures 3.5 and 3.6. Please note that in HTML version 3.2 this will not happen since there is no TBODY element type in this version. Instead, TR elements can be contained directly under TABLE elements.

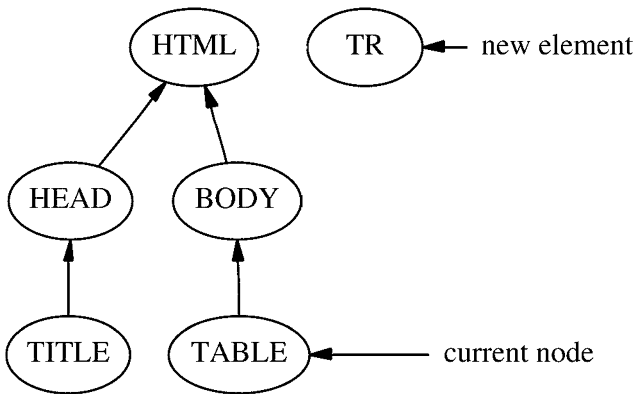


Figure 3.5: The DOM tree before the addition, the new element is TR and the current node is TABLE.

The situation in Figure 3.5 is solved by adding a TBODY element as a child to the TABLE element, and the TR element as a child to the new TBODY element. The result is presented in Figure 3.6

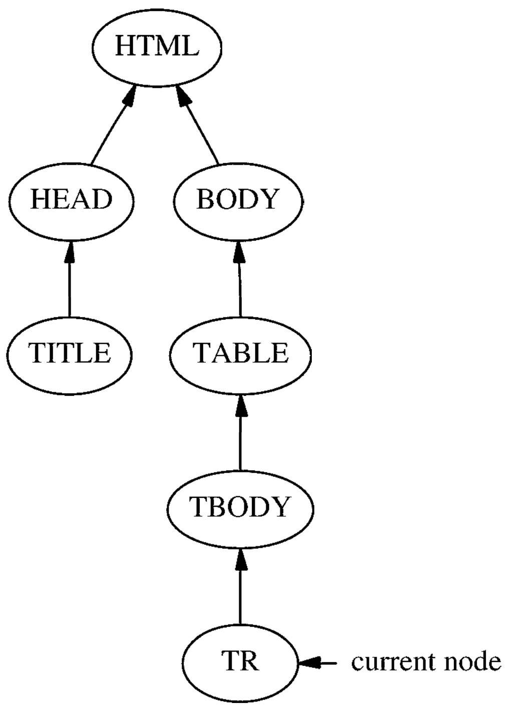


Figure 3.6: The DOM tree after insertion of TBODY and TR.

We will now present an example where we need to implicitly close the current node, in order to allow the addition of a new element. The example shows the situation which occurs when we attempt to add a TABLE element when the current node is a P element. It is presented in Figures 3.7 and 3.8.

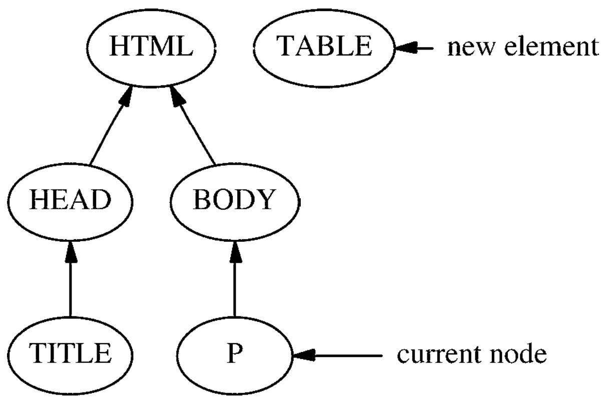


Figure 3.7: The DOM tree before the addition.

In Figure 3.7 the P element does not allow a TABLE element as a child in its content model. Therefore, we cannot add the TABLE element as a child to the P element. The solution to the problem is to close the P element since its element type has an omittable end-tag, and add the TABLE element as a child to the parent of the P element. This is shown in Figure 3.8.

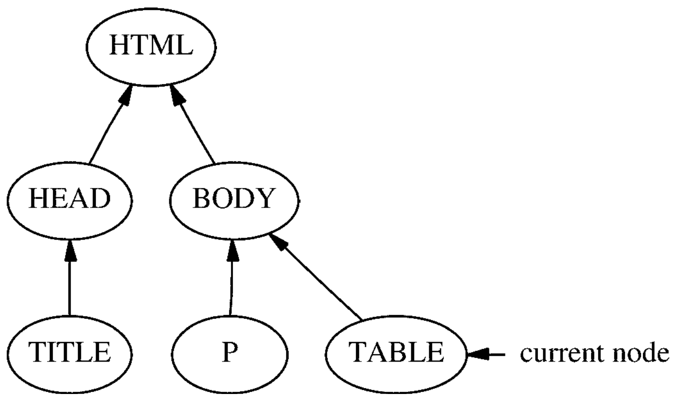


Figure 3.8: The current node is changed to be the BODY element before the TABLE element is added as a child to it. Afterwards we change the current node to TABLE.

3.2.2 Handling an end-tag

We will proceed with the handling of end-tag tokens. We propose the following algorithm:

1. Check if the end-tag corresponds to a valid element type. If it does not, discard the token, report the error and await further callbacks. If it does, go to the next step.

2. Make a new pointer to the current node called currentOld. This will help us backtrack if an error occurs. Proceed to the next step.

3. If the end-tag matches the current node and the content model of the current node is valid if closed at this point, we change the current node to the parent of the current node. If the current node cannot be closed because of its content model, we discard the current node and all nodes in its subtree, and report the error. If the end-tag does not match the current node, go to the next step.

4. Check if the current node's element type[11] has an omittable end-tag, the content model is valid if closed at this point and the current node is not the Document node. If all this is true, we change the current node to the parent of the current node and return to step 3. If not, we will proceed to the next step.

5. If we come to this step, we know the document is invalid, because we failed to process the end-tag token. We will report this error and backtrack to the position we were before the callback occurred. This is done by changing the current node to point to currentOld.

The Figures 3.9 and 3.10 show a trivial end-tag token handling.

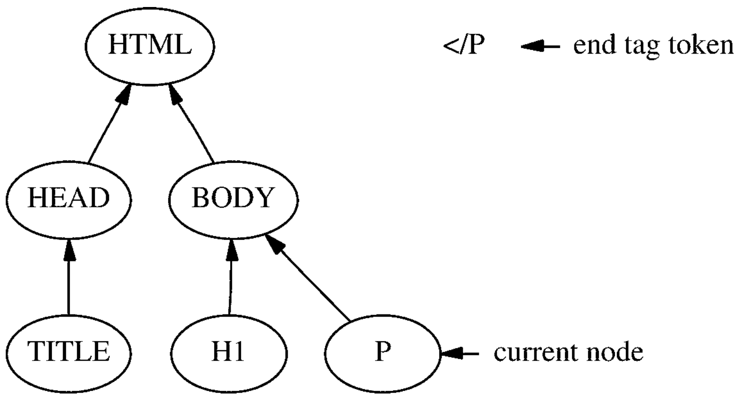


Figure 3.9: The DOM tree before receiving an end-tag token of element type P.

[11]The current node is almost always be an element. If the current node is a text node, we should make the current node point to its parent instead, since text nodes cannot have children.

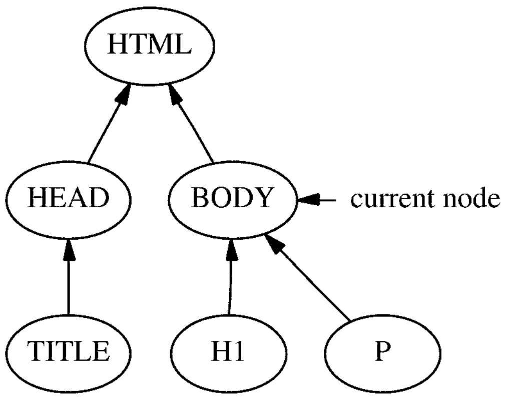


Figure 3.10: The DOM tree after handling the end-tag token.

In Figure 3.9 we see that the end-tag trivially matches the P element. The result, the closure of the P element, is shown in Figure 3.10. This is done by moving the current node one level up in the DOM tree, so that its parent is now the current node.

In Figure 3.11 we have received an end-tag token for TABLE. The current node is the second TD element when the token is received.

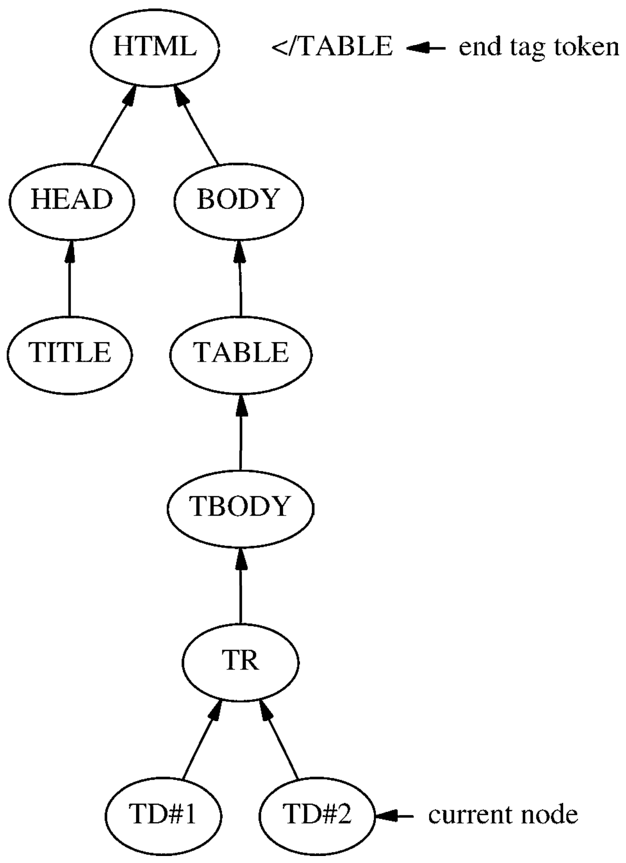


Figure 3.11: The DOM tree before receiving an end-tag token of the element type TABLE.

Because the TD, TR and TBODY elements have optional end-tags, these elements are implicitly closed when handling the end-tag token in Figure 3.11. The end-tag token matches the TABLE element which is also closed. Therefore, the current node becomes the parent of the TABLE element, which is the BODY element. This is shown in Figure 3.12.

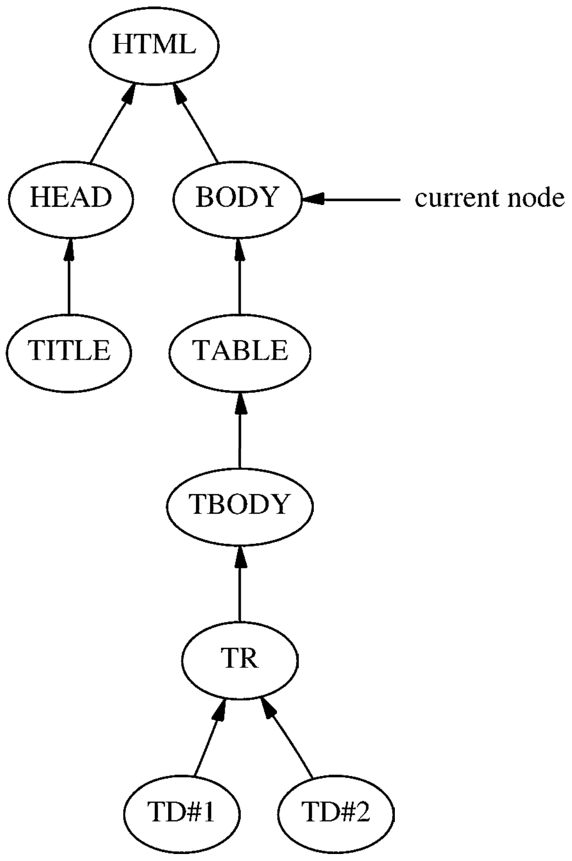


Figure 3.12: The end-tag did not match the element type of the current node, and resulted in an element higher up in the DOM tree than the current node being closed.

3.2.3 Handling a data token

The last token to use the primary callback function is Data or text. We propose the following algorithm:

1. Create an instance of a Text node representing the data. This includes expanding character and entity references if it does not represent a CDATA text node. Go to the next step

2. If the current node in its current state accepts a Text node[12] as a child, add the new Text node to it. We need not change the current node because Text nodes cannot have children, they are always leaves in the DOM tree.

[12]In HTML it accepts text if it has #PCDATA or CDATA in the content model.

If the current node does not accept Text nodes as children, go to the next step.

3. Check if the element type of the current node has an omittable end-tag and that the current node is not the document node. If both the statements above are true, change the current node to the parent of the current node and go to step 2. If the current node cannot omit its end-tag, go to the next step.

4. If we come to this step, we know the document is invalid because we failed to process the data token. We will report this error and backtrack to the position we were before the callback occurred. This is done by changing the current node to point to currentOld.

Note that if we were to parse SGML documents with the same features as HTML, we would need to add an extra step after step 3. This step would check if text could only be placed under an element type with an omittable start-tag. If there was such an element type and the current node allowed it in its content model, we would create an instance of it, add it to the current node and add the Text node to it. Otherwise, we would continue to the final step.

The Figure 3.13 shows the DOM tree directly after a Text node has been added. Notice that the current node is not the Text node, but its parent.

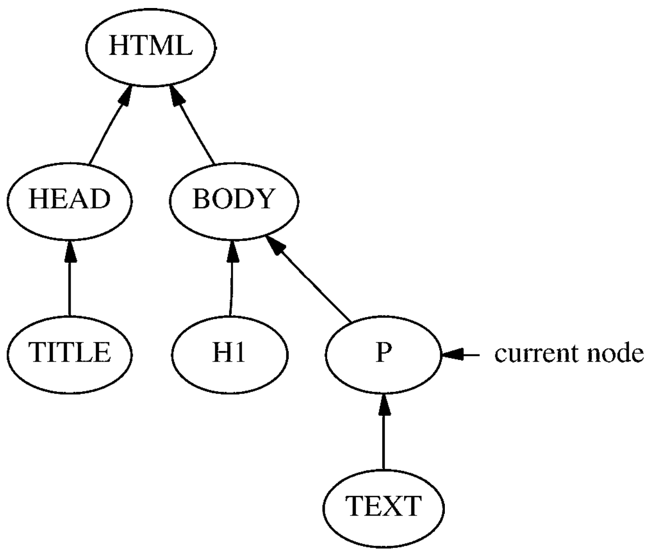


Figure 3.13: After trivially adding a Text node to the current node when the content model of its corresponding element type accepts #PCDATA.

3.2.4 Handling secondary callback tokens

The secondary callback function handles the document type declaration token (which we have already shown how to handle in Section 3.2), marked sections and the document end token.

The different marked sections are defined in the ISO 8879[ISO86] standard in Section 10.4.2 . We propose the following algorithm for parsing marked sections:

First replace any entity references in the name of the marked section. When encountering an:

- INCLUDE token, we should pass the characters inside the marked section back to the lexical analyser and let it analyse this data for markup immediately. The tokens from this section will then be passed on to the primary callback function
- IGNORE token, we should ignore the token
- RCDATA token, we should generate a data token from it and pass it on to the primary callback function
- CDATA token, we should generate a data token from it and pass it on to the primary callback function. This token must contain information that entity and character references should not be replaced.

The reason why marked sections cannot be handled by the SGML lexical analyser is that SGML documents with the same features as HTML can use entities instead of the keywords INCLUDE, IGNORE, RCDATA and CDATA. For example a DTD might specify the entity image to be the value IGNORE, while another version of the same DTD might give the value INCLUDE. This allows one document to adhere to both of these DTD by using the code in Example 3.2

```
<! [ &image; [ <IMG src=''myimage.jpg''> ] ]>
```

Example 3.2: Marked section that cannot be handled by the lexical analyser.

The lexical analyser has no access to the DTDs and cannot make a decision if this section should be included, ignored, treated as RCDATA or treated as CDATA.

If the keyword in a marked section declaration is not an entity, the lexical analyser could handle it without notifying the parser.

Generally, the lexical analyser can use a queue of tokens awaiting processing by the parser, instead of waiting for the parser to finish processing the current token before the lexical analyser creates the next token. If a marked section is encountered, the lexical analyser should not proceed to read new tokens before the marked section is handled properly by the parser. This must be done since the parser can return the contents of a marked section for further analysis by the lexical analyser, and any tokens from this marked section should precede tokens generated from any HTML code after the marked section declaration.

If we encounter a document end token, we should attempt to close the Document element. The Document element is the root of the DOM tree and there will always exist one and only one such element for each Document. The Document element in HTML is of the element type HTML. Therefore, we treat the occurence of the document end token in the same manner as an end-tag of the element type HTML. If any open elements cannot be closed due to an incomplete content model, we have an invalid document. The error should be reported and the invalid elements should be discarded.

3.2.5 Handling error callback tokens

The error tokens will be handled at the application level. The parser should use callback methods for this purpose.

3.2.6 Recommendation for implementation of the parser described

In this section we will briefly describe how the parser could be implemented in Java. Although the code is in Java, it is possible to use a similar code in most object oriented programming languages.

We assume we have a lexical analyser written in Java and we call the class SGMLlex.

Since Java does not have function pointers we need to specify callback functions a bit differently than we would in for example C++. The approach we recommend involves creating an interface, which we call SGMLlexCallback, that contains the methods shown in Example 3.3.

```
primaryCallback(PrimaryToken token)
secondaryCallback(SecondaryToken token)
errorCallback(ErrorToken token)
```

Example 3.3: The SGMLlexCallback interface.

Our HTML parser implements the interface in Example 3.3, meaning that any object with a reference to it is guaranteed that these methods exist.

The SGMLlex provides a method for specifying which object it shall use for callbacks. This method only accepts objects that are instances of SGMLlex-Callback. The method stores a reference to the SGMLlexCallback instance, and when the lexical analyser discovers a token it calls the appropriate callback method on this reference. Since our parser implements the SGMLlexCallback, it is also an instance of it and can be used as a parameter to this method.

Let us see an example of how this works. We show the two methods setCallback and parse of the SGMLlex class in Example 3.4.

```
public void setCallback (SGMLlexCallback callback) {
   //Store the reference to the callback object
   myCallback = callback ;
}

public void parse() {
    Token newToken = getNextToken() ;
    if (newToken instanceof PrimaryToken){
        myCallback.primaryCallback((PrimaryToken)foundToken) ;
    } else if (newToken instanceof SecondaryToken) {
        myCallback.secondaryCallback((SecondaryToken)foundToken) ;
    } else if (newToken instanceof ErrorToken) {
        myCallback.errorCallback((ErrorToken)foundToken) ;
    }
}
```

Example 3.4: Callback functions in the lexical analyser.

As shown in Example 3.4 we cast the Token into either PrimaryToken, SecondaryToken or ErrorToken before we call the callback function. Which Token we use depends on the callback function to be used.

In order for this to work we have the inheritance between the different Token classes as specified in Figure 3.14.

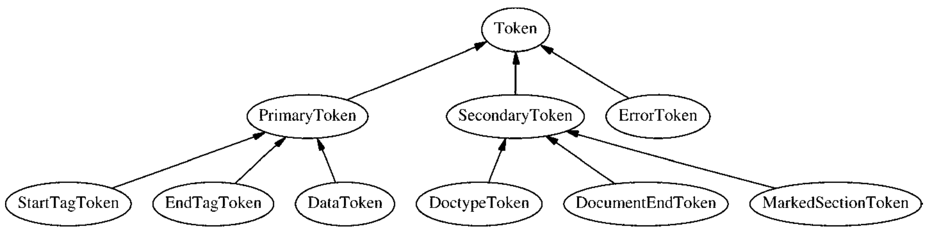


Figure 3.14: Inheritance between the different tokens.

We have created a Java class which implements the features of the valid parser described in this chapter. It has not been compiled and should be considered not be considered as working Java code. The reason why it has not been compiled is because the classes which it depends on have not been developed. It is well documented, and included as Appendix A.

The parser should allow the presentational module to get access to the DOM structure before the parser is finished parsing all the tokens. This should be done to allow incremental viewing of an HTML document. In order to do this the parser should have its own callback functions which specifies when the DOM structure has changed and is relatively stable. If an error in the HTML document occurs which changes the relatively stable DOM structure, we should invalidate the DOM structure through a callback function so that presentational module knows that it should process the DOM structure again.

Most browsers seem to display any child of the BODY element as soon as the child is closed[13]. We propose to do the same.

[13]This property of browsers was found by using a cgi script which added a time delay after each line of HTML code.

4. Invalid HTML

4.1 Types of invalid HTML

Having given a method on how to parse valid HTML, we will now turn our attention to invalid HTML.

We have two types of errors in HTML documents:

- Semantical
- Syntactical.

4.2 Semantical errors

Semantical errors are categorized by misuse of the element types and attributes available in HTML, and their intended meaning. This type of error is often spawned from the fact that browsers interpret descriptive elements into procedural commands. The author or viewer of an HTML document see how different element types instances are display in browsers and add his own semantics to them based on this. Thereby he wrongly assumes that the descriptive element types will always result in the same procedural command.

Let us take a closer look at some common semantical errors.

```
<P>And then the big bad wolf said</P>
<H1>Then I'll huff and I'll puff and I'll blow your house down</H1>
```

Example 4.1: First example of semantical errors in HTML.

It is clear that the text in the heading element of level 1 (H1) in Example 4.1 is not a heading. The motive of the author of the page is probably to present the quote in a large font, which is how browsers render H1 elements.

```
<UL>This text will be indented, even though semantically and
syntactically wrong.</UL>
```

Example 4.2: Second example of semantical errors in HTML.

In Example 4.2 we see a common semantical error in HTML documents. All major browsers[1] display an unordered list element (UL) as indented from the margin, with a bullet in front of each subsequent list item (LI) element. But they also allow #PCDATA sections directly under the UL element, which is a syntactical error, and when this text is displayed it will also be indented. Therefore, authors may mistakenly believe that the UL element provides a method for indenting text. This problem is even more critical since most people have never heard about the HTML standard or the HTML DTD.

My last example of semantical errors shows some of the problems that may occur when user-agents use other means of presenting HTML documents than the method used in graphical browser. Alternative browsers may display HTML documents as text only[2] or convert it into audio to assist the visually impaired[3].

```
<IMG src=''image4.jpg'' alt=''image4.jpg''>
```

Example 4.3: Third example of semantical errors in HTML.

As stated in [W3C99] the alt attribute should be used as an alternative textual representation of the element. In Example 4.3 the alt attribute says nothing about what the image depicts. The alt attribute should not always be a description of the image rather a textual alternative for the meaning of the image[Hic, Fla01].

Charles Goldfarb mentions in [Gol90],p.216 that semantical errors can only be detected by a person. The HTML DTD cannot enforce any semantical rules regarding HTML.

In the case of semantical errors we can therefore only guide the authors of HTML documents into good coding styles, and not correct their previous errors.

4.3 Syntactical errors

When an HTML document instance does not conform to the HTML DTD and the constraints implied by SGML, a syntactical error occurs.

[1]The most dominating browsers at the moment are Internet Explorer and Netscape Navigator.

[2]An example of a web browser which only displays text is lynx.

[3]An example of a browser for the visually impaired is IBM Home Page Reader.

We can check an HTML document for syntactical errors by using a validating SGML parser.

The basic operations of a validating SGML parser when validating a document is:

1. Find out which DTD the document uses
2. Validate the structure and syntactical validity of the DTD
3. Parse the document into a document structure
4. Check for markup errors in the document structure. If it has no errors it is valid according to the DTD it uses.

Most online HTML validators use a validating SGML parser to do the validation of an HTML document. There are are two major online HTML validators:

- The W3C validator `http://validator.w3.org`
- The WDG validator `http://www.htmlhelp.com/tools/validator`.

In October 2001 the W3C validator validated approximately 80000 HTML documents per day.

Before we start analysing the different error types in HTML documents let us first present the goals we have for our method of parsing invalid HTML documents.

4.3.1 Goals for our method of parsing invalid HTML documents

Our goals for parsing HTML documents are:

1. After the parsing process is finished we should have a DOM structure which can be translated into a document which is valid according to a known SGML DTD[4] with the same features as HTML.
2. This DOM structure created from the HTML document should contain sufficient information for a presentation equal to major browser, especially Internet Explorer[5]
3. Valid HTML documents should not be affected by the method of error fixing
4. It should be easy to change the rules of the error fixing.

[4]Not necessarily the W3C HTML DTD.

[5]Statistics from October 2001 published by thecounter.com show that 79% of all requests to webservers are done by different versions of Internet Explorer. The actual percentage is probably not that great since other browser identify themselves as Internet Explorer for compatibility reasons, but there is little doubt that Internet Explorer is by far the most used browser.

Goal number one will help us separate the parsing process of an application from the process of representing the information present in the HTML document. The presentational module we use do not have to worry about every possible combination of elements, only those allowed in the DTD the DOM structure should conform to. Goal number one will also allow us to validate the result of the parsing by using a validating SGML parser. This will help us test that our parser functions correctly.

4.3.2 Error types identified

All of the error types found will be presented in the coming sections.

The error types have been identified using an SGML parser during the work done on testing HTML documents for validity, as describe in Chapter 5. This means that these error types are all present and probably handled in major browsers. Otherwise, the error would have been fixed by the author since the document would not display as he desired.

In order to allow our parser to function in the same manner as major browsers, we need to provide a method of correcting these errors as well.

Each error type will presented with:

- A description of the error type
- One or more examples of the error type
- How the error type can be identified
- How to correct the error.

We will assume the parsing process is executed as described in Chapter 3. It is especially important to understand that we use a lexical analyser to generate tokens which our parser processes.

4.3.3 Missing or wrong document type declaration

Description: According to the HTML standard[W3C99], Chapter 7, an HTML document is composed of three parts :

1. A line containing HTML version information
2. A declarative header section
3. A body, which contains the actual content.

The line containing the HTML version information is called the document type declaration, and is a reference to the Document Type Definition (DTD) used in this document as stated in[ISO86][6].

Example: We provide two examples of valid document type declarations.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                           "http://www.w3.org/TR/html4/strict.dtd">
```

Example 4.4: A valid document type declarartion for HTML 4 strict.

The HTML code in Example 4.4 will identify the document as an HTML 4.01 document complying to the strict DTD. The first word after the start of the declaration is used to specify the element type of the root of the document, and should always be HTML in HTML documents.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

Example 4.5: A valid document type declaration for HTML 3.2 .

The document type declaration in Example 4.5 will identify the document as an HTML 3.2 document. HTML version 3.2 provides only one DTD and not two like HTML 4.

User-agents rarely use the document type declaration in a normal way, due to the fact that they are “hard-coded” to parse HTML documents, and not generic SGML documents. Lately, browsers have had two modes, one strict- and one “quirks”-mode. The “quirks”-mode is normally used for HTML documents with errors or which use the transitional DTD, while the strict-mode is used for HTML documents that specify that they use the strict DTD. If an HTML document is parsed using the strict-mode and an error is found, the mode is changed to “quirks”. The reason why they seperate between these two modes, is that there is less work done when parsing a valid HTML document.

Validating SGML parsers need the DTD of an SGML application in order to validate a document which conforms to the DTD. This allows a single validating SGML parser to validate documents of all markup languages defined through SGML. Since most HTML validators use a validating SGML parser, the document type declaration of an HTML document must be defined for it to be validated with these.

[6]It is actually a bit more complicated than this because one can omit the document type declaration if the processing application can work without it. Since the SGML parser needs the document type declaration, and it is specified as a necessity in the HTML standard, we require it for all HTML documents.

```
<HTML>
<HEAD>
<TITLE>Invalid document</TITLE>
</HEAD>
<BODY>
<P>Invalid because of missing document type
declaration at the top of the source
```

Example 4.6: Missing document type declaration at the top of the HTML document.

Example 4.6 is invalid because it lacks the document type declaration.

```
<!DOCTYPE HTML ``HTML4.01''>
<HTML>
<HEAD>
<TITLE>Invalid document</TITLE>
</HEAD>
<BODY>
<P>Invalid because of wrong  document type
declaration at the top of the source
```

Example 4.7: Invalid document type declaration at the top of the HTML document.

Example 4.7 is invalid because it uses an invalid syntax to define the document type definition.

Identify: We can notice the error if one of the following situations arise:

- The first non-comment token the lexical analyser finds is not a document type declaration token, but instead a start-tag, end-tag, data token or marked section
- The document type declaration is not in accordance with the specification defined in [ISO86]. The lexical analyser should notice this
- The document type declaration is syntactically valid, but refers to an unknown DTD. This is actually not an error, but our parser cannot process general DTDs.

Correct: It is very difficult to correct a missing or wrongly defined document type declaration. Without it we can not be certain that the document at hand is in fact an HTML document and not a normal text file.

We have four different alternatives to choose between:

1. Use a specific DTD

2. Attempt to use “doctype sniffing”, a method in which we attempt to find out which DTD was used

3. Check the content-type in the http header which is sent from the server. If this is text/html, we know we should treat it as an HTML document and can use approach number 1. If it is not text/html, we should not display it since it is not an HTML document

4. Do not parse the document at all, only report an error.

If we use the first alternative we must decide which DTD to use. This could be the transitional DTD, since it contains most element types and attributes in use in the different HTML versions, or a custom made DTD. We propose to use two different DTDs while parsing HTML documents. We should first attempt to process each token with one of the standard DTDs. If the token cannot be added to the DOM structure in a valid manner according to this DTD, we attempt to process the token using the rules of the custom DTD. When we later refer to the DTD used, we refer to this combination of two DTDs. One should be able to represent most HTML documents on the WWW with the custom DTD. In Section 4.3.10 we will take a closer look at a custom DTD.

If we use “doctype sniffing”, we search for certain words in the existing invalid document type declaration. Generally, this is not recommended because it is difficult to know which DTD the user intended to use. An exception is when searching for the URI[7] of the W3C standard DTDs, such as “http://www.w3.org/TR/html4/strict.dtd” and “http://www.w3.org/TR/html4/loose.dtd”

Unfortunately, alternative number 3 will not work well in practice. This is because a lot of servers send HTML documents with a http header defining the content-type to be text/unknown instead of text/html. To be compatible with other browsers, we need to display these documents.

If we do not parse HTML documents with a missing or invalid document type declaration at all, our parser will not be able to process 81% of all HTML documents on the web[8], which is not acceptable.

From the above, we must conclude that if the document type declaration is syntactically wrong, look for an URI which uniquely identifies it. If there is no such URI, or no document type declaration is specified, we use a combination of the transitional DTD and a custom DTD, even though the content-type is text/unknown.

[7]Universal Resource Identifier, most URIs are URLs(Uniform Resource Locator.

[8]See Section 5.2 for more statistical information.

4.3.4 Invalid markup declaration

Description: We have an invalid markup declaration if we in an HTML document have a markup declaration open (by typing the characters `<!`) and which does not correspond to valid comment or marked section[9].

Example:

```
<!-I thought this was a comment->
```

Example 4.8: Invalid comment declaration.

The HTML code in Example 4.8 is not a valid comment, see Section 3.1 for how a valid comment is defined.

Identify: This problem is identified by the lexical analyser when it attempts to make a comment, or marked section token.

Correct: We ignore the invalid markup declaration.

4.3.5 Unknown entity or character reference

Description: General entities or character references are used in HTML documents to type characters which are not available on all keyboards. The entities are included into the HTML DTDs from the files HTMLspecial.ent, HTMLsymbol.ent and HTMLlat1.ent, while the character references are defined in the Unicode standard[The91]. General entities are used by typing &nameofentity; or &nameofentity , while for character references we use the notation &#number; or &#number. We have an "unknown entity or character reference" error if we try to expand an entity or character reference that is not valid in HTML.

Example:

```
&thisdoesnotexist;
```

Example 4.9: Non-existing entity.

In the Example 4.9 the entity clearly does not exist. A common mistake in HTML documents is to use entities in CDATA attributes, without actually knowing it. This is especially common when we have URIs pointing to dynamic web pages, as we see in the Example 4.10.

[9]There are two other markup declarations which according to the SGML specifications should be valid in HTML; Short reference use declarations and link set use declarations. These are never used in HTML.

```
<a href=''index.php?category=3&documentID=2''>
```

Example 4.10: Unintentional reference to the non-existing entity called documentID.

In Example 4.10 we are actually trying to reference the non-existing entity &documentID, probably without knowing it[10].

Identify: The parse will notice the error when it tries to expand entities or character references in CDATA attributes or in #PCDATA nodes.

Correct: If we discover an unknown entity or character reference, we do not expand the reference at all, instead we keep the reference text.

4.3.6 Non-standard element type

Description: In the history of HTML, major browsers, such as Internet Explorer and Netscape Navigator, have changed the HTML DTD they use by adding new element types and attributes. This is very unfortunate, because it has created many separate branches of the HTML standard, where certain elements only occur in some of these branches. W3C has to take some of the blame for this since the HTML standard has evolved slowly. The work done by the W3C on HTML version 3.0 and HTML+ was discarded for the benefit of HTML 3.2 which contained more or less the same element types and attributes used in the major browsers at that time.

Examples Here are the proprietary elements for Internet Explorer and Netscape Navigator:

- bgsound (Internet Explorer)
- marquee (Internet Explorer)
- embed (Internet Explorer and Netscape Navigator)
- noembeded (Internet Explorer and Netscape Navigator)
- blink (Netscape Navigator)
- multicol (Netscape Navigator)
- spacer (Netscape Navigator)
- nobr (Internet Explorer and Netscape Navigator)

[10]The correct version is to encode the & character as the &amp; entity.

- wbr (Internet Explorer and Netscape Navigator)
- layer (Netscape Navigator)
- ilayer (Netscape Navigator)
- nolayer (Netscape Navigator)
- keygen (Netscape Navigator).

Identify This error is noticed when processing a start-tag or an end-tag token. If we cannot create an element representing the start-tag or process an end-tag since the element type is unknown, we have a "non-standard element type" error. This error will not only occur when a proprietary element type is used, but also when the generic identifier is misspelled.

Correct: The recommended response is to ignore the tag with the unknow generic identifier, and attempt to parse the nested data. The HTML code in Example 4.11 will give the same result as the HTML code in Example 4.12.

```
<BLINK>You can't beat our low low prices</BLINK>
```

Example 4.11: Propietary elements in the HTML code.

will be parsed as

```
You can't beat our low low prices
```

Example 4.12: Example 4.11 will be parsed as if the start- and end-tags of the proprietary element were missing.

The method shown in Examples 4.11 and 4.12 will allow our parser to degrade gracefully. It enables us to correctly parse the contents of an element pertaining to a new unknown element type that may be used in an HTML document.

Since many HTML documents use these proprietary element types, we should add support for them in our custom DTD.

4.3.7 Errors in attributes

There are several errors which could occur during the specification of attributes. They are:

1. A required attribute was not specified

2. A non-standard attribute was specified
3. Attribute specified is not a member of any group
4. An attribute was specified more than once
5. Misquoted attribute
6. Invalid attribute value.

We will now show how to handle each of these.

1. Required attribute not defined

Description: Some of the element types in HTML have required attributes[11]. These are attributes which instances of these element types would not have a meaning without[12].

Example:

```
<IMG alt=''My src attribute is missing''>
```

Example 4.13: IMG element missing a required attribute.

In the Example 4.13 the IMG element is missing the required attribute src.

Identify: We will notice this error when we have received a start-tag token from the lexical analyser and we are trying to create an element instance of its element type. The parser should check the DTD used when it is creating the instance, for required attributes and therefore notice the error.

Correct:We need to separate the absolutely required attributes for the type of user-agent we are developing, from attributes that are needed for other types of user-agents.

For example the element type BDO which represents bi-directional over-ride requires an attribute dir which represents the direction of the text contained. Without this attribute an element of this element type does not represent anything useful and should be discarded.

We might choose to elements missing required attributes to the DOM structure, so that they are available for post-processing. If we do that, we need to change our custom DTD so that the attributes whose value is required, are changed to an attribute whose value is implied. Otherwise we cannot represent the DOM structure as a document which is valid according to a known DTD.

[11]Which attributes required are defined in the DTD.
[12]The meaning should exist for all types of user-agents, not only visual.

In Table 4.1 we present a list of which required attributes in the transitional DTD[13] which are absolutely required for visual agents.

Element type	attribute	Absolutely required
BDO	dir	true
BASEFONT	size	false
AREA	alt	false
IMG[14]	src	false
IMG	alt	false
PARAM	name	true
APPLET	width	false
APPLET	height	false
FORM[15]	action	false
OPTGROUP	label	false
TEXTAREA	rows	false
TEXTAREA	cols	false
META	content	true
STYLE	type	false
SCRIPT	type	false

Table 4.1: Which attributes are absolutely required in HTML 4 transitional for a visual browser.

2. A non-standard attribute was specified

Description: As we mentioned in Section 4.3.6, the major browsers have added some element types and attributes to the DTD they use for interpreting HTML documents[16]. The attributes added usually give the authors more control over how the HTML documents are displayed, thereby making HTML more procedural.

Example: Both Netscape Navigator and Internet Explorer have added proprietary attributes to the BODY element type. Netscape Navigator has added marginheight and marginwidth, while Internet Explorer allows leftmargin, rightmargin, topmargin and bottommargin attributes to specify margins. The rec-

[13]Since the attributes used in the strict DTD are a subset of those used in the transitional, these are also in this table.

[14]The IMG element type is a special case in browsers. If there is a missing src attribute or the src attribute points to a non-existing image, an icon representing a broken image is displayed.

[15]A lot of web pages use javascript to interact with forms and do not want to submit them.

[16]It might be not be entirely correct to say that the browsers use a DTD, since they often use "hard-coded" values. But we can usually translate how a browser handles element types into a DTD.

ommended approach to page margins is to use stylesheets, but this approach was not possible when the attributes were first introduced.

Identify: We will notice this error when we have received a start-tag token from the lexical analyser and we are trying to create an element of its element type. When creating an element the parser should always check the DTD used to see if the attributes used in the start-tag exist, and therefore notice the error. This error will not only occur on proprietary attributes, but also attribute specifications where the attribute name is misspelled.

Correct: The response recommended by the W3C is to ignore the unknown attribute but interpret the rest of the tag. This is shown in Examples 4.14 and 4.15

```
<BODY marginwidth=''0'' marginheight=''0'' bgcolor=''white''>
```

Example 4.14: Proprietary attribute present in BODY element.

Example 4.14 will be parsed as if the HTML code was identical to that of Example 4.15.

```
<BODY bgcolor=''white''>
```

Example 4.15: Proprietary attributes are ignored.

Unfortunately, the approach suggested is not sufficient. A lot of the attributes used in HTML documents on the web are not part of the standard. Therefore, we need to support most of the proprietary attributes from Netscape Navigator and Internet Explorer. We do this by adding them to our custom DTD.

3. Attribute specified is not a member of any group

Description: It is valid HTML to shorten a name token group attribute specification to consist only of the attribute value, if the value is unambiguous. This is not supported by most browsers.

Example:

```
<P center>
```

Example 4.16: Valid use of a name token group attribute.

Example 4.16 is valid because the align attribute is the only name token group attribute for the element type P, which contains the value "center". The error occurs when an author shortens the name token group attribute wrongly, like the Example 4.17 shows.

```
<P senter>
```

Example 4.17: Invalid use of a name token group attribute.

In Example 4.17 the value "senter" is not a member of any name token group attribute for the element type P, and the attribute specification is therefore invalid.

Identify: We will notice this error when we have received a start-tag token from the lexical analyser and we are trying to create an element of its element type. The parser should check the DTD used when it is creating the element for wrongly shortened name token group attributes.

Correct: We solve this error by ignoring the attribute.

4. An attribute was specified more than once

Description: Sometimes an author makes a simple mistake and defines an attribute twice for a single element.

Example:

```
<P align=''center'' align=''left''>
```

Example 4.18: The align attribute for the P element is specified twice.

In Example 4.18 the align attribute for the P element is specified twice. Should the paragraph be centered or aligned to the left ?

Identify: This error will normally be noticed by the lexical analyser when it is creating a start-tag token where an attribute is specified twice. If the start-tag contains a shortened name token group attribute specification, it will not be noticed by the lexical analyser since it has no access to the DTD used. If this is the case, the parser will notice the error when it is creating an element based on the start-tag, since it checks the DTD for valid attributes matching the shortened name token group attribute specification.

Correct: Browsers and SGML parsers handle this error by keeping the first occurrence of an attribute and ignoring any duplicates. We propose the same method.

5. Misquoted attribute value

Description: In HTML documents attribute values must be quoted if they contain any other characters than a-z,A-Z,0-9, hyphens or periods. The best practice for authors is to always quote attribute values.

Example:

```
<IMG src=family.jpg alt=My family>
```

Example 4.19: Error caused by unquoted attribute.

The src attribute value in Example 4.19 is correct since it contains no other characters than the ones mentioned in the description. The alt attribute value in the same example, which is intended to contain the string "My family", contains whitespace data, and should therefore be quoted. It is not quoted and therefore invalid.

Identify: This error is recognized by the lexical parser when it is building the start-tag token. When it finds a character that is not valid in an unquoted attribute value, it immediately ends the unquoted attribute value. The rest of the tag is treated as normal for attributes. In the above example this means that the token will consist of 3 different attributes; src, alt and the shortened name token group attribute specification with the value "family".

Correct: Most browsers allow almost every character inside an unquoted attribute value. A space will of course begin a new attribute and is therefor not allowed. We propose to extend the characters allowed within an unquoted attribute value to all characters accepted within a quoted attribute, except whitespace characters. The lexical analyser should have a strict mode as well, which should be valid according to SGML specifications.

6. Invalid attribute value

Description: An invalid attribute value is an attribute value which is not valid according to the declared value of its attribute's specification in the DTD. In HTML we use 6 different declared values in attribute specification: CDATA, NAME, NUMBER, ID, IDREFS and name token group[17].

[17]CDATA can contain all data characters, NAME must begin with a-z or A-Z and can continue with a-z, A-Z, 0-9, hyphens, and periods, NUMBER can only contain one or more digits, ID is a NAME that must be unique, IDREFS is a list of NAME, and a name token group can contain several name tokens (which can contain a-z, A-Z, 0-9, hyphens, and periods).

Example:

```
<TEXTAREA rows=''fifty'' cols=''thirty''></TEXTAREA>
```

Example 4.20: Invalid attribute value.

In the Example 4.20 the declared value for the attributes rows and cols is NUMBER. In the example we are using character data, thereby producing this error.

Identify: We will notice this error when we have received a start-tag token from the lexical analyser and are trying to create an element of its element type. The parser should check the DTD used when it is creating the element to ensure that the attribute values are correct according to specifications.

Correct: We correct the problem by ignoring any attribute with invalid value and instead using the implied or default value. If the attribute in question is absolutely required, we should discard the element.

4.3.8 End-tag not specified

Description: As we have mentioned before, since HTML allows tag minimization, some elements may omit the start- and/or end-tag depending on their element type. Whether or not an element type instance can omit tags is defined in the element type's declaration in the DTD. This error occurs when the following conditions are met:

- An element N is an ancestor of element M. The element type of M does not have an omittable end-tag
- An end-tag token matching element N is processed while the element M is still open.

Example:

```
<DIV>
   <A href=''mylink.html''>Go to my sockpuppet site
</DIV>
```

Example 4.21: Missing end-tag for the A element.

In the Example 4.21 the DIV element corresponds to the element N and the A element corresponds to the element M from the conditions stated in the description above. We see that the element A is not closed before the end-tag of the DIV element is encountered.

Identify: This error will occur when the parser analyses an end-tag token and the conditions stated in the description are present.

Let us see how the parser notices this error.

The parser will when it is given an end-tag token, first attempt to match it with the current node. If it does not match, the parser will attempt to close the current node if its element type has an omittable end-tag and perform the same algorithm on the parent of the current node. If the end-tag of the current node is not omittable, and we have an ancestor which matches the end-tag this error occurs. If there is no such ancestor, an "end-tag for element not open" error occurs instead.

Correct: The most logical action when encountering such an error is to close all elements which are in the subtree of the element matching the end-tag. Unfortunately, this is not how browsers handle this situation.

Netscape Navigator and Internet Explorer handle this error by leaking the unclosed elements so that they are still open even after their parent is closed, hence creating overlapping elements.

After extensive testing I found that among all of the element types which could contain block elements only one is strong enough to permit this leaking, namely TD. All the others, including but not limited to P, H1-H6, DIV, SPAN and PRE, leak a subset of the open elements out depending on the element type. This subset has not been identified, but it appears to contain almost all element types. By making this group a subset of all elements, we can easily change the subset if an element type is found to have the property that it does not to leak.

Since the DOM structure does not implicitly allow us to use nested elements due to its parent-child relationships, we need to find a means of representing overlapping elements in SGML and therefore also HTML.

Overlapping elements in SGML have been analysed in [SMB94] and [HSM00]. Four different solutions, each with its own strengths and weaknesses, are suggested:

1. Use of the CONCUR feature

2. Use of milestone elements

3. Use of virtual elements

4. Use of fragmentation.

For our purpose fragmentation is the best solution. Fragmentation breaks up what is considered to be an overlapping element into multiple smaller elements in order for them to fit into a model which uses parent-child relationships. If we consider the first example in this section, it would be fragmented like shown in Example 4.22.

```
<DIV>
   <A href=''mylink.html''>Go to my sockpuppet site</A>
</DIV>
<A href=''mylink.html''>
```

Example 4.22: Fragmentation which occurs after handling Example 4.21.

The HTML code in Example 4.22 can be represented as part of a valid DOM tree. This part of the DOM tree is shown in Figure 4.1.

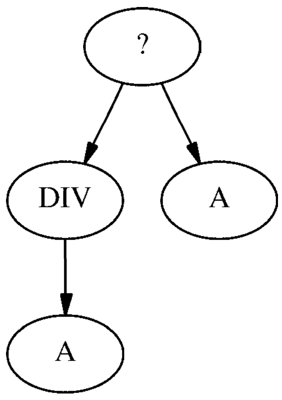


Figure 4.1: The DOM structure after fragmentation.

It is crucial that all attributes are copied to the generated fragmented element. If the element contains any ID attributes which uniquely identify the element, we have failed to fulfil our first goal of error-correcting since we are not allowed to contain two elements with the same ID in any SGML document. A solution to this problem is to use a custom DTD and change all ID attributes to NAME attributes thereby allowing duplicates, or to not copy any ID attributes to the fragmented elements. The fragmented element could be fragmented further if the same situation occurs.

A question that arises is, can and should we allow applications to access the individually fragmented elements or only the top element? Since the fragmentation process is transparent to the user, we feel that it should not be possible to change the individual fragmented elements. Instead, we propose an addition to the DOM model. The fragmented elements occur as ordinary elements in the DOM structure, but they all are marked in some way as fragmented elements. The original element should be marked as a fragment parent. From the fragment parent there is a linked list which reaches all fragmented elements of the parent. The fragmented elements can be accessed but not directly changed in any way. The fragment parent behaves like any other element in the DOM structure except when it is changed. When an attribute is changed it sends the change down the linked list, thereby propagating the change to all the fragmented elements. If a fragment parent is moved in any way, all descending fragment elements are removed, and new ones should be calculated from the parent's new location. It is however, not the the DOM structure's responsibility to calculate this.

A subset of the leaking behavior in browsers is documented within the source code of the Mozilla project[18][Org99]. It is called residual style handling. In residual style handling any unclosed style element with a closed parent element is pushed onto a stack. When an end-tag for an element on the stack is processed, the element on the stack is popped off. Before any element whose element type can contain style elements is added to the DOM tree, the residual style elements are added. Due to this information we should be careful when fragmenting, and only add the fragmented elements where they are allowed as children of the current node. This allows us to restrict our custom DTD a bit further, since we do not have to allow almost every element to contain almost any element.

To conclude; When an element is closed, we fragment any open element inside which is part of the subset of element types that can be leaked. This fragmentation could be done by using a stack with unclosed elements, and emitting the elements before any added element, as long as they can be a valid child of the current node. If the open elements' element types are not part of the subset of leaking element types, we should implicitly close these elements.

4.3.9 End-tag for element not open

Description: This error occurs if the parser receives an end-tag token, which does not match any open element.

Example:

```
<HTML>
  <HEAD><TITLE>The happy sockpuppet land</TITLE>
  </HEAD>
  <BODY>
    <H1>The story of sock puppets</TABLE>
```

Example 4.23: End-tag for non-existing TABLE element.

In Example 4.23 the end-tag for the TABLE element type does not match any open elements when it is processed. The open elements at this time are HTML, BODY and H1.

[18]Mozilla is an open-source browser which Netscape Navigator 6 is based on.

If we in the Example 4.23 had an element of an element type which did not belong to the subset of leaking element types, and therefore was closed implicitly, this error would occur if the author switched the order of two end-tags.

```
<DIV>
   <X href=''mylink.html''>Go to my sockpuppet site
</DIV></X>
```

Example 4.24: "End-tag for element not open" error occurs if we have a non-fragmented element and wrong order of end-tags.

If the element type X in Example 4.24 did not belong to the subset of leaking element types, it would implicitly be closed by the closure of its parent. This error would then occur when we process the matching end-tag.

Identify: The parser can notice this error in almost the same manner as in the last error("end-tag not specified" error). After trying and failing to process the end-tag in a valid manner, the parser attempts to find a matching element among the open elements. The open elements are all ancestors of the current node. This search starts with the parent of the current node and proceeds to its parent and so on. If no element matching the end-tag is found, this error occurs.

Correct: We solve this error by discarding the end-tag token. Internet Explorer and Netscape Navigator render this error as a line break. We do not want to introduce for example a BR element into our DOM tree to simulate this behavior, since it would create content not present in the HTML document. Neither is it the responsibility of the parser to enforce such a line break. The parser should rather in some way inform the module which does the presentation of what happened at this point, and let it decide whether or not to use a line break.

Since the primary way the parser communicates with the presentation module is the DOM structure, the information should be put there. We propose adding a processing instruction, which is a valid SGML construct, representing the event that occurred. This processing instruction should be added as a normal node child to the current node.

SGML does not enforce any special syntax on processing instructions other than that they should start with `<?` and end with `>`. XML, see Section 7.2 for more information on XML, requires a starting `<?` and an ending `?>`. To be compliant with both we use the XML version. We propose using a name to identify the creator of the processing instruction which should be placed right after the processing instruction start. The name should be followed by one or more whitespace characters and then followed by one or more attributes. This is shown in Example 4.25.

```
<?html-parser event=''unmatched end-tag'' elementType=''a'' ?>
```

Example 4.25: Added processing instruction in case of an ”end-tag for element not open” error.

A feature in browsers is that they define a list of similar element types and allow an end-tag of one element type to close an element corresponding to a similar element type. For example, they allow the end-tag of H1-H6 to close any H1-H6 element. Netscape Navigator 4.x even allows an end-tag representing a style element type to close any style element.

Let us demonstrate with an example.

```
<H1>Heading 1</H4>Heading is closed
```

Example 4.26: Similar elements close each other.

In Example 4.26 the H1 element is closed by the end-tag for the element type H4.

We therefore propose to define a list of similar element types. The list should only contain H1-H6, since not even Netscape Navigator 6 is compatible with the style element type matching of Netscape Navigator 4.x. This means that when we attempt to match an end-tag with an element type, we should also check the list of similar element types.

4.3.10 Violation of the element’s content model

Description: This is the most comprehensive error type. This error may occur in two situations.

The first situation is if we encounter an element or text node which the current node cannot contain, and the current node cannot omit its end-tag. The current node cannot contain any elements or text nodes which would cause it to violate the content model of its element type.

Violating a content model means placing an element or text node M as a child of element N when

- M is not present at all in the content model of N’s element type
- or M would break the count it is allowed to appear in the content model of N’s element type
- or M would break the order of the element types allowed in the content model of N’s element type.

The second situation in which this error can occur is when an end-tag which matches an element, is processed. If the content model of the matched element is not valid at the moment it is attempted closed, in effect it does not contain all the elements required, this error occurs. The second violation does not often occur in HTML.

Examples:

```
<TABLE>
  <TBODY>
  <TR>
    <TD>Product</TD>
    <TD>Prize</TD>
  </TR>
  <TR>
   <TD>T-shirt</TD>
   <TD>12 £ </TD>
  </TR>
 </TBODY>
 <P>As you can see from our price list ...
```

Example 4.27: Violation of the content model when adding a new element.

This error occurs in Example 4.27 because the element type P is not present at all in the content model of TABLE (which is (CAPTION?, (COL*| COLGROUP*), THEAD?, TFOOT?, TBODY+)). It may be agreed that the author's error is forgetting the end-tag of the TABLE element, but the parser cannot understand this.

```
Our products are :
<UL>
Sock puppets
Ballerina figures
</UL>
```

Example 4.28: Violation of the content model when closing an element.

Example 4.28 shows an attempt to make an unordered list. Unfortunately, the author has not included any list items (LI) within it, and has instead used text directly. The content model of UL is (LI)+.

The example thereby illustrates both situations which lead to this error. The first situation arises when we attempt to add a #PCDATA node to the UL element. The second situation occurs when we encounter an end-tag matching the UL element. Since the UL element does not have any LI elements as children, the content model is invalid.

Identify: If the first situation occurs, it will always occur during the processing of a start-tag of a valid element type. If the parser cannot add the created element in a valid way (see Section 3.2.1) , this error occurs.

If the second situation occurs, it will always occur during the processing of an end-tag of a valid element type. If the end-tag matches an open element, but its content model is not valid if closed at this time, the error occurs.

Correct: Let us first propose a method to solve the second situation, an invalid content model when closing an element.

There are a few elements types in HTML which require more than zero elements for the content model to be valid. These are HEAD, FRAMESET, TABLE, TBODY, TR, UL, OL, DL, OPTGROUP, FIELDSET and MAP.

In the second situation we have three choices:

1. Discard the entire element with any children (there exist probably none) and perhaps add a processing instruction. This processing instructions allows the presentation module to simulate common rendering in browsers, often a line break

2. Keep the element where it is and add one or more children to make it valid

3. Use a custom DTD which allows empty content models for these element types. This would be useful for exposing the DOM structure to any other application which might process it in some way.

We choose a different approach for each element. We never choose solution number 1. This is because we want to allow post-processing. For example it might be possible that an author wants to create a basic structure, and then use scripting to add new children, thereby making it valid according to one of the standard DTDs.

We will now describe for each element type how we propose to handle it.

HEAD: The HEAD element must always be present in an HTML document. It requires a child of element type TITLE to be valid. We propose to use solution 2, adding an empty child of element type TITLE to the HEAD element. If we encounter a TITLE element anywhere else in the document, it should change the value of our empty TITLE element.

FRAMESET, TABLE, TBODY, TR, UL, OL, DL, OPTGROUP, FIELDSET and MAP : We propose to use solution 3. We add to our custom

DTD that all of these element types do not require any children[19]. Therefore this error will never occur for these element types if we use the combination of two DTDs described earlier. By doing this we give the presentation module the responsibility of deciding how to present these uncompleted elements.

We will now continue with situation 1(an element or text node which the current node cannot contain), which may occur when we process a start-tag token which represents a valid element type.

Browsers generally cope with this situation in three different ways:

1. Using a custom DTD specification which is looser than the W3C standard DTD, thereby preventing this error from occurring
2. Moving the new element up the DOM tree in some way
3. Providing some special handling for an element if it contains a certain other element.

Let us view one example for each of the three solutions just to demonstrate that they are used in browsers.

```
<HTML>
  <HEAD><TITLE>Custom DTD demonstration</TITLE>
  <BODY>
    <LI>An unorderlist item
    <LI>And another one
```

Example 4.29: Common HTML code when using list items.

In the Example 4.29 above we see that LI tries to be a child of BODY. According to the W3C standard DTD LI can only be a child of UL. The UL elment cannot omit its start-tag and can therefor not implicitly be a parent of LI.

There are two possible changes to the DTD that could make the previous example valid. The addition of LI to the content model of the element type BODY, or allowing the tags of the element type UL to be omittable.

If the tags of the element type UL were omittable, Example 4.30 would be displayed identical to the previous example.

[19]If we do not do this, we would fail to fulfil the first goal of our parsing : *After the parsing process is finished we should have a DOM tree which is valid according to a known SGML DTD.*

```
<HTML>
  <HEAD><TITLE>Custom DTD demonstration</TITLE>
  <BODY>
    <UL>
      <LI>An unorderlist item
      <LI>And another one
    </UL>
```

Example 4.30: Is this example rendered in the same way as Example 4.29?

Examples 4.29 and 4.30 are in practice not displayed equally since Example 4.30 is displayed with an extra line break at the top of the list.

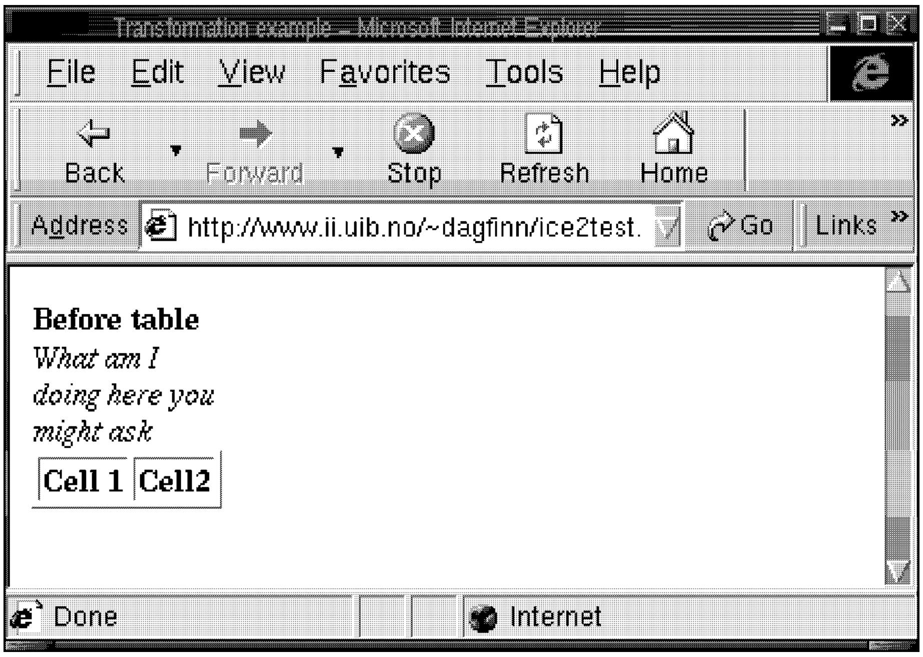
Therefore, we come to the conclusion that the element type LI has been added to the content model of BODY. It is most likely that it has been added to the inline entity, and is thereby a legal child to most element types with a non-empty content mode.

The second way browsers handle this error, moving the new element up the DOM tree, is demonstrated in Example 4.31.

```
Before table
<TABLE border=''1''>
   <P>What am I doing here you might ask</P>
   <TR><TD>Cell 1</TD><TD>Cell2</TD>
</TABLE>
```

Example 4.31: P element as a child to a TABLE element is not allowed.

The presentation of the HTML code in Example 4.31 is displayed in Figure 4.2.


Transformation example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home
Address http://www.ii.uib.no/~dagfinn/ice2test.
Go Links
Done
Internet

Before table
What am I doing here you might ask

Cell 1	Cell2

Figure 4.2: Illustration of how transformations in browsers work

As we can clearly see from Figure 4.2, the P element has been moved up the DOM tree and is now a sibling to the TABLE element. If the P element was replaced with an element type instance which had TABLE in its content model, the element would be placed as the parent of the TABLE element. It might be more natural for a human to move the P element so that it becomes a child of the first TD element, but remember that the P element is processed before we know that any TD element exists.

The third solution and final way in which browsers correct this error type can be illustrated with the Example 4.32.

```
<A href=''mysite.html''>To my site<A href=''home.html''>Home</A>
```

Example 4.32: Nesting A elements?

The element type A does not allow elements of its own element type as children, and it has no omittable end-tag. In Example 4.32 the solution is to close the first A element when the second is encountered. Therefore, the second A element is added as a sibling to the first A element. The end-tag at the end of the HTML code is parsed normally.

One might wonder why this situation cannot be solved by changing the custom DTD so that the element type A has an omittable end-tag. That would help in this case, but unfortunately the fragmentation described in Section 4.3.8 would then not be possible.

We will now propose a method of solving both the moving of elements up in the DOM tree and special handling of elements.

We propose to define a transformation language which allows us to specify both the conditions for the transformation, and how it should be carried out. The transformation language should have a simple syntax and yet be expressive enough to handle all our needs.

We choose XML as our syntax and define a DTD regarding which element types and their relationships should be present. This will be described further in Section 4.4, but for now we will give an example which is presented in Example 4.33.

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE transformation SYSTEM "transformation.dtd">

<transformation>
  <current type=''A''/> <!-- Prerequisite for the transformation to
                            begin. Here we say the current node
                            must be of element type A -->
  <new type=''A''/>     <!-- This is the new element type which we
                            have encountered which is not in the
                            DOM tree yet -->
  <action> <!-- What should we do if the prerequisite are meet-->
     <closeCurrent/> <!-- This closes the current node,
                          thereby making its parent the current
                          node -->
     <addChild parent=''current'' child=''new''/>
     <!-- Add the new element to the current element -->
  </action>
</transformation>
```

Example 4.33: Definition of a transformation.

The transformation in Example 4.33 will be executed if the current node at the time corresponds to the element type A and the new node corresponds to the same element type. The transformation executed is to close the current node, thereby making its parent the current node, and add the new node as a child to the new current node.

We will now mention some of the transformations that we propose:

- If an A element is the new element, and the current node is an A element, close the current node and add the new A element to its parent.

- If an AREA element is found anywhere except within a MAP element, attempt to find an open MAP element and add it as a child
- If a PARAM element is found anywhere except within an OBJECT or APPLET element, attempt to find an open OBJECT or APPLET element and add it as a child
- If a LINK element is found anywhere except within the HEAD[20] element, add it as a child to the element HEAD
- If any element except a CAPTION, THEAD, TFOOT, TBODY, COLGROUP, COL, TR, TH or TD element is found when the current node is an TABLE element, add it as a child to the parent of the TABLE element if it cannot contain TABLE, or add it as the TABLE element's parent if it can. If it is a sibling to the TABLE element, it should be placed before the TABLE element.
- If a STYLE element is found anywhere except within the HEAD element, add it as a child to the HEAD element
- If a META element is found anywhere except within the HEAD element, add it as a child to the HEAD element
- If a BASE element is found anywhere except within the HEAD element and a BASE element which is child of the HEAD element does not already exist, add it as a child to the HEAD element

We will now propose some changes to the W3C standard DTD, that will be placed in our custom DTD. It is not the intention that this custom DTD should always be used. Rather, we try to use the standard DTD for all new elements and end-tags, but if that fails we should attempt to use the custom DTD. This will be explained in more detail in Section 4.8. Anything allowed in the standard DTD should also be allowed in the custom DTD.

- Browsers generally allow all inline element types within the element type PRE. Therefore the preexclusion entity should be empty, and PRE should allow all inline element types
- The element type LI should be added to the inline entity since browsers generally allow to be used most places without a UL element as a parent
- The element types DD and DT should be added to the inline entity since browsers generally allow them to be used most places without a DL element as a parent
- The element type UL should allow all inline and block element types in its content model.

[20]There will always be one and only one HEAD element.

There are probably several other changes that must be made to the custom DTD to allow it to represent all HTML documents. Most of these changes will be discovered during the process of testing the parser, therefore the most important part is to make it easy to change this custom DTD.

4.4 Transformation system

In this section we will describe the transformation system mentioned earlier in this chapter.

The main goal of the transformation system is to allow us execute transformations of the DOM structure when a certain prerequisite is fulfilled.

A similar approach for the correction of invalid SGML documents is described in [Bir98]. This approach is not ideal for our purpose since it requires added markup by an editor, and it uses text-to-text transformations.

For our purpose we use DOM-to-DOM structure transformations and therefore the transformation engine should be closely connected to the DOM interface.

The entire system is portrayed in Figure 4.3. We see that the input from the parser to the transformation system is the DOM structure and the new element or text node. The transformation attempts to find a fitting rule, and if it finds one it applies a transformation to the DOM structure. Afterwards, the transformation system returns the DOM structure to the parser.

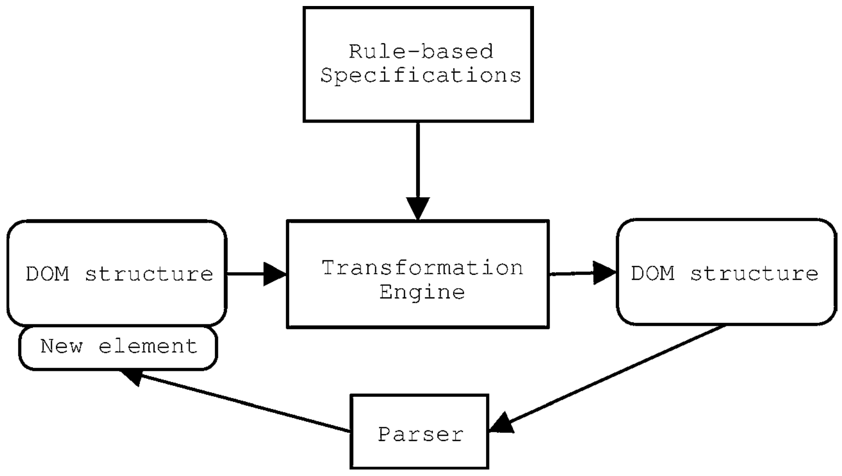


Figure 4.3: How the transformation system works.

As mentioned earlier we use XML for specifying the syntax of the rules. These rules are formalized in a XML DTD.

Each transformation definition in an XML document consists of two parts:

1. Prerequisites for the transformation to execute
2. The transformation to be executed if the prerequisites are fulfilled.

The prerequisites are specified using either an element of the current, new or prerequisite element type.

The element type "current" represents the element type of the current node. The element type "new" represents the element type of the node to be added. The element type "prerequisite" represents various other prerequisites for the transformation to take place. Elements from at least two of these element types should be present in a transformation specification, although this cannot be enforced using a DTD.

The transformations will be specified using the following element types: addChild, closeCurrent and createElement.

The element type "addChild" represents the insertion of a node into the DOM structure. The element type "closeCurrent" closes the current node and makes its parent the new current node. While the element type "createElement" allows one to create new elements and afterwards add them to the DOM structure using the addChild element.

This is just an outline of what the DTD would look like. It would probably contain many more elements.

We have created a DTD for this transformation system. It is included in Example 4.34.

```
<!-- DTD for transformation rules -->
<!ELEMENT tranformations (transformation)+ -- Top element -- >

<!ENTITY %@ conditions ‘‘current?,new?,prerequisite?’’>

<!ELEMENT transformation ‘‘description,%conditions;+,action’’
 -- Represent one transformation rule
    At least one of the conditions should be present -->

<!ELEMENT description (#@PCDATA)
 -- A description of the transformation rule -->

<!--START OF CONDITION ELEMENTS -->

<!ELEMENT current EMPTY
 -- What is the element type of the current node -->
<!ATTLIST current
  type    CDATA    #REQUIRED -- Name of the element type
                        Could be multiple elements separated by a |
                        All elements should be valid HTMLelements-->
```

```
<!ELEMENT new EMPTY
 -- What is the element type of the newly created element -->
<!ATTLIST new
  type    CDATA    #REQUIRED -- Name of the element type
                         Should be one of the HTML elements-->

<!ELEMENT prerequisite (attribute)*
 -- Another condition for the transformation to take place -->
<!ATTLIST
 target  CDATA    #REQUIRED  -- The element which is the basis of the
                             prerequisite. Could be one of new,
                             current, root,HTML, HEAD or BODY
                             Could also use XSL syntax f.ex.
                             root//TITLE , meaning the first TITLE
                             element which root is an ancestor to.
                             --
 isType  CDATA    #IMPLIED   -- Name of the element type
                          Could be multiple elements separated by a |
                          All elements should be valid HTMLelements--
 status  (exist|null)#IMPLIED -- Prerequisist fulfilled if target exist
                             and status is exist. Also fulfilled if
                             status= null and target does not exist -- >

<!ELEMENT attribute EMPTY
 -- Attribute used in prerequisite -->
<!ATTLIST
  attributeName  CDATA  #REQUIRED -- Name of the attribute --
  attributeValue CDATA  #IMPLIED  -- Value of the attribute -->

<!-- START OF ACTION ELEMENTS -->
<!-- More actions could easily be added-->
<!ENTITY %@ operations (addChild|createElement|closeCurrent)>

<!ELEMENT action ‘‘%@operations’’>

<!ELEMENT createElement (attribute)*
-- Create a new element-->
<!ATTLIST
  type  CDATA  #REQUIRED -- The element type --
  ref   ID     #REQUIRED -- The ID you can later refer to -->

<!ELEMENT addChild EMPTY
 -- Add a new child -->
<!ATTLIST addChild
  parent  CDATA  #REQUIRED -- The parent to add a child --
  child   CDATA  #REQUIRED -- Which child to add, probably new or
                              the id specified in createElement --
  before  CDATA  #IMPLIED -- Should this be added before any element
                             Reference through new,current ... -- >
```

Example 4.34: DTD for transformation system.

The transformation engine needs to parse the XML document which uses the DTD in Example 4.34 and to build an internal structure. This internal structure is used to easily match possible transformations given the input of the DOM structure and the new element.

One of the advantages of using an XML document to define transformations and not specifying the transformations within the source code, is that we are able to change the transformations while our application is running. Hence it is very easy to debug new transformations. If the transformations were included in the source code of the parser, we would have to recompile the parser and restart the application in order for the changes to take effect.

The transformation engine should use the methods exposed by the DOM interface to execute the transformations. If the transformation engine uses these methods it will be neither very large nor complicated because the basic methods for changing the DOM structure are already present.

4.5 Combining all the different methods

We have just shown how each error type should be handled, but it might be difficult to see how all these different error-correcting methods relate. Especially important is the order which the different methods are carried out.

To clarify this we specify how we should handle a start-tag, end-tag, data and document ended token. These tokens are passed to the parser from the lexical analyser.

The parser should be able to be configured to disallow fragmentation, residual style handling, similar element types, transformations and the use of the custom DTD. If all these features are turned off, the parser should function like the valid parser described in Chapter 3.

4.5.1 Handling start-tag tokens

When we receive a start-tag token, we should first attempt to create an element of it. The lexical analyser will have corrected some of the attribute value errors that may have been present in the start-tag.

We create the element by:

- Looking in the DTD the document uses (or the transitional DTD if it does not specify any) and attempts to find the element type the start-tag refers to. If this element type does not exist, the first attempt at creating the element fails. If it does exist, the element is created

- For each attribute in the start-tag token we attempt to find the corresponding attribute specification for the element type we just created an instance of. These attributes can be found in the DTD.

If the element was not created, or one or more attributes were not found, we use the same method as above, only using the custom DTD instead of the DTD the document uses.

If the element has not been created now, we discard the element and the start-tag token, and report the error.

If the element has been created, we attempt to add it to the DOM structure in a valid manner using the rules defined in the DTD the document uses. Note that we should add the residual style elements before this element if the suitable parent for the new element allows style element types in its content model. This should be done no matter what method is used to insert the element.

If we fail to add the element to the DOM structure in a valid manner, we attempt again, this time using the rules defined in the custom DTD.

The last alternative is to use the transformation system.

In order to use the transformation system we pass a reference to DOM structure, a reference to the current node in this DOM structure and the newly created element. The transformation system will return a boolean value informing us of whether a suitable transformation was found and executed. If this boolean value is false, we failed to insert the element and should report an error.

4.5.2 Handling end-tags tokens

When we receive an end-tag token, we first attempt to find which element type it matches.

We first check the DTD the document uses, for the element type, and if this fails we look in the custom DTD. If we find the element type in the custom DTD, we should ignore the DTD the document uses for the rest of the handling of this token.

If we cannot find the element type in any of the DTDs, we cannot process the end-tag. We might include a processing instruction informing the module which does the presentation of the occurrence of an end-tag to a non-standard element type, but this is not necessary since no browser alters its presentation because of this event.

Most likely we will find an element type which matches the end-tag. If in the method below find any element which matches the end-tag but has open elements ,which cannot omit its end-tags, in its subtree, we attempt to fragment or implicitly close the open elements using the approach described earlier in this chapter. To find the matching element we will:

1. Attempt to process the end-tag in a valid manner using the rules of the DTD the document uses. We use the algorithm proposed in Section 3.2.2. If we do not find a matching element, we proceed to step 2.

2. Attempt to process the end-tag in a valid manner using the rules of the custom DTD. When we try to match the end-tag with an element we also look in our list of similar elements for a match. If we do not find a matching elment, we proceed to step 3.

3. Start on the current node and check all its ancestors to see if any of them match the end-tag. If any of them match the end-tag, we close the element and either fragment or implictly close the open elements in its subtree. If we do not find a matching ancestor, we proceed to step 4.

4. If we come to this step we cannot process the end-tag. We generate a processing instruction informing of the event which has occurred, and add it to the current node.

We might also use the transformation system on end-tags, but we have yet to discover a situation where this is required to mimic browser precedence.

4.5.3 Handling data tokens

When we receive a data token, we use the method described in Section 3.2.3. The first time we use this method for a data token, we use the DTD the document uses. If this fails, we try again with the custom DTD. If both these attempts fail, our last resort is using the transformation system.

4.5.4 Handling the end of the document

When the document has ended we attempt to close the HTML element which is the root of the DOM tree. We simulate this by adding a virtual end-tag token matching the element type of each open element, starting with the current node[21] and continuing with its parent and so on. When we are handling these end-tags we do not have to fragment any elements, since there are no new elements they can leak into.

Any elements which do not have a valid content model either according to the DTD used in the document or the custom DTD, and can therefore not be closed, should be discarded. But as we remarked in Section 4.3.8, the custom DTD should be very lenient regarding required elements in the content model of an element type. This will allow post-processing of the DOM structure.

4.6 Correctable and uncorrectable HTML

A fundamental question which should be asked ourselves when correcting errors is; where do we draw the line between errors that we can and will correct, from the errors we might but should not correct ?

In order to reach our goals we need to place some restrictions on our parser:

- The DOM structure we create should contain enough information so that it can be displayed equally to how major browsers display the HTML document it represents. This means that we should correct errors which major browsers correct, but also we should not correct errors which major browsers do not correct

[21]If the current node is a #PCDATA node, we start with its parent.

- The DOM structure should always represent an HTML document which conforms to a DTD. However, we should not change this custom DTD to allow any combination of elements.

The reason why we should not do error-correcting which is not done in major browsers, is that if an author views an HTML document with a browser using the parser described, he should be certain that the HTML document should be display equally in all major browsers.

There seems to be a golden rule used in major browsers namely; *Thou shalt not add any element to a closed element.* The only placed were this rule could be broken using the method described, is in the transformation system. However, it could easily be enforced here by allowing only elements which are ancestors of the current node, as a parent of the new element.

Like most rules, this one has an exception. The HEAD element should be allowed to be the parent of new elements, even though it is closed. This should not be a big problem since there is always one and only one HEAD element, and most of the element types in its content model can not be placed under an element of any other element type.

4.7 A type of error-correcting we cannot copy

The method of error-correcting we have described in this chapter depends on one important property in major browsers; That a browser does not correct something which is valid HTML code. Unfortunately, this is not always true.

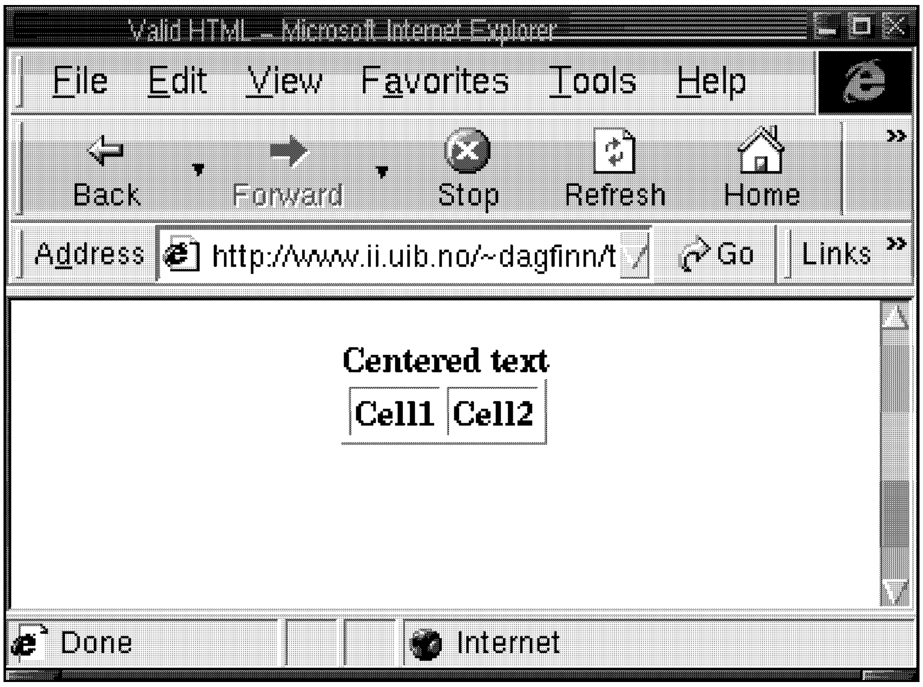
I have found one case where Internet Explorer does not interpret the valid HTML documents as it should.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
  <HEAD>
     <TITLE>Valid HTML</TITLE>
  </HEAD>
  <BODY>
    <P align="center">Centered text
    <TABLE border=1>
      <TBODY>
        <TR>
          <TD>Cell1</TD>
          <TD>Cell2</TD>
        </TR>
      </TBODY>
    </TABLE>
  </BODY>
</HTML>
```

Example 4.35: HTML document showing an uncorrectable error.

Example 4.35 is a valid HTML document. Since the element type P has an omittable end-tag and does not contain TABLE in its content model, the P element should be closed when the start-tag for the TABLE element is encountered. The TABLE element should be added as a child to the BODY element.

Unfortunately, Internet Explorer does not handle this HTML document as described above. Instead it guesses that since you created a P element aligned to the center of the page directly before the TABLE element, you would want to center the TABLE element as well, and therefore make the P element the parent of the TABLE element. This is shown in Figure 4.4.


Valid HTML - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home
Address http://www.ii.uib.no/~dagfinn/t Go Links
Done
Internet

Centered text

Cell1	Cell2

Figure 4.4: Internet Explorer showing the HTML document specified in Example 4.35.

This behaviour makes us decide between two of our goals.

1. The DOM structure should contain enough information for the presentational module to display the HTML document as Internet Explorer
2. Valid documents should not be affected by the error-correcting.

If we simulate Internet Explorer's behaviour, by adding TABLE to the content model of P, we will break the second goal stated. While if we treat the HTML document in a valid manner, we will not be able to display the DOM structure equal to how the HTML document is displayed in Internet Explorer.

In version 5.0 of Internet Explorer the HTML document in Example 4.35 is handled correctly if the strict DTD is used in the document. Netscape Navigator version 4.x and 6.0, and Opera 5.0 always handles this HTML document in a correct manner no matter which DTD is used.

Based on the fact that no other browser correct this error, we will not attempt to simulate corrections of valid HTML documents as done by Internet Explorer.

4.8 Summary, invalid HTML

We have in this chapter proposed a method of handling invalid HTML. We will now review the method with respect to the goals specified at the start of the chapter.

Goal 1: *The DOM structure should represent a document which is valid according to a DTD.*

The DTD which the document should be valid according to is :

- The custom DTD if it is used during the parse process
- The DTD used in the document, if the custom DTD is not used.

If we are not careful when defining the tranformation rules and how the fragmentation is executed, we might not have a valid document according to a DTD. We should try to validate the document produced with a validating SGML parser, so that we may become aware of any errors in our parser.

Goal 2: *The DOM structure should contain enough information for the presentational module to display the HTML document as Internet Explorer*

By adding processing instructions when an "end-tag for element not open" error occurs, we allow the presentational module to simulate common browser response in this situation.

The extended DOM structure we propose should be sufficient to hold all the information needed, but unfortunately we had to make a choice between simulating erroneous handling of valid documents or treating valid documents correctly. We chose to handle valid documents correctly.

Goal 3: *Valid documents should not be affected by the error-correcting*

Since we always attempt to use the valid method first, all valid documents should not be affected.

Goal 4: *It should be easy to change the rules of the error-fixing*

The transformation rules are very simple to modify.

The simplicity of changing the custom DTD and which element types to fragment, depend heavily on our implementation. We could use an XML document to specify these, thereby allowing easy modifications. Fragmentations could be reformulated as transformations, but we would need to extend the expressiveness of the transformation system to allow this. This extension would significantly complicate the transformation system and is therefore not recommended.

All in all we feel that the goals have been reached and we are pleased with the method described.

5. Statistics on syntactical HTML errors on the Internet

Until now there have been no major studies published on the amount of invalid HTML on the Internet. The online HTML validators mentioned in Section 4.3 might contain some statistics on the subject, but this data is not realistic because:

- Extremely few authors validate their HTML documents. The authors who do are usually more familiar with the HTML standards than those who do not. Validator statistics do therefore not accurately depict the diversity of HTML authors
- The process of validation usually consists of a HTML document being validated until it is correct. There needs to be some mechanism to exclude duplicates.

Since there were no results available I incorporated the task into my thesis. I will split the work done into two parts:

- A description of how the results were acquired
- A presentation of the results acquired.

5.1 How the results were acquired

The first problem that arose was finding a set of HTML documents which represented the diversity of authors on the Internet. Once we had found a set of appropriate HTML documents, each HTML document could be validates using a validating SGML parser.

Several alternatives for finding such a set of HTML documents and how to incorporate a validating SGML parser to test them, were investigated. The conditions to be present in an adequate solution were:

- It had to analyse a representative subset of the HTML documents on the WWW
- It had to be feasible in a relatively short period of time(one month maximum)
- It should be able to run with minimal user intervention.

We will now describe some of the alternatives investigated:

- Developing a crawler, not unlike the one described in [BP98], which :

  1. Retrieves a web page from the WWW
  2. Finds all anchors(links to other web pages) if the web page is an HTML document and adds the URI of these anchors, which is the anchor's href attribute, to a queue
  3. Validates the web page if it is an HTML document, using a validating SGML parser, and stores the result
  4. Removes one URI from the queue, goes to step 1 and downloads the URI.

  The problem with this approach is finding the starting URIs and ensuring that not only web sites with many ingoing links are validated. The implementation of this solution is non-trivial, for example how would the crawler know if a page is already validated.

- Incorporating a validating SGML parser in a web browser, such as IceStorm Browser, and validate every HTML document the user visits. Although it would capture a natural subset of the WWW, it would require a lot of users since most users only visit a very small subset of the WWW. Another drawback is that users are generally not very happy about things that go on behind their backs, and the extra computing would make a noticeable delay.

The solution I chose was a simple one, but it fulfilled all the conditions stated before.

5.1.1 The method used for testing

The program developed would do the following :

1. Get a URI to a HTML document
2. Download the HTML document
3. Validate it using a validating SGML parser
4. Store the result and go to point 1.

Our test data set consisted of 2.4 million different URIs. We will describe how we acquired this test data set in Section 5.1.2.

From actual tests we found that it took on average approximately 2 seconds to download a URL and 1 second to validate it. Thus the time used in a naive sequential computation would be :

$$2\,400\,000 * (2\ + 1)s = 7\,200\,000\ s$$

or 83 days and 8 hours of constant computation.

To improve the execution speed we needed to introduce some sort of parallelism. The computation at hand is clearly an embarrassingly parallel computation, as defined in [WA99], since each program instance is independent of the rest[1]. The test data set was partitioned, and each packet contained 8000 URIs.

Some external programs were used. They were:

- GNU's wget : This program was used to download HTML documents onto the local file system. It was modified so that there was a timeout of 10 seconds on the connection to the web server. This was done to ensure non-existing servers with a DNS[2] entry would not hang for the program for hours
- James Clark's lq-nsgmls :This validating SGML parser was used to validate the HTML documents
- SGML-lib from W3C : Contains all the HTML DTDs and other files needed for lq-nsgmls to function properly
- WDG's validate perl script : This is the perl script WDG uses for its online validator. It uses the lq-nsgmls for parsing. It was modified to return a binary code showing which errors types that were encountered in the HTML document validation process.

A bash script for the validation process was developed. The pseudocode is shown in Example 5.1.

```
acquire_packet()
for each uri in packet
    wget $uri $tempfile
    strip_zero_bytes $tempfile[3]
    validate $tempfile
end_for
```

Example 5.1: Pseudocode for the validation script.

[1]During the implementation of this problem each program instance used the same network connection and file server, but the effect of that was negligent as long as the number of validating program instances was kept reasonably low.

[2]Domain Name Server, translates Internet domain name into Internet protocol names. For example www.ii.uib.no into 129.177.16.249 .

The strip_zero_bytes command in the script in Example 5.1, was required because some HTML documents contained 100 kb of zero bytes in the middle of the document. The output of the script was redirected into a unique file for each instance of the script.

This script was distributed among 30 solaris sparc workstations which were all connected to the network at the University of Bergen. Each workstation executed 2 script jobs at a time, for approximately six hours during the night for six days[4]. When the processing was complete a Java program was made to analyse the data acquired.

5.1.2 The test data set

The set of HTML documents used in the validation process was acquired from the Open Directory Project. The Open Directory Project is a categorizing service for web sites, and its sections or categories are organized in a tree structure. For example, the category **Top: Regional: Europe: Norway: Weather** gives you a category with web sites containing information about the weather in Norway. ODP's most special feature is that it consists of 30 000[5] moderators and all the data is available in the RDF[6] format.

After downloading the data[7] the URIs were separated from the rest of the data with the simple Unix command shown in Example 5.2.

```
grep 'link r:resource=' content.rdf |
awk 'print substr($2,13,length($2)-15)'
```

Example 5.2: Unix command used to seperate URIs from the rest of the data.

After seperating the URIs from the data we ended up with 2.5 million URIs. We removed similar URIs[8] and URIs which did not point to HTML documents, and the count was down to about 2 400 000 URLs.

[4]The fact that the scripts generated 400 000-500 000 connections to other servers per night did not go unnoticed. During the first day of testing the central IT department thought a virus had occurred, and severely limited the Internet access of the local subnet in order to contain it.

[5]30 000 more or less active persons which are not financially compensated.

[6]Resource Description Framework, description available at `http://www.w3.org/RDF/` .

[7]The data took almost 1 gigabyte decompressed.

[8]There were several calls to scripts (such as cgi and php) which only differed in the parameters. From experience these scripts return almost the same HTML document structure and should therefore not be counted more than once.

5.2 The results of the testing

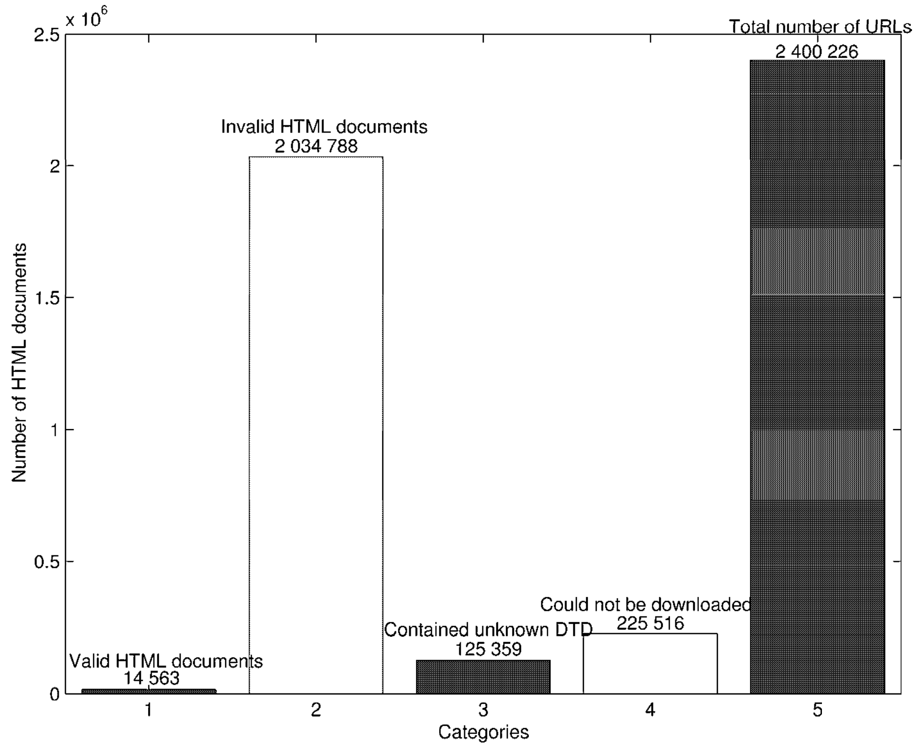


Figure 5.1: How many documents in each of the categories.

Figure 5.1 shows the result of the HTML validation process. Out of approximately 2.4 million URIs, 2 049 351 were attempted validated by a validating SGML parser. We were not able to download 225 516 of the remaining documents, either because the web server could not be found by the DNS, the web server was not available at the moment the URI was processed or the web server returned a response indicating that the HTML document was no longer present at the current address[9]. This does not imply that 9.4 % of outgoing links from the ODP are unavailable, since the downloading program was very aggressive and only allowed ten seconds from it attempted to connect to the web server until the HTML document had to be finished downloading. The crawling done by the google search engine [BP98] is more representative. It shows that 6.25 % of the attempted fetched documents were unavailable.

125 359 or 5.2 % of the HTML documents could not be validated because of

[9]This happened if the server returned a 401,403 or 404 error code. These error codes are defined in the HTTP protocol.

either an invalid document type declaration or because it used a custom DTD[10]. Although Section 7.2 of [W3C99] states that a valid HTML document declares what version of HTML is used in that document, I have chosen not to include HTML documents with an invalid document type declaration into the invalid group. The reasons for this is:

- It is difficult to separate an HTML document with a wrong document type declaration from an HTML document which uses custom DTD[11] based on the output from the validating SGML parser
- The validating SGML parser stops parsing when it does not find a correct DTD declaration, or finds a reference to a DTD it does not know. Therefore, any subsequent errors in the HTML document will not be noticed. This would affect the statistics we have made.

The conclusion based on the previous bar chart is decisive. Only 0.71 % of the HTML documents attempted validated are valid. This means that one in 141 HTML documents is valid. This shows the complete unawareness of the HTML standard among most HTML authors. We also know that a lot of the HTML code is generated from WYSIWYG programs, and we can see that even these fail to convey valid HTML to the WWW.

The next chart, Figure 5.2, shows how often the different errors occur in HTML documents.

[10]WYSIWYG(What You See Is What You Get) HTML editors usually use their own DTD which differs from the HTML DTD. This DTD often has a looser syntax than the HTML DTD.

[11]The HTML documents with a custom DTD were not be checked for validity because the SGML parser needs the DTD the document uses. The document type declaration declaration usually specifies a URI where the DTD can be downloaded, but this feature was not implemented because there is no easy way to know if the document at hand is an HTML document and not a document from another SGML application.

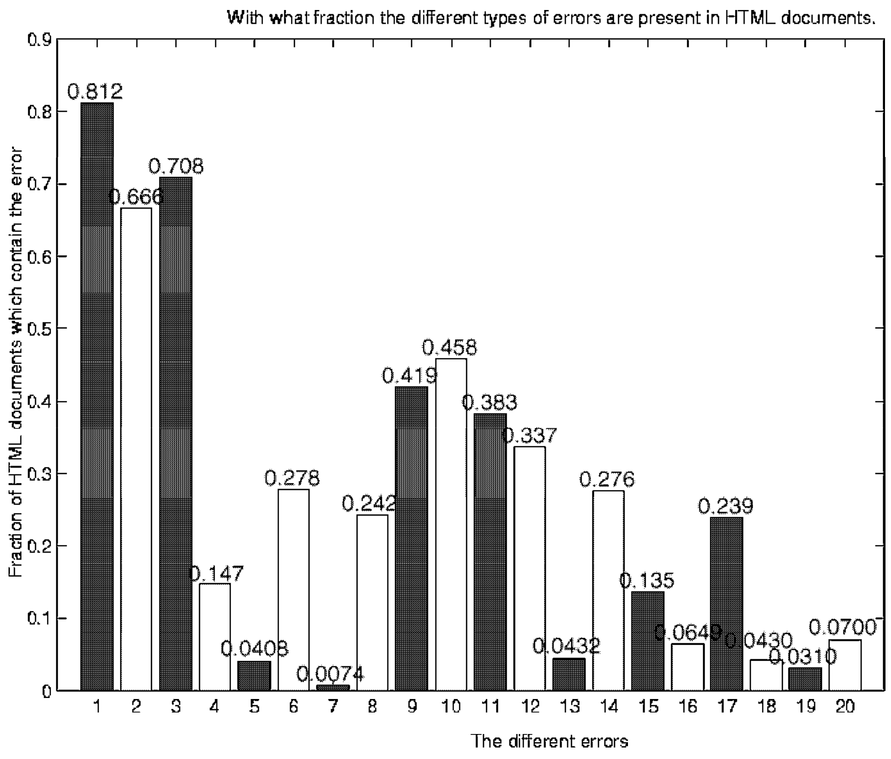


Figure 5.2: With what fraction the different errors occur in HTML documents

The errors types in Figure 5.2 are:

1. No DTD declared
2. Required attribute not specified
3. Non-standard attribute specified
4. Value not a member of a group specified for any attribute
5. An attribute was specified twice
6. An attribute value was not one of the choices available
7. Invalid attribute value
8. Misquoted attribute
9. Omitted end-tag.
10. End-tag for element not open.
11. Break of content model

12. Inline element containing block element
13. Start-tag omitted[12]
14. Unknown entity
15. Missing a required sub-element.
16. Invalid comment
17. Non-standard element
18. Start-tag omitted.
19. Premature end-tag.
20. Text placed where it is not allowed to be[13].

Although we have 20 different error types here, many of them can be mapped to the same fundamental error type. Error types 11, 12, 13, 15, 19 are all errors of the type "Violation of the element's content model" as defined in Section 4.3.10.

The most common error is to omit the DTD declaration. Although this is an error according to the HTML standard, the Annex K:Web SGML Adaptions of [ISO86] states that an SGML document instance does not need a declaration subset if it can be correctly parsed without one. It would be reasonable to say that a document downloaded with a mime type of text\html is expected to adhere to the latest transitional HTML DTD[14].

The errors which cause most problems for parsing HTML documents are errors 9,10,11,12,15,19 and 20. Upon closer analysis of the data collected, I found that the fraction of HTML documents containing one of these errors is 0.71. This is very serious because these errors can not be easily fixed.

Notice that 23.9 % of HTML documents contain non-standard elements and 71.8 % of HTML documents contain non-standard attributes. In order to fully support[15] HTML documents on the WWW, a browser needs to implement support for the non-standard elements and attributes.

[12]Only occurs when there is only one valid element type which the parent of another element type can be. For example the element type TBODY is the element type TR's only possible parent.

[13]Text should be placed in an element containing CDATA or #PCDATA in its content model.

[14]We choose the transitional because it includes the deprecated elements from earlier versions of HTML.

[15]With support we mean display as the author intended.

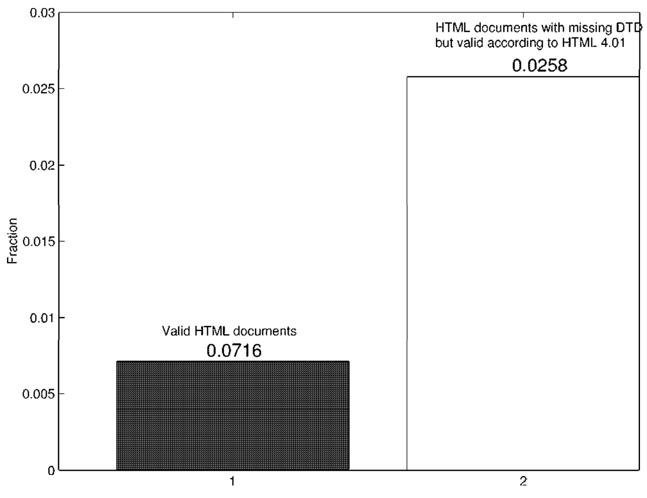


Figure 5.3: Fraction of HTML documents valid.

The Figure 5.3 shows that if we do not treat a missing DTD declaration as an error, 2.58 % of the HTML documents are valid. This is a significant increase, and most browsers do not separate between no DTD declaration and a valid one[16], since they are "hard-coded" into understanding the latest HTML DTD.

The last chart, Figure 5.4, shows how many different error types the HTML documents have. The error types are the ones presented under Figure 5.2.

[16]The newest versions of most browsers do a more strict parsing if the document specifies that it uses the strict DTD.

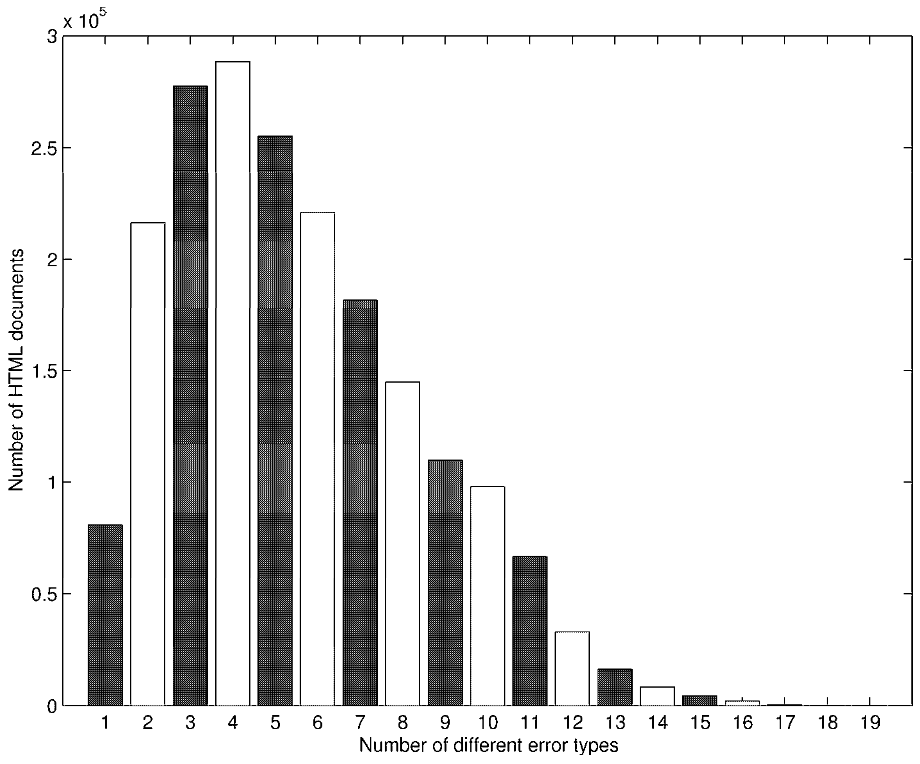


Figure 5.4: How many error types HTML documents have.

The Figure 5.4 shows that the most commonly occurring number of different error types occurring in an HTML document is 4. The median is 5 different error types and the mean is 5.2 different element types. Four documents contained 19 of a possible 20 errors, which must be considered quite a feat.

5.2.1 Conclusion based on validity tests

As we have seen there is little correlation between the official HTML standard from W3C and the de-facto standard of the WWW.

The validation done here raises the question if the HTML standard is of any use on the WWW. It seems very odd to have a standard that only 0.7 % of the HTML documents adhere to. Although most current browsers parse valid documents correctly, they allow almost any error in HTML documents. With the growth of the web to embedded systems, these also have to replicate the error-correction of the browsers to allow a homogeneous view. We hope this thesis can help developers of new browsers do the "required" error-correcting in a systematic way.

6. Regression testing of the IceStorm Browser

The previous chapters have shown how invalid HTML can present a serious problem for browsers. Testing how well a browser corrects errors in HTML documents is therefore a significant part of browser development. Because the source code is complex and changes in it may have unforseen consequences, tests need to be rerun whenever the source code of a browser changes to ensure that the previously corrected errors are still corrected.

This chapter describes the work done while developing a program for testing new versions of the IceStorm Browser, a product of Wind River Systems Incorporated.

This chapter will consist of:

- A short presentation of the IceStorm Browser
- The goal of the program
- The design of the program
- The implementation of the program
- Test results.

6.1 The IceStorm Browser

The IceStorm Browser is a Web browser component written in Sun's Java technology. IceStorm Browser can be integrated into any Java application. The IceStorm Browser provides a very high level of modularity and extensibility to web applications, and the browser component can be used for displaying various content types in any Java-written application.

HTML rendering is the central part of the product. The latest version of IceStorm Browser contains an HTML4 Pilot with fully compliant standards support (including HTML 4.0, CSS2, XML/CSS rendering, DOM).

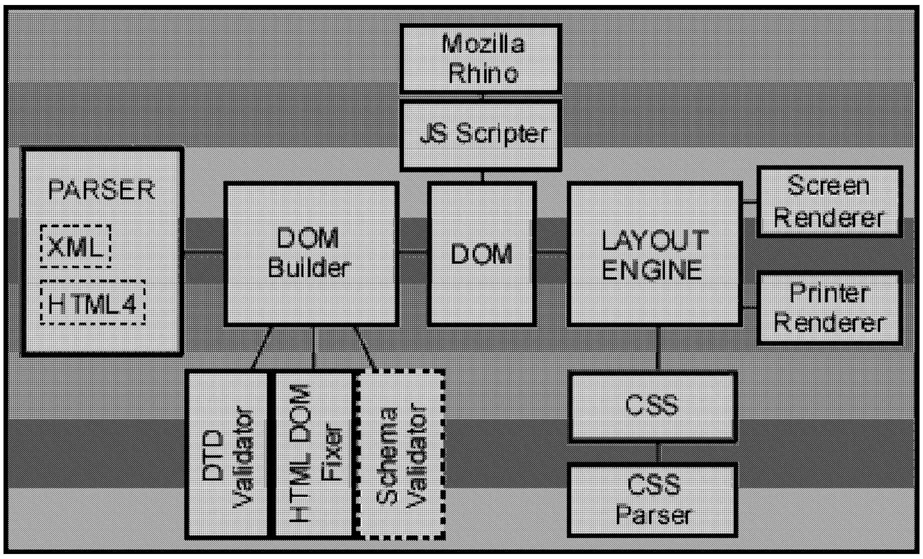


Figure 6.1: Architecture overview of the HTML rendering component in IceStorm Browser.

In the IceStorm Browser, the DOM builder, as shown in Figure 6.1, will attempt to create a valid DOM structure according to a DTD. If any errors are present in the HTML document being processed, the HTML DOM fixer will be called upon.

While the DOM structure is being "built", the layout engine will display a page representing the structure.

6.2 Goals

Traditionally three separate quality assurance mechanism for the IceStorm Browser have been used.

1. Simple test cases to which one knows the correct output. These tests are usually done by the developers. The test cases include nested tables, simple style sheets and other short examples

2. Feedback from customers and users telling the company which web pages display incorrectly in the IceStorm Browser

3. Customers specifying a list of URLs which must be displayed correctly in the IceStorm Browser.

Note that customers and users generally define "displayed correctly" as displayed like Internet Explorer and not necessarily as the standards(HTML and CSS) specify.

I was part of a team testing URLs for different customers(point number 3). This work consisted of:

- Opening a URL in IceStorm Browser, Internet Explorer and Netscape Navigator
- Finding differences between the data displayed in the different browsers, with focus on differences from the IceStorm Browser. This difference was then classified by the degree of it
- Through normal user interaction, checking to see if transition to other linked pages functioned correctly and that dynamic elements of the web page worked as intended
- Finding which elements and parts of the HTML document that caused any anomalies in the two previous points.

Most anomalies discovered were caused by either invalid HTML documents or javascripts.

During the process of bug fixing, the goal is of course to enable web pages which were displayed incorrectly, to be displayed correctly. Unfortunately, a change in the source code has a slight chance of having the effect that a few web pages previously displayed correctly, are now displayed incorrectly. This is because of the complexity of a code trying to mimic ad-hoc parsing rules needed because of the precedence of earlier browsers. Because of this phenomena, extensive testing must be done in advance to releasing a new version of the IceStorm Browser.

The primary goal of the program developed is to automate, to some extent, the testing of new versions of the IceStorm Browser. The program should be able to run in batch mode, meaning without user interaction once started. It should report URLs which are being displayed differently with the new version of the IceStorm Browser, than they were with the old version of the IceStorm Browser[1]. We often call this form of testing regression testing. Bennetan defines in [Ben95] regression testing to be:

> *"Part of the test phase of software development where, as new modules are integrated into the system and the added functionality is tested, previously tested functionality is re-tested to assure that no new module has corrupted the system."*

Note that regression testing does not test if the changed functionality is correct, only if the changes in the source code has altered the functionality that was present before the change.

As a secondary goal, it should be easy to extend the program to include other tests for equality than those implemented during the development phase.

[1]In order to decide if it is displayed correctly in the new version, a human user must analyse it more closely.

6.3 Design

We will in this section describe the major elements of the regression testing system.

The five main elements of the system are:

- StormTestBase
- Comparer
- ComparerAccesser
- DataAccesser
- VersionTester

6.3.1 StormTestBase

The interface of the IceStorm Browser has and will evolve and change with time. To cope with this we need a uniform way to access all the different IceStorm Browser versions. To do this we use the StormTestBase class.

The StormTestBase provides the methods we need to call on the IceStorm Browser instances during the execution of the program. To make an instance of the IceStorm Browser we create an instance of the class ice.storm.StormBase. This is the class that is encapsulated by StormTestBase.

The most important methods in the StormTestBrowser are:

void addPropertyChangeListener (java.beans.PropertyChangeListener listener)
Adds a listener to the StormTestBase. This listener is called when the IceStorm Browser undergoes a change in its properties. For example the web page has finished loading.

org.w3c.dom.Element getElementRoot ()
Returns the document element which is the root of the DOM tree.

String getVersionDate ()
Identifies the IceStorm Browser by returning when it was compiled.

void renderContent (String location)
Loads and displays the URL represented by the location. This call is asynchronous. To find out when the page is fully rendered you need to attach a listener via the addPropertyChangeListener method

void setUpFrame(int x, int y, int width, int height)
Creates the container in which the IceStorm Browser is displayed. The parameters define the position and size of the container.

If the interface of the IceStorm Browser and especially the class ice.storm.StormBase changes, we may have to subclass the StormTestBase and change the implementation of the methods which depended on the old interface. The name of this new class will be StormTestBase<version>.

6.3.2 Comparer

We call the part of the program which does the actual comparing of the two StormTestBase instances a Comparer.

A Comparer has methods for returning an id, name and a description of how it works. It has one crucial method

double compare(StormTestBase compare, StormTestBase compareTo)

This method assumes that the StormTestBase instances are finished rendering the web page. It compares the two StormTestBase instances and returns a double value from 0.0 to 1.0. A score of 0.0 represents no likeness at all, and 1.0 represents that they are identical.

How the StormTestBase instances are compared is defined by the class implementing the Comparer interface. Two different Comparers have been implemented; the DOMComparer and the LayoutComparer. These will be described later in this chapter.

The system can be expanded by adding new Comparer classes.

6.3.3 ComparerAccesser

In order to add a new Comparer without having to change the existing code of the VersionTester, a ComparerAccesser was made. The ComparerAccesser's responsibility is to find all existing Comparers and create instances of them.

The most important methods in the ComparerAccesser are:

Comparer getComparerById (int id)
Returns the Comparer with a specified id. If a Comparer with the specified id does not exist, null is returned.

Comparer[] getAllComparers ()
Returns an array with all Comparer available

One ComparerAccesser, XMLComparerAccesser, was implemented which uses XML documents to find Comparer classes.

6.3.4 DataAccesser

To acquire and store data we introduced a DataAccesser. A DataAccesser is an interface defining methods to obtain the test data and save the test results.

The test data is encapsulated in a class called TestData. It consists of:

- A unique id.
- Two IceBrowserData instances which represent the IceStorm Browser to be compared. The information in these objects is used to instantiate the two StormTestBase objects.
- An array of URLData objects each containing information about one web page to be tested
- An array specifying which Comparer classes should be used to compare the IceStorm Browsers.

A test score is the comparison between two IceStorm Browsers done by one Comparer on one web page. The test scores are encapsulated in a class called TestScore. It consists of:

- The name and id of the Comparer used.
- Two IceBrowserData instances representing the IceStorm Browsers compared
- A URLData object representing the web page used for the comparison
- The result of the comparison represented by a double value from 0.0 to 1.0 .

The most important methods of the DataAccesser are:
TestData getNextTestData ()
Returns the next TestData set. If there is no more TestData it returns null.

boolean hasMoreTestData ()
Are there any more TestData available?

void markFinished (TestData testData)
Marks the TestData as processed.

void saveTestScore (TestScore testScore)
Saves a TestScore instance.

void saveTestScores (TestScore testScore)
Saves multiple TestScore instances.

The method of which the DataAccesser acquires and stores the data, could be very diverse. Two different DataAccessers were implemented, SQLDataAccesser which uses a database and an XMLDataAccesser which uses XML documents. These will be described more closely in the section where we deal with implementation details.

6.3.5 VersionTester

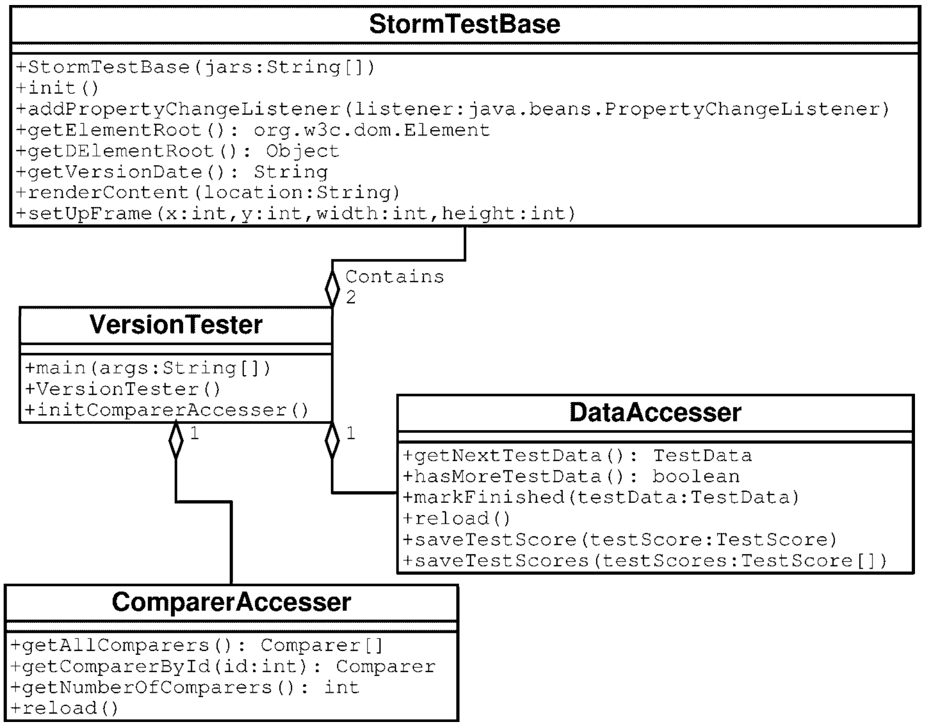

StormTestBase
+StormTestBase(jars:String[])
+init()
+addPropertyChangeListener(listener:java.beans.PropertyChangeListener)
+getElementRoot(): org.w3c.dom.Element
+getDElementRoot(): Object
+getVersionDate(): String
+renderContent(location:String)
+setUpFrame(x:int,y:int,width:int,height:int)
Contains
2
VersionTester
+main(args:String[])
+VersionTester()
+initComparerAccesser()
1
1
DataAccesser
+getNextTestData(): TestData
+hasMoreTestData(): boolean
+markFinished(testData:TestData)
+reload()
+saveTestScore(testScore:TestScore)
+saveTestScores(testScores:TestScore[])
ComparerAccesser
+getAllComparers(): Comparer[]
+getComparerById(id:int): Comparer
+getNumberOfComparers(): int
+reload()

Figure 6.2: UML diagram showing the relations between the major parts of the program.

As seen in Figure 6.2 the central element in the system is the VersionTester. It binds all the other classes together and starts the different operations of the system.

The VersionTester first creates instances of a DataAccesser and a ComparerAccesser. If one of these instances is not correctly initialized, the program stops immediately.

After the initialization of the DataAccesser and ComparerAccesser the VersionTester will go into a loop which exits when there are no more test data. For each iteration of the loop the program will:

- Get the next test data
- Create two instances of the StormTestBase, each of them representing one of the IceStorm Browsers to be tested
- Then do the following for each URL in the URLData array:
  - Render the URL in both of the StormTestBases

  - Run the appropriate Comparers with the StormTestBases as parameters
  - Log the result of the comparisons using the DataAccesser

- Mark the test data as tested using the DataAccesser
- Dispose of the StormTestBases.

6.4 Implementation

This section will describe the algorithms and non-trivial implementation details.

6.4.1 StormTestBase

The greatest challenge in the StormTestBase was instantiating two IceStorm Browsers with the same class name at the same time. Before we can look closer at how this was solved, we need to know how Java loads classes from class files.

Each version of the IceStorm Base is distributed in several Java archive (jar) files. A jar file is a compressed archive file containing Java class files. In order to run the IceStorm Browser normally from the command line, we need to include its jar files to a system variable called CLASSPATH. Once this variable is defined we can run the IceStorm Browser with the command shown in Example 6.1

```
java ice.browser.Main
```

Example 6.1: How to start the IceStorm Browser from the command-line.

When java is executed by giving a class name on the command line it searches the files in the CLASSPATH for a match. When it has found a matching class, the class is loaded using the system classloader. The main method of the class loaded is then executed[2]. The system classloader is responsible for loading all classes in the CLASSPATH as well as the standard Java classes (in effect, classes in the java.* packages) as noted in [MD97]. If two different IceStorm Browsers with the same package and class names are added to the CLASSPATH, by including all their jar files to the CLASSPATH variable, we do not know which version will be executed.

Fortunately, there is another way to load classes. Using classes which extend the abstract ClassLoader class we can create several classloaders each with their own namespace. This allows us to use the same class name several times in our program, as long as the classes are loaded using different classloaders. Thereby we could instantiate different objects which contain the same class name, but use different implementations, such as two different IceStorm Browsers. This is illustrated in Figure 6.3.

[2]If the class has not got a main method with the correct signature, an error is reported and the Java Virtual Machine terminates.

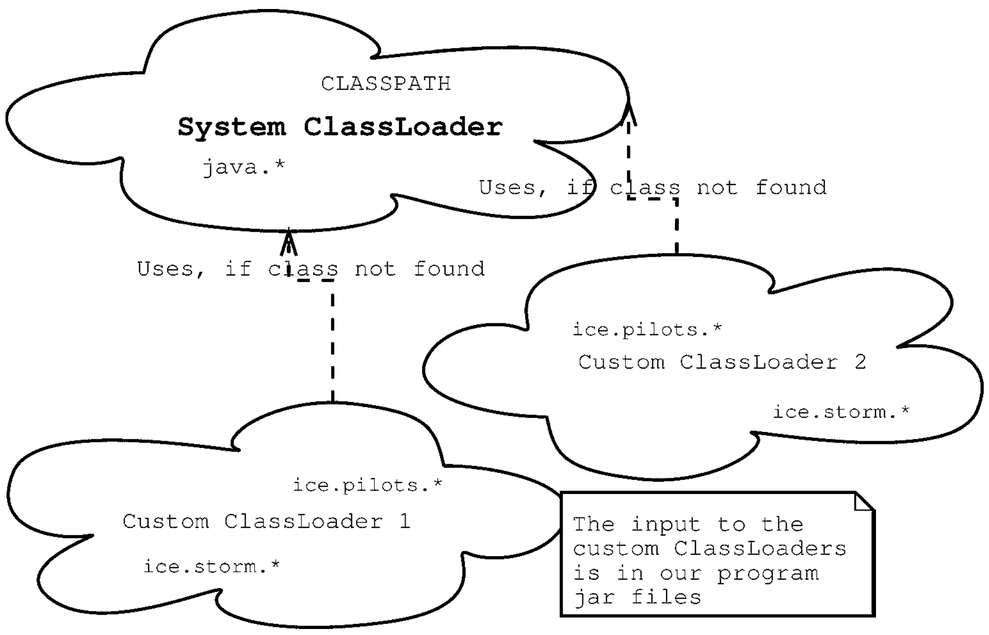


Figure 6.3: The ClassLoaders may contain different classes with the same name.

Although this approach is very convenient for our goal, it has some drawbacks:

- Because there is no common interface to the IceStorm Browser, we have to treat all instances of the classes loaded as Objects[3] instead of the classes they really are instance of. Any method call has to be done through the Java Reflection classes
- When using the Java Reflection classes all type matching occurs at runtime. This means severe loss in performance
- The code will be much more complex when using the Java Reflection classes. This complicates the development phase
- When using the Java Reflection classes, many programming errors will not be noticed during compiling, only when the program is run.

To show the complexity added by the Java Reflection classes I have made two examples which do the same thing. Example 6.2 use the syntax we normally use in Java when we create an instance of a class and then call a method on this instance. Example 6.3 however use the Java Reflection classes to do the same.

[3]All classes in Java extend the Object class, therefore we can treat instances of any class as if they were instances of the Object class.

```
StormTestBase stormbase = new StormTestBase()  ;
stormbase.renderContent(``http://www.ii.uib.no'') ;
```

Example 6.2: Code required using normal Java syntax.

We see that by using the standard Java as we do in Example 6.2 we use only two lines of code to make an instance of the StormTestBase class and call a method upon it.

In Example 6.3 we assume that when we use the Java Reflection classes we have a classloader called classloader that has access to the classes we need.

```
Class StormTestBaseClass =
classloader.loadClass(``com.icesoft.automation.StormTestBase'') ;
Object stormbase = StormTestBaseClass.newInstance() ;
Method renderContentMethod =
  StormTestBaseClass.getMethod(``renderContent'',
  new Class[] {classloader.loadClass(``java.lang.String'')} ) ;
renderContentMethod.invoke(
stormbase, new Object[]{``http://www.ii.uib.no''}) ;
```

Example 6.3: Some of the code required using Java Reflection classes.

Example 6.3 does not include the necessary error handling. The following exceptions can occur, and must be handled: ClassNotFoundException, NoSuchMethodException, SecurityException, IllegalAccessException, IllegalArgumentException and InvocationTargetException.

We see that there is a big difference in both size and complexity of the code.

6.4.2 DOMComparer

The DOMComparer implements the Comparer interface. Its job is to compare the work done by the different parsers of the IceStorm Browser. This comparison is done by comparing the DOM structures the parsers construct

The algorithm used for comparing DOM structures is a variation of the longest common subsequence problem as described in [THCR97]. We will describe the algorithm and the changes made to it.

First we need to define some terms used in the longest common subsequence problem.

Given a sequence $X = [x_1, x_2, \ldots, x_m]$, another sequence $Z = [z_1, z_2, \ldots, z_k]$ is a subsequence of X if there exists a strictly increasing sequence $[i_1, i_2, \ldots, i_k]$ of indices of X such as for all $j = 1, 2, \ldots, k$ $X_{i_j} = Z_j$.

A common subsequence Z of X and Y is a sequence that is a subsequence of both X and Y.

The problem of the longest common subsequence is, given sequences X and Y, what is the longest common subsequence.

For example, given $X = [A, B, C, D, E]$ and $Y = [B, A, B, E]$ the longest common subsequence is $[A, B, E]$.

In our problem instance we convert the DOM structure into sequences by using a recursive method. The DOM tree in Figure 6.4 is converted into the following sequence of strings: [<HTML, <HEAD, <TITLE, mytitle, </TITLE, </HEAD, <BODY, <P, first paragraph, </P, </BODY, </HTML].

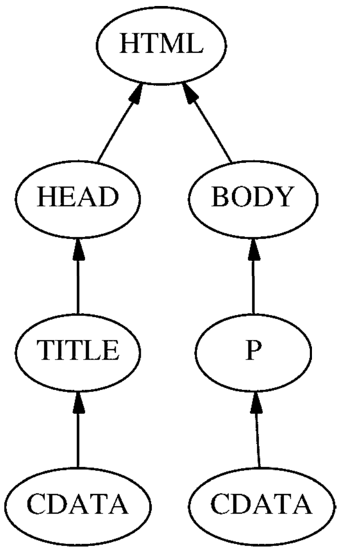


Figure 6.4: The DOM tree which is converted to a sequence.

The longest common subsequence problem can be solved using a dynamic programming algorithm. Dynamic programming is characterized by the fact that it combines the solutions to subproblems to solve the main problem. We can solve the problem by using the following recursive formula, where $c[i,j]$ denotes the length of the longest common subsequence of X_i and Y_j.[4]

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

The length of X and Y is m and n respectively. We see that each calculation is only dependent on three other calculations and the base cases are when i or j is

[4]X_i is the same as $[x_1, \ldots, x_i]$.

equal to 0. In a 2 dimensional array (which c is), a calculation is only dependent on the cells North, West and North-West of it. Therefore we fill in zeros in the cells c[i,j] where $i = 0$ and/or $j = 0$. Afterwards we start calculating $c[1,1]$ to $c[1,n]$ in that order, before we continue on to the next row by calculating $c[2,1]$ to $c[2,n]$ and so on.

The pseudocode in Example 6.4 shows how to calculate the table c.

```
for(i=0 to m)
  c[i,0] = 0
for(j=1 to n)
  c[j,0] = 0
for(i=1 to m)
   for(j=1 to n)
     calculate c[i,j]
```

Example 6.4: Pseudocode used for calculating the table c.

By using this technique we the calculate the table c as shown in Table 6.1. Numbers in bold indicate equal elements between the sequences.

i	j	0 y_i	1 B	2 A	3 B	4 E
0	x_i	0	0	0	0	0
1	A	0	0	**1**	1	1
2	B	0	**1**	1	**2**	2
3	C	0	1	1	2	2
4	D	0	1	1	2	2
5	E	0	1	1	2	**3**

Table 6.1: Calculation of the c table.

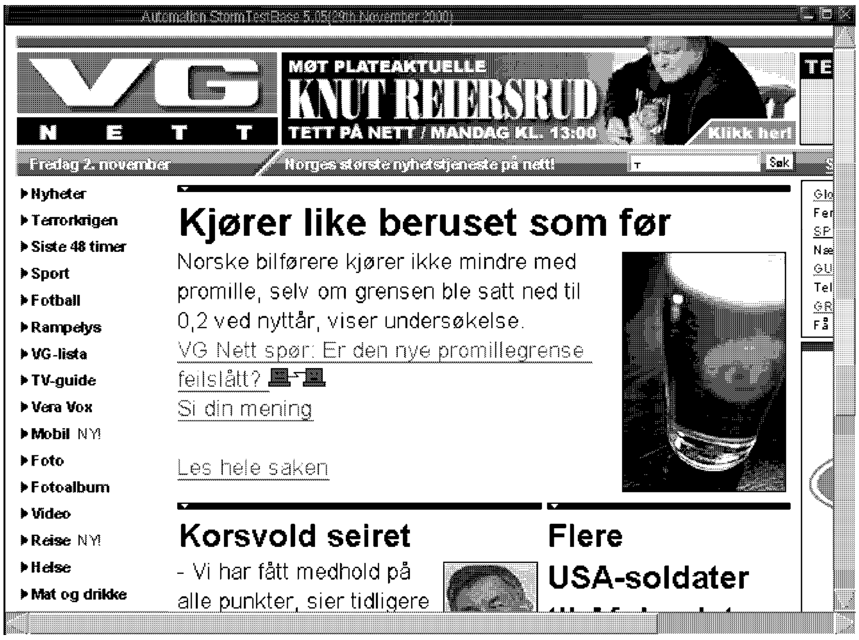
When we have finished calculating the c table we have the length of the longest common subsequence in cell $c[m,n]$. Using the c table we can in $O(m+n)$ time find the longest common subsequence by backtracking from $c[m,n]$.

When we have found the longest common subsequence in the DOMComparer of the sequences representing the DOM structure, we walk linearly through each of the original sequences reporting differences from the longest common subsequence. Normally there should be no differences, but if some elements have been discarded or have been handled differently by the parsers, this will be noticed and reported.

The work done to calculate one cell in the table c is one string comparison and one integer comparison if the strings are not equal. We optimize this code by first checking to see if the length of the strings are equal. Since the strings are very short (long blocks of text are hashed), we can assume the time to compare them is a constant.

Since there are m*n cells and to calculate each cells takes constant time, the time to compute the c table is $O(mn)$.

We will now present an example of how the testing worked in practice. In Figure 6.5 and 6.6 we present two screen shots showing to different instances of the StormTestBase executed at the same time.


Automation StormTestBase 5.05(29th November 2000)

MØT PLATEAKTUELLE KNUT REIERSRUD TETT PÅ NETT / MANDAG KL. 13:00 Klikk her!

Fredag 2. november — Norges største nyhetstjeneste på nett!

- Nyheter
- Terrorkrigen
- Siste 48 timer
- Sport
- Fotball
- Rampelys
- VG-lista
- TV-guide
- Vera Vox
- Mobil NY!
- Foto
- Fotoalbum
- Video
- Reise NY!
- Helse
- Mat og drikke

Kjører like beruset som før

Norske bilførere kjører ikke mindre med promille, selv om grensen ble satt ned til 0,2 ved nyttår, viser undersøkelse.

VG Nett spør: Er den nye promillegrense feilslått?

Si din mening

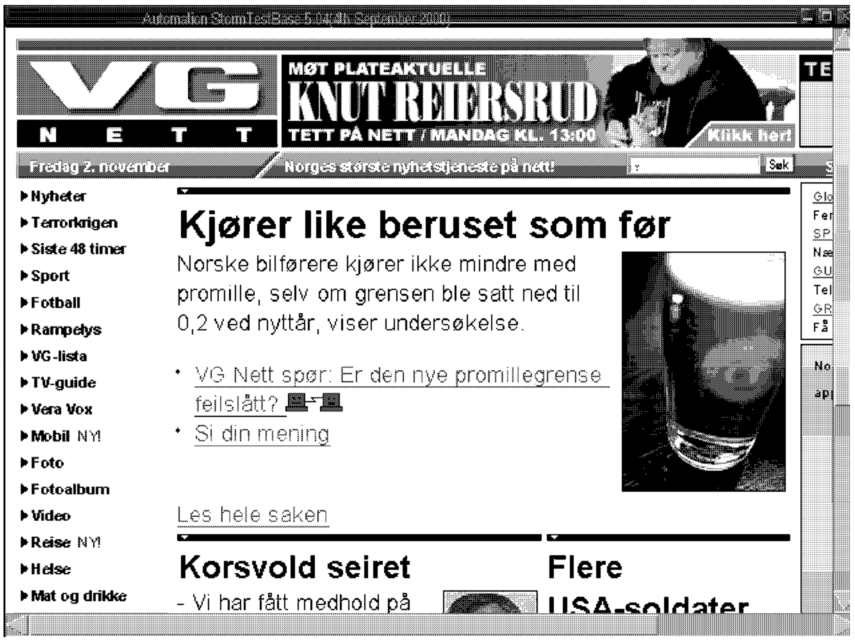
Les hele saken

Korsvold seiret

- Vi har fått medhold på alle punkter, sier tidligere

Flere USA-soldater

Figure 6.5: StormTestBase using IceStorm Browser version 5.05



Kjører like beruset som før

Norske bilførere kjører ikke mindre med promille, selv om grensen ble satt ned til 0,2 ved nyttår, viser undersøkelse.

- VG Nett spør: Er den nye promillegrense feilslått?
- Si din mening

Les hele saken

Korsvold seiret

- Vi har fått medhold på

Flere USA-soldater

Figure 6.6: StormTestBase using IceStorm Browser version 5.04

We can see from the Figures 6.5 and 6.6 that the 5.05 version (Figure 6.5) does not display bullets in front of the anchors at the middle of the page. These are displayed in version 5.04 (Figure 6.6). The HTML code causing the difference is invalid, it contains an LI element without a encapsulating UL element.

The log from the test shows us the UL element was not in the longest common subsequence, but it existed in the sequence from the 5.04 version. Therefore we can draw the conclusion that version 5.04 implicitly creates a UL element if an LI element is not encapsulated by one.

The result of the DOMComparer is

$$\frac{\text{length of longest common subsequence}}{\text{length of longest sequence}}$$

Since the length of longest common subsequence is always equal or less than the length of the longest sequence, this result will be from 0.0 to 1.0.

6.4.3 LayoutComparer

The LayoutComparer implements the Comparer interface. Its job is to compare the web page presented to the user.

Instead of using an image comparison algorithm such as the one described in [KC94], we develop an algorithm for comparing the corresponding elements in the two web pages.

The first problem we needed to solve was to find the corresponding elements. The parsers might not return the same DOM structure, and therefore we use the longest common subsequence algorithm described in Section 6.4.2. After we found the longest common subsequence we do a walk of the DOM structure that is equal to how the sequences were generated, find the elements in the longest common subsequence in both DOM structures and compare them.

This comparison is not based on the elements' content, only its bounding box. The bounding box is acquired through a CSS (Cascading Style Sheets) property of the element.

The result of the comparison between two bounding boxes depends on the point of origin and the size of the bounding boxes.

The width of the bounding box is set to be more important than the height of it, but this could easily be changed.

6.4.4 XMLComparerAccesser

The XMLComparerAccesser implements the ComparerAccesser interface. Its responsibility is to find Comparer classes and make instances of these classes.

As its name imply, the XMLComparerAccesser fulfil its responsibility by using XML. We look closer at XML in Section 7.2, but for now we only need to now that XML is a simplified version of SGML.

We will show the DTD used, and an example of a XML document conforming to this DTD.

```
<!ELEMENT stormComparers (stormTest)+
  -- Top element -->

<!ELEMENT comparer (java-class,description)
 -- One Comparer class -->
<!ATTLIST comparer
 id   ID    #REQUIRED -- Unique ID --
 name CDATA #REQUIRED -- Name      -->

<!ELEMENT java-class  (#PCDATA) >
<!ELEMENT description (#PCDATA) >
```

Example 6.5: DTD used by the XMLComparerAccesser.

As we can see from Example 6.5, the DTD used is very small. A comparer element contains two attributes, id and name, as well as two children, java-class and description. A java-class element contains the name of the package and the name of the class in which the Comparer is defined in. This is seen in Example 6.6.

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE StormComparers SYSTEM "stormComparers.dtd">
<stormComparers>
  <comparer id=''1'' name=''DOMComparer''>
    <java-class>com.icesoft.automation.test.DOMComparer</java-class>
    <description>This is Comparer compares the parsed
            DOM structure</description>
  </comparer>
  <comparer id=''2'' name=''LayoutComparer''>
    <java-class>com.icesoft.automation.test.LayoutComparer</java-class>
    <description>This is Comparer compares the CSSBox of correpsonding
            elements</description>
  </comparer>
</stormComparers>
```

Example 6.6: XML document conforming to the stormComparers DTD defined in Example 6.5.

The Example 6.6 shows the XML document used for specifying the two Comparers developed.

After parsing through the XML document, the XMLComparerAccesser uses the Java Reflection classes to create instances of all the Comparers. This allows us to add a new Comparer by creating its Java class and then changing the XML document in Example 6.6. We do not need to recompile any of our source code for this change to take effect.

6.4.5 SQLDataAccesser

The SQLDataAccesser implements the DataAccesser interface. Its responsibility is to retrieve test data and store test results.

To fulfil its responsibility the SQLDataAccesser communicates with a database, using JDBC. The JDBC API is a Java API primary used for accessing relational database systems. The JDBC API allows a uniform way of accessing virtually all database systems. This is done by using a JDBC driver for each of the database systems.

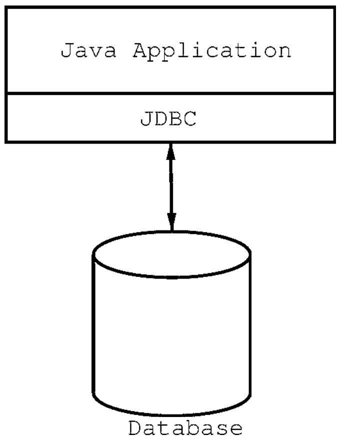
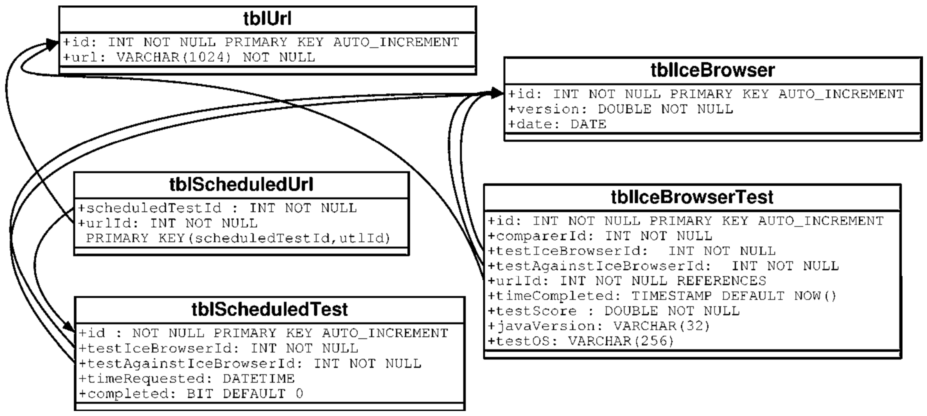


Figure 6.7: JDBC using a two tier architecture for data access.

Figure 6.7 shows that it is the JDBC driver which communicates with the database.

In the SQLDataAccesser the JDBC driver is given as a parameter, thereby allowing an easy transition of the test data to another database system.



```
tblUrl
+id: INT NOT NULL PRIMARY KEY AUTO_INCREMENT
+url: VARCHAR(1024) NOT NULL

tblIceBrowser
+id: INT NOT NULL PRIMARY KEY AUTO_INCREMENT
+version: DOUBLE NOT NULL
+date: DATE

tblScheduledUrl
+scheduledTestId : INT NOT NULL
+urlId: INT NOT NULL
 PRIMARY KEY(scheduledTestId,utlId)

tblIceBrowserTest
+id: INT NOT NULL PRIMARY KEY AUTO_INCREMENT
+comparerId: INT NOT NULL
+testIceBrowserId:  INT NOT NULL
+testAgainstIceBrowserId:  INT NOT NULL
+urlId: INT NOT NULL REFERENCES
+timeCompleted: TIMESTAMP DEFAULT NOW()
+testScore : DOUBLE NOT NULL
+javaVersion: VARCHAR(32)
+testOS: VARCHAR(256)

tblScheduledTest
+id : NOT NULL PRIMARY KEY AUTO_INCREMENT
+testIceBrowserId: INT NOT NULL
+testAgainstIceBrowserId: INT NOT NULL
+timeRequested: DATETIME
+completed: BIT DEFAULT 0
```

Figure 6.8: Tables in the database.

Figure 6.8 shows the tables used in the database. The test data is retrieved primarily from the tblScheduledTest table and is stored in the tblIceBrowserTest. The lines in the figure indicate relations between the tables.

6.4.6 XMLDataAccesser

The XMLDataAccesser implements the DataAccesser interface. Its responsibility is to retrieve test data and store test results.

Using the XMLDataAccesser we can create tests without having access to a database. It uses XML for the test data, while the test results are written as ordinary text files to allow easier processing with unix commands.

```
<!ELEMENT stormTests (stormTest)+
--Root element-->

<!ELEMENT stormTest
 (iceBrowserData,iceBrowserData,(acceptComparer)+,(url)+ )  >
<!ATTLIST stormTest
  id          ID #REQUIRED  --unqiue-->

<!ELEMENT iceBrowserData (stormtestbase-class,(jar)+) >
<!ATTLIST iceBrowserData
  id          CDATA   #REQUIRED --not unique for all tests
  version     CDATA   #REQUIRED
  dateCompiled CDATA   #REQUIRED >

<!ELEMENT stormtestbase-class (#PCDATA)
--Name of the StormTestBase class that should be used-->
<!ELEMENT jar                (#PCDATA)
-- One of the jar files used by the IceStorm Browser-->

<!ELEMENT url (#PCDATA)>
<!ATTLIST url
  id    CDATA   #REQUIRED  --Is not unqiue for all tests--
  value CDATA   #REQUIRED  --The URL-->
<!-- Set this to -1 to allow all tests -->
<!ELEMENT acceptComparer (#PCDATA) >
<!ATTLIST acceptComparer
  id    CDATA #REQUIRED  --Is not unqie for all tests-->
```

Example 6.7: DTD used for acquiring the test data.

As shown in Example 6.7, a stormTest element represents a test data set. It consists of an id, two iceBrowserData, which Comparers to use and the URLs to be tested.

In Example 6.8 we will show a document which conforms to this DTD.

```
<!DOCTYPE StormTests SYSTEM "stormtests.dtd">
<stormTests>
  <stormTest id="1">
    <iceBrowserData id="1" version="5.04" datecompiled="unknown">
      <stormtestbase-class>com.icesoft.automation.StormTestBase
      </stormtestbase-class>
      <jar>ib_jars/ib504/ib5core.jar</jar>
      <jar>ib_jars/ib504/ib5http.jar</jar>
      <jar>ib_jars/ib504/ib5js.jar</jar>
      <jar>ib_jars/ib504/ib5ref.jar</jar>
      <jar>ib_jars/ib504/ib5https.jar</jar>
    </iceBrowserData>
    <iceBrowserData id="2" version="5.05" datecompiled="unknown">
      <stormtestbase-class>com.icesoft.automation.StormTestBase
      </stormtestbase-class>
      <jar>ib_jars/ib505/ib5core.jar</jar>
      <jar>ib_jars/ib505/ib5https.jar</jar>
      <jar>ib_jars/ib505/ib5js.jar</jar>
      <jar>ib_jars/ib505/ib5ref.jar</jar>
    </iceBrowserData>
    <acceptComparer id="-1" />
    <url id="1" value="http://www.google.com" />
    <url id="2" value="http://www.fark.com" />
  </stormTest>
</stormTests>
```

Example 6.8: XML document conforming to the stormTests DTD defined in Example 6.7.

The Example 6.8 represents a test with IceStormBrowser version 5.04 and IceStormBrowser 5.05. All Comparers available should be used, and the URLs to be tested are `http://www.google.com` and `http://www.fark.com`.

6.5 Summary, regression testing

We have in this chapter presented a system for regression testing of a browser. The system has a high degree of modularity and can easily be changed to allow new Comparers, DataAccessers or ComparerAccessers.

7. The future of data presentation on the Internet

In this chapter we will take a closer look on how data will be presented on the Internet in the future. It will not be a definitive guide, but mostly based on the standards and specifications which the W3C has already or plan to, release.

We will first present several shortcomings of HTML stated in literature, before we will present different technologies/standards which solve some of these shortcomings. Of course these new technologies/standards might produce new problems which we will briefly analyse.

7.1 HTML limitations

We will begin this chapter with several quotations about HTML.

> *"In the frenzy of the growth, much of the discipline and good practice of the mature SGML world has been lost, and browser developers have added additional features to the markup language such as new tags[1] and new semantics for tags. (...) Common web practice is to hide any syntactical problems detected by the browser and thus the reader is not aware that a page being browsed is not always faithful to the original authored document."*
>
> — *Roger Price and David Abrahamson [PA00]*

> *"Many commonly-used utilities produce invalid HTML or introduce vendor-specific extensions. Moreover, most users do not validate their HTML source code and browsers do not object to invalid*

[1]The quoted persons have here used the term tag when the term element type is more appropriate.

HTML, (...) This makes it especially difficult to get consistent results between Web browsers and across platforms."

—Michel Goossens [Goo98]

"Most information on the Web is presented in loosely structured natural language text with no agent-readable semantics. HTML annotations structure the display of Web pages, but provide virtually no insight into their content."

—Rober Doorenbos, Oren Etzioni and Daniel Weld[DEW97]

"HTML 4 is a powerful language for authoring Web content, but its design does not take into consideration issues pertinent to small devices, including the implementation cost(in power, memory, etc) of the full feature set."

—W3C[W3C00a]

"HTML only allows a few elements, that is those that are explicitly defined in the standard. Whenever some authors' needs exceed the capabilities of the elements already defined in HTML, a different approach has to be used."

—Paolo Cincarini, Alfredo Rizzi and Fabio Vitali[CRV98]

As we can see there are many objections against HTML. Some of them arise from the fact that HTML became extremely popular and outgrew its original purpose, others are based on the lack of control the W3C had on the companies implementing the standards.

We will now describe the problems one by one in the order of seriousness:

1. Too many invalid HTML documents exist, therefore parsing is non-trivial
2. We cannot expand the set of element types, and the existing element types cannot properly describe all data[2]
3. Most HTML documents do not separate between content and presentation
4. Not enough structural information and meta data is available through the element types provided
5. HTML is too complex for most small devices
6. Although HTML is an SGML application, most user-agents do not adhere to the SGML standard
7. It is difficult to understand what the semantics of an element type is just by looking at the name of the element type.

[2]There is for example no elegant way of displaying mathematical expressions and no means what so ever to display vector graphics using HTML without using plugins.

7.1.1 Invalid HTML

As the statistical analysis in Section 5.2 show, there exist very few valid HTML documents on the WWW. This is mostly caused by error-correcting browsers and authors lacking knowledge of the HTML standard. This severely complicates the development of any user-agent.

To solve this problem we need to simplify the syntax of HTML and require that user-agents do not attempt to correct an invalid document, or specify the way a user-agent can correct an invalid document. A user-agent should always tell the user what is wrong with the document being viewed.

On the other hand, the fact that browsers do error-correcting has drawn many authors to the Internet that would not otherwise be active in the field. We must remember that not all people read standards and specifications.

7.1.2 Fixed set of element types

The second most serious shortcoming of HTML is that it provides a fixed set of element types, and that these element types cannot properly describe all the data we wish to publish on the Internet. This set cannot be expanded except when new versions of HTML is published. Even then, authors need to be careful using the new element types, since older browsers do not support them.

If we need to display data that cannot be displayed using the element types available, we need to use Java applets or some other plugin, or define another markup language. We will look closer at both these methods later in this chapter.

On the other hand, one might say that it is because of this limited set of element types HTML has become so popular. HTML is easy to learn and is fairly intuitive. Extending the element type set would make HTML less intuitive and more complex.

We acknowledge that HTML is suitable for its use, but we cannot describe complex structures or very big documents with it. Therefore, we need a more expressive language for dealing with such documents.

7.1.3 Not enough separation between content and presentation

Although some element types used for presentational purposes are deprecated in HTML 4, they are still, and will in the nearest future, be used by authors. The approach recommended by the W3C involves using a stylesheet language for presentation.

The only way to solve this problem is to remove the element types which are used for presentational purposes from the markup language, and use stylesheets or equivalent instead.

7.1.4 Not enough structural information and meta-data available

There are too few element types in HTML with semantics that define the structure of the document. Although we can produce an HTML document with chapters and sections through the heading element types(H1-H6), this is seldom done. Instead presentational element types such as IMG are used to separate logical parts of the document.

Meta-data is definitional data that provides information about a document. HTML contains one element type which provides meta-data only, namely META. The problem is that it has not been specified how this element type should be used, except for when overwriting http headers.

The lack of structural information and meta-data severely limits the possible usage of HTML, such as the existence of automated agents, since there is little semantics to extract from HTML documents.

7.1.5 HTML too complex for small devices

Small devices like cell phones, PDAs and wrist-watches generally have little processing power and memory. Since layout calculations (especially tables) and HTML parsing are a relatively complex set of operations, small devices might have problems supporting the whole HTML specification. The transitional DTD contains 91 element types which should be supported, as well as proprietary element types.

To solve this problem we need to create a subset of the element types which are to be used and simplify layout algorithms and parsing.

7.1.6 Ambiguous element type names

HTML contains several element types the names of which do not inform an author of the element type semantics. For example what do you think TT, KBD, DFN and BDO represent[3]?.

A balance must be struck when naming element types. A long name will give more intuitive understanding of the semantics, but it will also lead to larger documents and more tedious work for authors who hand-code HTML.

7.2 XML

XML is not an alternative to HTML, but a simplified version of SGML.

XML is a restricted form of SGML and is specified in [BPSM98]. It is said that XML offers 90 % of the expressibility of SGML with only 10 % of the complexity. Some of the key design goals of XML that were reached are:

[3]The element types mentioned are abbreviations for Teletype(which means display with a mono spaced font), keyboard(indicates text to be entered by the user), define instance of a term, and bi-directional override(allows right to left characters).

1. XML should be compatible with SGML
2. It should be easy to create programs that process XML documents
3. It should be easy to create XML documents
4. The number of optional features should be as low as possible, ideally zero.

XML is a product of SGML's immense expressive powers. Not many languages can define a subset of themselves by using themselves.

All markup languages that are applications of XML have not got any features of markup minimization which SGML defines (DATATAG, OMITTAG, SHORTTAG and SHORTREF). All XML document also have the feature NAMECASE GENERAL as no, meaning that all names are case sensitive.

There are other differences between XML and SGML, these are covered in detail in [Cla97].

XML makes it very easy to create new element types. Element types are defined in a DTD just like in SGML. In an XML document we can use elements from several DTD, thereby allowing us to extend the element type set available in an XML application.

We demonstrate how we can create a DTD for a class in a programming language. This DTD is presented in Example 7.1

```
<!ELEMENT class (attribute*,method*) >
<!ATTLIST class
 lang  CDATA  #IMPLIED -- which programming language used --
 visibility (public|private|protected) public
--visibility is public if none specified-->

<!ELEMENT attribute EMPTY --attribute to the class-->
<!ATTLIST attribute
 name  CDATA #REQUIRED
 type  CDATA #REQUIRED
 visibility (public|private|protected) public
--visibility is public if none specified-->

<!ELEMENT parameter EMPTY --Parameter to a method-->
<!ATTLIST parameter
 name  CDATA #REQUIRED
 type  CDATA #REQUIRED>

<!ELEMENT method parameter*>
<!ATTLIST method
 name CDATA #REQUIRED
 returnType CDATA #IMPLIED -- void if none specified --
 visibility (public|private|protected) public
--visibility is public if none specified-->
```

Example 7.1: XML DTD for classes.

If we were to describe some of the methods on our parser from Section 3.2 using the DTD in Example 7.1 we would get the XML document shown in Example 7.2.

```
<?xml version=''1.0''?>
<!DOCTYPE class SYSTEM ''class.dtd''>
<class lang=''java'' visibility=''public''>
   <attribute name=''sgml_lex'' type=''SGMLlex''/>
   <method name=''primaryCallback'' returntype=''boolean''
       visibility=''public''>
      <parameter name=''primaryToken'' type=''PrimaryToken''/>
   </method>
</class>
```

Example 7.2: XML file which describes the parser in Chapter 3

This is only a small example, not a definition on how to create XML documents and DTDs for classes. It could easily be extended with element types for modules, constructor methods, inheritance and so on.

Although the XML document gives a human reader quite a lot of information if the element type names and attributes are chosen wisely, it provides no semantics for a computer. There is also no intuitive method of displaying it, since a computer program does not have any information on how to do this.

A well-formed XML document is an XML document which adheres to all the basic XML syntax rules, such as every element has a start- and an end-tag, attributes are quoted, elements are correctly nested and no superfluous markup exist. To allow parsing of an XML document it must be well-formed. An XML document can be valid if it uses a DTD and conforms to it.

XML documents can be displayed by using a stylesheet language, such as CSS[BLLJ98] or DSSSL[fG96][4], but unfortunately these stylesheets are not expressive enough to display non-textual data.

The most serious problem in using XML on the WWW, is the existence of multiple similar DTD. This will severely limit the usefulness of XML because we cannot easily compare XML documents which conforms to different DTDs. To cope with this problem we propose that there should be made DTD published by interest groups for commonly used data. This has already been done for Mathematical formulas (MATHML), Chemical data (CML), vector graphics (SVG) as well as for many other purposes.

7.3 Displets

Displets as described in [VCB97, CRV98], is a method to extend the element types of HTML and define how the new element types should be displayed. It

[4]DSSSL is also a transformation language.

can also be used to display XML documents instead of stylesheet language.

A displet is a display module programmed to display one element type. After an HTML/XML document is parsed, one displet for each element is instantiated. The displets instantiated know how to display the element type of their element. They form a tree structure equivalent to the DOM tree, and a displet can affect how its children are displayed.

The displet system is implemented as a Java applet and can therefore be used in most graphical browsers. Each displet is in effect a Java class, so to extend the element type set one needs to write a new Java class for each new element type.

Compared to a stylesheet language, which would be the alternative for displaying XML documents, the displet solution can display more complex data. This is especially important for non-textual data.

However, there are quite a few disadvantages:

- It requires support for Java applets which is not available in all user-agents
- The Java applet takes quite a long time to execute since it has to start the Java Virtual Machine
- In its current form the document is included as a CDATA attribute to an applet. This makes it very difficult to process the data in any other way. A reference to an invalid HTML document or a well-formed XML document would have been preferred since this allowed us to process these documents in other ways
- To introduce new element types some Java programming skills must be acquired. This will limit the number of authors.

Displets are not intended to replace HTML, but to provide a method to display non-textual data in a way that is supported by most browsers.

7.4 XHTML

As of October 2001 four different XHTML specifications exist, XHTML 1.0[W3C00b], Modularization of XHTML[Alt01], XHTML 1.1[W3C00c] and XHTML Basic[W3C00a].

XHTML 1.0 is a reformulation of HTML 4.01 as an application of XML version 1.0. This means that XHTML has the same relationship to XML as HTML has to SGML. This is shown in Figure 7.1.

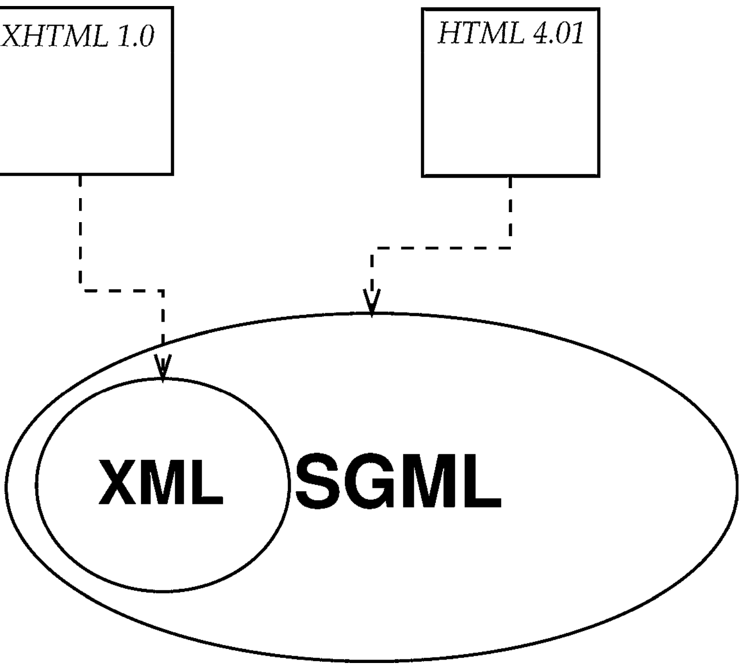


Figure 7.1: XML is a restricted version SGML. XHTML is an application of XML and HTML is an application of SGML.

XHTML 1.0 contains the same element types as HTML with the same semantics and some new element types for specifying the ruby annotation[5]. The difference between XHTML and HTML is that the notation available has been simpler and more restrictive through XML. This has some effects on how the language can be used:

- Much simpler to develop user-agents that display or process XHTML documents in some way. This is because there are less special cases that must be handled
- New authors will understand the parent-child relationships of a document more quickly since all tags are present in the document. This allow the author to understand the markup language better, and thereby use in a more sensible way.

All element types names must be typed in lowercase in XHTML. Unfortunately, all examples in the HTML 4 specifications are written in uppercase. This might turn out to be an annoying feature of XHTML.

[5]The ruby annotation is used in East Asian documents to indicate pronunciation or to provide a short annotation.

The basis for XHTML 1.1 and XHTML Basic is the specification of XHTML modules. It specifies the separation of all element types ever used in HTML into separate modules according to their semantics. These modules are for example:

Structure module body, head, title

Hypertext a

List module dl, dt, dd, ol, ul, li.

These modules can be combined to create different subsets of XHTML.

XHTML 1.1 is a reformulation of XHTML 1.0 strict using XHTML modules. The major difference between XHTML 1.1 and XHTML 1.0 is the removal of the deprecated element types and attributes present in XHTML 1.0 (and in HTML 4.01 using the transitional DTD).

XHTML Basic is designed as a basis for user-agents on platforms which do not support the full set of XHTML features. The specification provides a set of core XHTML modules which must be supported by all user-agents. This set can be extended by a user-agent to include other XHTML modules or new modules.

We will now describe some of the shortcomings of HTML which are solved to some extent by using XHTML:

- XHTML allows extension of the element types available
- XHTML has a stricter notation and all documents should be well-formed. Invalid content models can still occur and will probably be a problem. The trend among XML parsers is to do no error-correcting and this hopefully stops the traditional ad-hoc method of most browsers
- XHTML 1.1 removes the deprecated presentational element types and therefore distinguishes between structure and presentation to a greater degree than HTML
- There are no new element types which provide meta data. New element types can be introduced, but meta-data element types will not be of much use until they are standardized, preferably as an XHTML module
- XHTML Basic provides a subset of modules suitable for small devices.

Some new problems have arisen. The semantics of new element types must be defined. If interest groups publish new DTDs, they should define the semantics of the element types at the same time and thereby solving the problem.

Another problem is that stylesheet languages which are used to display unknown element types, generally do not have enough expressive power to properly describe non-textual data. There seems to be no easy solution to this. Either we must cope with the lack of expressiveness of stylesheets, or user-agents must build in support for the various markup languages defined through XML.

The future will in any case still contain markup languages.

8. Conclusion

This thesis has described in detail how error-fixing in major browsers is done. We have also proposed a set of methods which combined should handle all the different error-fixing methods in use in major browsers, except those that treat valid documents in an invalid way. The methods used for error-correcting include using two DTDs, fragmenting overlapping elements, implictly closing open elements, defining similar element types and defining a transformation system.

The structure of the parser proposed is a well defined module, which could easily be extended to handle documents from other SGML applications. The parser does not change how valid documents are parsed, it is very easy to change the rules of error-correction and it can easily report a precise error report on what errors were present in an HTML document.

We have also performed the first[1] major test on the validity of HTML documents on the Internet. The results collected here show the necessity of an error-correcting parser.

The regression testing software allows two IceStorm Browsers to be tested against each other. The system is based on several different module interfaces. Each module could therefore be internally changed, without directly affecting the other modules. The system could be extended to include more tests between the two IceStorm Browsers by making a Java class for each new test, and thereafter adding a few lines of XML code into an XML document.

[1]The author has tried to find other statistics in vain.

A. Java code for a parser

This parser is based on the methods found in chapter 2 : “Valid HTML”. It will only handle valid HTML.

```
import org.w3c.dom ;
import java.io.Inputstream ;

public class ValidHTMLParser implements SGMLlexCallback {
   SGMLlex sgml_lex
   Document document =null ;
   Node current = null ;

   Element indirect = null; //Used within the code
   /**
    * The constructor of the HTML parser takes as a stream of data
    * This stream is passed onto the lexical analyser which
    * immediately start to tokenize the data.
    * @param Inputstream data. A stream which represents the HTML document
    */
   public ValidHTMLParser (Inputstream data) {
      sgml_lex = new SGMLlex (data) ;
      sgml_lex.setCallback(this) ;

      sgml_lex.init() ; //start the lexical analyser
   }

   /**
    * The primary callback function. Takes as a parameter a PrimaryToken
    * which represents a start-tag with attributes, an end-tag or data.
    * This method is always called from the lexical analyser
    * It returns true if the token has been handeled correctly and false
    * otherwise. This is only done to increase readability.
    * @param PrimaryToken token, The token to be processed
    * @return boolean If the token was processed in a correct manner
    */
   public boolean primaryCallback (PrimaryToken token) {
      if ( document == null ) {
         //report error
         return false ;
      }
      Node currentBefore = current ;
      //Start tag handeling
```

```
if ( token.isStartTag() ) {
   //This call includes the attributes as well
   Element element = createElementNode((StartTagToken)token) ;

   do {
      if ( current.acceptsAsChild(element) ) {
         current.appendChild(element) ;
         current = element ;
         return true;
      } else if ((indirect=
          getElementWithOmitableStartTagWithChild(element))!= null
               && hasValidContentModel(current) )) {
         /*We come here because we can add an element
           with omitable start tag */
         current.appendChild(indirect) ;
         current = indirect ;
         //do one more round
      } else if ( canOmitEndTag(current)
               && current!= document) {
         current = current.parent ;
         //Move upwards in the DOM tree, do more rounds
      } else {
         //Backtrack and Report error
         current = currentBefore ;
         return false;
      }
   } while (true) ;/*We do this untill the token is processed
                  correctly or we have an error */

   //End tag handeling
} else if (token.isEndTag()) {
   do {
      if ( current.tagName().equalsIgnoreCase(token.getTagName()) ){
         current = current.parent ;//trivial close
         return true ;
      } else if (canOmitEndTag(current)
         && hasValidContentModel(current) ) {
         current=current.parent ;
         //Move upwards in the DOM tree, do more rounds
      } else {
         //Backtrack and Report error
         current = currentBefore ;
         return false ;
      }

   } while ( true ) ;/*We do this untill the token is processed
                  correctly or we have an error */

} else if (token.isData() ) {
   Text textNode = createTextNode(token) ;
   do {
      if ( current.acceptsText() ) {
         current.appendChild(textNode) ;
              /*text nodes can not have children,
           do not change current node*/
         return true ;
      } else if (canOmitEndTag(current)
                && hasValidContentModel(current ) ) {
            current=current . parent  ;
            //Move upwards in the DOM tree , do more rounds
         } else {
            //Backtrack and Report error
            current = currentBefore  ;
            return false  ;
         }

      } while ( true ) ;/*We do this untill the token is processed
                  correctly or we have an error */

   }


}

public boolean secondaryCallback (SecondaryToken token) {
   if ( token . isDoctype () ) {
      DocumentType documentType =
         createDocumentType ((DocumentToken)token ) ;
      //this is not a method in the dom interface
      document = DOM. createDocument (documentType ) ;
      current = document  ;
   }
   else if ( token . isMarkedSection () ) {
      //Find the type of marked section
      String sectionType =
         expand(((MarkedSectionToken)token ). keyword ()) ;
      if (sectionType . equals ("IGNORE")){
         return true ; //ignore
      } else if (sectionType . equals ("INCLUDE")) {
         sgml_lex . addMarkup ((MarkedSectionToken)token . markedSection ()) ;
         return true ;
      } else if (sectionType . equals ("CDATA")) {
         sgml_lex .addCDATA((MarkedSectionToken)token . markedSection ()) ;
         return true ;
      } else if (sectionType . equals ("RCDATA")) {
         sgml_lex .addRCDATA((MarkedSectionToken)token . markedSection ()) ;
         return true ;
      } else {
         //report error
         return false ;
      }
   }

}

public boolean errorCallback (ErrorToken token ) {
    //report error
    return true;
}


}
```

Bibliography

[Alt01] Murray Altheim. Modularization of xhtml, 2001. Available at http://www.w3.org/TR/xhtml-modularization/.

[Ben95] E.M. Bennatan. *On Time, Within Budget - Software Project Management Practices and Techniques.* McGraw-Hill International, 1995.

[Bir98] David J. Birnbaum. The problem of anomalous data. 1998. Presented at the Markup Technologies '98 conference in Chicago.

[BK93] A. Bruggemann-Klein. Formal models in document processing, 1993.

[BLLJ98] B. Bos, H. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, level 2 CSS2 Specification. W3C Recommendation. http://www.w3.org/TR/REC-CSS2, May 12, 1998.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[BPSM98] T. Bray, J. Paoli, and C. Sperberg-McQuee. Extensible markup language (xml) 1.0, 1998.

[Cla97] James Clark. Comparison of sgml and xml, 1997. available at http://www.w3.org/TR/NOTE-sgml-xml-971215.

[Con97] Dan Connolly. A lexical analyzer for html and basic sgml, 1997. Available at http://www.w3.org/MarkUp/SGML/sgml-lex/sgml-lex.html.

[CRV98] P. Ciancarini, A. Rizzi, and F. Vitali. An extensible rendering engine for xml and html, 1998.

[DEW97] Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. A scalable comparison-shopping agent for the world-wide web. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 39–48, Marina del Rey, CA, USA, 1997. ACM Press.

[fG96] I. for and S. Geneva. Document style semantics and specification language, 1996.

[Fla01] Alan Flavell. Use of alt in images, 2001. Available at `http://ppewww.ph.gla.ac.uk/~flavell/alt/alt-text.html`.

[Gol90] Charles F. Goldfarb. *The SGML handbook.* Oxford University Press, 1990.

[Goo98] Michel Goossens. The web prepares for the future, 1998. Available at `http://consult.cern.ch/cnls/230/cnl230.html`.

[Hic] Ian Hickson. Alt text minifaq. Available at `http://www.hixie.ch/advocacy/alttext`.

[How] Denis Howe. Free on-line dictionary of computing. Available at `http://www.foldoc.org`.

[HSM00] Claus Huitfeldt and C.M. Sperberg-McQueen. Goddag: A data structure for overlapping hierarchies, 2000. Seminar at the Universisty of Bergen.

[ISO86] ISO. Information processing systems - text and office systems - standard generalized markup language (sgml), iso is 8879, 1986.

[KC94] Patrick M. Kelly and T. Michael Cannon. CANDID: Comparison algorithm for navigating digital image databases. In *Statistical and Scientific Database Management*, pages 252–258, 1994.

[MD97] Jon Meyer and Troy Downing. *JAVA Virtual Machine.* O'Reilly & Associates, 1997.

[Org99] The Mozilla Organization. Mozilla web browser project, 1999.

[PA00] Roger Price and David Abrahamson. User's guide to iso/iec 15445:2000 hypertext markup language (html), 2000. Available at `http://www.purl.org/NET/ISO+IEC.15445/Users-Guide.html` A non normative guide to the ISO/IEC International Standard 15445:2000 for HTML.

[SMB94] C. SPERBERG-MCQUEEN and L. BURNARD. Tei guidelines for electronic text encoding and interchange, 1994.

[THCR97] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. *Introduction to algorithms.* MIT press, 1997.

[The91] The Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding. Version 1.0. Volumes 1 and 2.* Addison-Wesley, Reading, MA, USA, 1991.

[VCB97] Fabio Vitali, Chao-Min Chiu, and Michael Bieber. Extending HTML in a principled way with displets. *WWW6 / Computer Networks*, 29(8-13):1115–1128, 1997.

[W3C] W3C. Document object model(dom). Available at `http://www.w3.org/DOM/`.

[W3C99] W3C. Html 4.01 specification, 1999. Available at `http://www.w3.org/TR/html40/`.

[W3C00a] W3C. Xhtml basic, 2000.

[W3C00b] W3C. Xhtml[tm] 1.0: The extensible hypertext markup language, 2000. Available at `http://www.w3.org/TR/xhtml1/`.

[W3C00c] W3C. Xhtml[tm] 1.1: Module-based xhtml, 2000. Available at `http://www.w3.org/TR/xhtml11/`.

[WA99] Barry Wilkinson and Michael Allen. *Parallel programming: Techniquesand applications using networked workstations and parallel computers.* Prentice-Hall, 1999.