# HTML5 Parser

Dissertation submitted to the University of Manchester for the degree of Master of Science in the Faculty of Computer Science.

**2013**

**Mohammad Al Houssami**

**School of Computer Science**

## The Author

Mohammad Al Houssami has a Bachelors Degree in Computer Science from the American University of Beirut in June 2012 and currently pursuing a degree in Advanced Computer Science with concentration on Software Engineering.

Apart from the current work done on both project and research no previous work was done. The bachelor's degree acquired did not include any projects or research to be done.

Knowledge in the field of HTML in general and HTML5 specifically was very basic before the project. Parsing markup was introduced during the semi-structured data and the Web.

## Abstract

As HTML5's key feature is the detailed algorithm designed to parse any arbitrary content, the project is about producing an independent implementation for the HTML5 parsing algorithm. The project is built for the Smalltalk community which does not have its own native parser. The implementation also has a testing suite that can be used to check if things are broken or to continue the development on the project. As a final step, the project offers feedback to the standardization groups.

The paper includes a brief history of HTML and parsing. A part of the paper is dedicated for the language before going into the details of the implementation and the challenges faced. A simple guide to continue the implementation is also provided in the paper for individuals aiming to complete the parser.

## Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning

## Intellectual Property Statement

## Table of Contents

## List of Figures



## List of Tables

## Abbreviations

1. SGML: Standard Generalized Markup Language
2. ISO: International Organization for Standardization
3. HTML: Hyper Text Markup Language
4. CERN: European Organization for Nuclear Research
5. DTD: Document Type Definition
6. XHTML: Extensible Hypertext Markup Language
7. W3C: World Wide Web Consortium
8. XML: Extensible Markup Language
9. DOCTYPE: Document Type Declaration
10. XMLNS: XML Name Space
11. WHATWG: Web Hypertext Application Technology Working Group
12. SVG: Scalable Vector Graphics
13. MATHML: Mathematical Markup Language
14. RELAX NG: Regular Language for XML Next Generation
15. IDE: Integrated Development Environment

## Introduction

The HTML5 specifications are the latest contribution of the WHATWG community to the world of the Web in an attempt to create a markup that would mitigate the problems that the Web has been facing for the past years. HTML5 promises to be able to solve the mess of the Web with error handling being covered for the first time in specifications for a Web markup. Specifications by themselves are of course not enough to solve these problems so an implementation is required. Many other things need to be used and built as well, specifically a parser to parse the HTML as a first step. In order to achieve that, a transliteration process has to be done to change the specifications' pseudo code into actual programming code. The project being covered in this paper is about building a parser for HTML5 according to the specifications of WHATWG. The main goal of the project is to build a parser for a community that does not have its own. The project will also aim to find any ambiguities in the specifications as well as giving feedback about them. Learning a new programming language is also another goal of the project.

## Project Goals and Motivation

HTML5, being the latest version of HTML, has caught the attention of all the major Web browser companies in the world. All of these browsers are already using HTML5 even though it is still not a standard but an ongoing project which is expected to become a standard in 2014(1). With all new features and error handling introduced, it is obvious that a bright future is awaiting it. HTML5 introduces a whole new world of features never used in earlier versions of HTML. What is even more interesting is that HTML5, and for the first time; has the specifications for error handling in an attempt to solve all the chaos the Web is in. It is important to note that the specifications for HTML5, unlike the earlier versions; do not follow a grammar and thus parsing would be a challenging job to do. Building a parser based on a grammar is not so difficult to do. The grammar can be fed to a program that generates the parser. In the case of the new specifications, a very complicated state machine is being described and the state machine has to be changed into code. Moreover the specifications are written primarily for developers to

use which makes building the parser even more challenging for someone with little experience in the field of Web and programming.

Now for the technical part, the Web content is obviously a mess. Most of HTML pages suffer from errors and, according to a study about the DTD based validation of Web pages on the Web, it turned out that almost 95% of these pages are not valid HTML pages(2). Moreover, a test by Google was made on more than a billion documents by passing them through the HTML5 parsing algorithm. 93% of these documents had low level syntax errors (3). These facts obviously reflect how the Web is structured of broken pages and thus HTML5 rises in an attempt to find a solution to these problems and fix them. Building an HTML5 parser based on common specifications that are adhered to, would definitely be of importance for Web authors particularly and browser developers in general.

There are three objectives to this project. The first and the most important is building an HTML5 parser according to specifications. By doing that, we would be helping in validating the specifications and being of help to HTML5 project. There are some bugs and issues in the WHATWG specifications as they are not completely perfect. Therefore, one basic deliverable at the end of the project is a report to the WHATWG community discussing and resolving any bugs or ambiguities encountered in the specifications throughout the process. In order to optimize the benefit of the parser implementation, it was decided that the parser be built in an environment that does not already have its own implementation of the parser, in an attempt to supply a tool that can open opportunities for building more applications. The third and last objective is learning a new language. Not only will the project be implemented for a community that does not have its own, but it would also be executed in an unfamiliar language. So in conclusion the objectives of the project are:

1- Validating the specifications and providing feedback

2- Providing an under-served community with a parser implementation

3- Learning a new language.

The scope to be covered was initially set to cover HTML syntax that does not include scripting. However, because of the lack of time, only a subset of HTML syntax has been covered.

## A Parser Outside a Browser

Parsers are mainly built to be used inside browsers. Browsers use parsers to generate a DOM tree and use the tree in order to render the document to the user inside the browser. So why would a parser be useful if not used inside a browser? Well a parser can be used in various applications and for various reasons other than in browsers. The simplest of all is for pretty printing. Pretty printing is the process in which an application reformats source code into a stylistic way through changing font size or color, changing the location and position of text, changing spacing or any other kind of modification that's sole purpose is to make the text easier for people to perceive and understand.

```
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <h1 style="font-family:verdana;">A heading</h1>
5          <p>A paragraph.</p>
6      </body>
7  </html>
```

**Figure 1 HTML in Notepad++**

In figure1 we can see HTML code as it appears in Notepad++. The code is styled in a way to make it more readable. We can see that the indentation shows the hierarchy of elements. Colors also make text easier to recognize and spot with elements in blue, attributes in red and attribute values in purple while text is in bold black. The doctype elements appear in black. The process of reformatting the HTML code is pretty printing.

Another purpose that requires a parser outside a browser is minification or what is also known as minimization. Minification is the procedure in which unnecessary characters that are not useful to the code and don't affect its functionality are removed. The main purpose behind this is to save space and thus bandwidth in the case of the Web. This

process is mostly common in the Web and especially with scripting languages like JavaScript. Extra white space and new lines as well as comments do not affect the functionality of scripts and can be removed. An example in HTML is consecutive white spaces in text or new line characters between consecutive HTML elements. Most of what is removed in minification is only used for reading purposes and to make the code of better quality with no effect on the functionality (4).

```
1 <!DOCTYPE html><html><body><h1 style="font-family:verdana;">A heading</h1><p>
  A paragraph.</p></body></html>
```

**Figure 2 Minified HTML in Notepad++**

In figure 2 we can see the same code of figure 1 but minified. All the spaces, tabs and new lines characters have been removed. We can see that from the number of lines on the left which is now 1 rather than 7 like in figure 1, because all new line characters have been removed. Readability is definitely affected by this process but the code is still the same and the functionality is not affected. Though this is true, pretty printing can still help in identifying elements.

```
1 <!DOCTYPE html><html><body><h1 style="font-family:verdana;">A heading</h1><p>A
  paragraph.</p></body></html>
```

**Figure 3 Minified HTML in Notepad++ as Plain Text**

Figure 3 shows the same HTML code as in figure 2 but with no formatting or coloring of any kind. The code appears as plain text with not formatting. It is much easier to read and spot elements in figure 2 than in figure 3.

Another important application that requires a parser is HTML sanitization. HTML Sanitization is the process in which HTML documents are scanned for any unsafe tags that might be problematic. Tags like script, embed and link are usually among the tags removed in sanitization process. This will protect and prevent any attacks like cross site scripting attacks. Cross site scripting attacks are attacks that aim at Web applications. Scripts are added to pages being viewed by the client that are not originally there. This is

the most common Web vulnerability used in Web attacks (5). The effect depends on the type of the script that has been added. Sanitizers do not always use parsers but use regular expressions. This turned out to fail because of the many scenarios that can be applied to break the sanitizer from doing its job as intended. Changing the encoding to one not supported by the sanitizer but supported by the browser could be a disaster. The sanitizer will fail to detect any problems and the attack will be successful.(6) This is one of the easy ways to break a regular expression based sanitizer. There are more cases related to CSS parsing, HTML quirks, tag detection and JavaScript obscuration that are more complex but also break the sanitizer. A solution is proposed by building a parser based sanitizer that would be able to detect vulnerabilities. The solution is basically running all HTTP responses through a parser first.

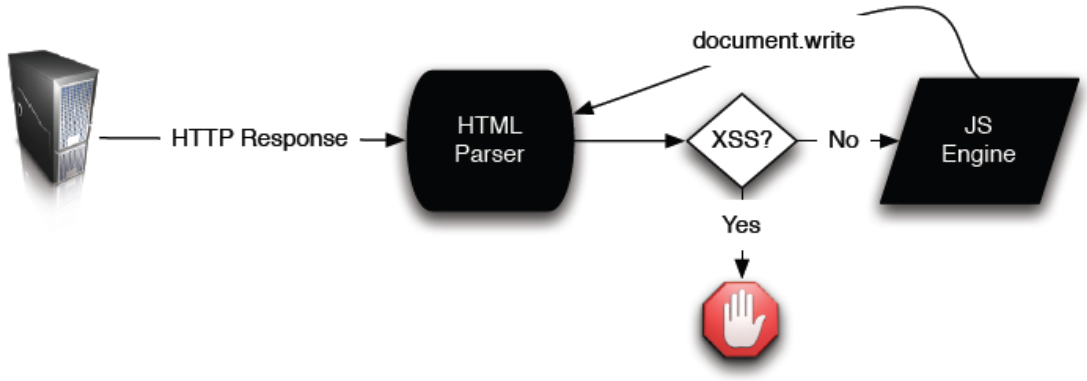



Figure 4 XSSAuditor Architecture (7)

XSSAuditor is a client-side XSS filter that does not use regular expressions but a HTML parser instead (7). In figure 4 we can see how the filter works. The parser looks for any XSS attacks. If any is found the response is blocked. If not, the response is then sent to the java script engine and back to the parser to parse again and check for any XSS attacks generated by any scripts(7). This is an example of a filter that uses a parser. The parser can also be used to produce a tree of elements that are then checked against either a black list or white list of elements where any unwanted elements are directly

removed. So sanitization requires a parser to be useful and trustable especially that sanitizers based on regular expressions are found not to work.

Moreover, Smalltalk has quite a good number of applications that use parsers in general including HTML parsers that are not used for browsers. Text lint is a project made using Smalltalk that uses a parser. The application checks for common English mistakes in scientific writing. The application accepts plain text, HTML as well as LaTex (8). A parser is used in the back end to parse the text and process it to give some output. Moose is another application that has a lot of functionalities. One of its major functions is reading code from different languages and doing some computation to give an analysis at the end. The process contains data mining and querying which require using a parser (9). The communities that use these applications are mainly Web application developers. Due to this fact, a parser for HTML5 built according to the specifications would be useful for these communities especially that the applications mentioned earlier still do not provide a parser for HTML5. Web developers can make lots of benefit from a HTML5 parser to help them do a range of things from building pages and manipulating them to developing applications that are based on a HTML5 parser.

## Environment and Language

As mentioned earlier, the parser was to be built in a language that does not have its own. In order to choose a language, a survey was made to check what languages do not have a parser implementation and from those choose the one that is in need for a parser the most. The process, which will be discussed later in this paper, resulted in choosing the Smalltalk language. Smalltalk is an object oriented programming language that is dynamically typed (10). Smalltalk appeared in 1972 and the stable release was released in 1980 (11). Smalltalk has several environments to work in. The most famous are Pharo and Squeak. For this project Pharo was chosen since most Smalltalk developers believe it to be more stable than Squeak albeit the fact that they share the same language. Moreover, parsers are used in several applications developed in Smalltalk like Seaside TextLint and Moose (12)(9)(8). Though this is true those applications do not use tools based on an HTML5 parser. Smalltalk does not have and

HTML5 based sanitizer or pretty printing services that might be useful for many of the projects built with Seaside like blogs for example. A parser in this language would be able to fill these gaps and allow these applications to be developed.

### Software Availability

One of the objectives of the project is to supply a community in need of a parser with one and thus, the parser is made available to all Smalltalk users that might be in need to use it. The HTML5 parser package can be downloaded from the gemstone repository on the following link: http://ss3.gemstone.com/ss/HTML5ParserMohammadHoussami.html The project is put under the MIT License and gives all users read and write permissions.

## Background Overview

### HTML: The Beginning

Text markup has been of interest for quite some time now and HTML has been noticed to be the most ubiquitous of all markup languages that showed up. Document markup started for printing purposes at first. As the industry of computers was rising, customers seeking to print documents started using computers to transfer electronic data rather than using hard copies. TeX is one of the oldest markup languages which was created in 1980(13). Another markup language created by IBM is called SCRIPT made as part of a product by IBM called IBM Bookmanager.(14) Another markup language also created by IBM is called GML which started in 1960(15). GML was then extended to SGML. In 1986 SGML became an ISO standard i. e. almost 4 years before HTML was derived. SGML is a structural paradigm based on either an inverted tree structure with a root and several branches and leaves at the end or a sequence of nested containers. The basic structure of SGML is based on elements. Elements have attributes to describe them. Elements are differentiated from text data using the less than sign “<” and the greater than sign “>”. These delimiters contain the name of element types and their positions in the document indicate the beginning and the end of the element.

HTML, the abbreviation of Hyper Text Markup Language, is a worldwide standard system for tagging text files to obtain fonts, colors, graphics and hyperlinks on the World Wide Web. HTML started in 1989 by Tim Berners-Lee while working at the computing services section at CERN, the European Laboratory for Particle Physics in Geneva, Switzerland. Berners-Lee invented the Web along with HTML as a language to be used for publishing on the Web. Since SGML was already an ISO standard, Berners-Lee based his HTML specifications on it. SGML already contained methods to markup text into structural entities like paragraphs, headings, lists, items and others. Berners-Lee was taking pairs of tags from SGML and adding them to HTML. The most important contribution that Berners-Lee did, however, was the anchor element which was not in SGML previously but rather purely added by him. The anchor element is used for hypertext links on the Web with the HREF attribute that takes a page address as a value. Out of this came the famous *www.name.name* format for addressing computers on the Web. (16).

## *The Web Chaos and the DTD Era*

People started showing interest in the Web and many companies started building browsers. Groups of people from different backgrounds and nationalities were assembled to improve HTML and many computer researchers and academics showed interest in HTML and the Web. With no constraints or grammars for people or developers to abide by, things started going out of control. Browsers started adding their own elements to the HTML language and different browsers had different elements for HTML. Developers were also writing messy HTML code with no rules to guide them. Web developers were using other pages' source code to learn HTML. In the absence of any tools to help building HTML pages, text editors were being used for writing the code and things can easily go wrong this way. Copying other pages' code also helped wrong HTML practices to spread quickly. The "view source" phenomena was spreading and everyone was learning HTML through reading and copying HTML source code. Let's take this simple example.

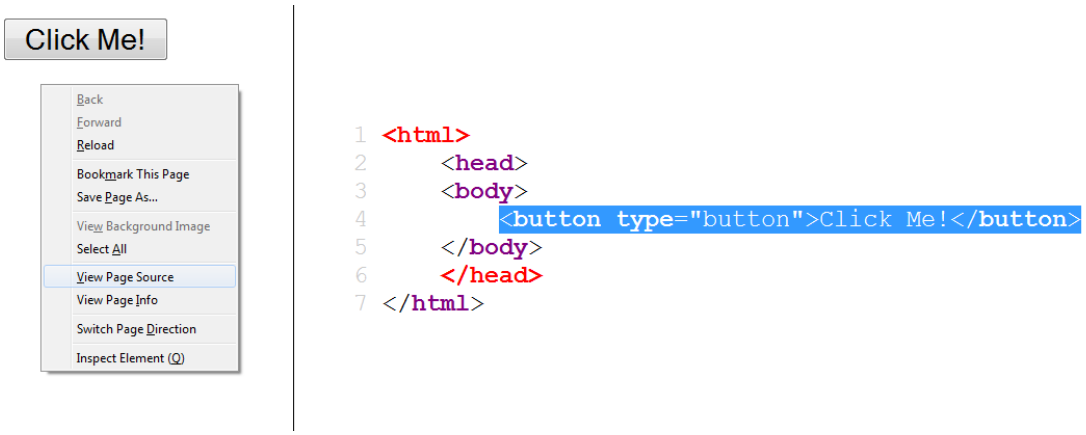



**Figure 5 View Source in Webpage**

On the left panel of figure 5, we see a button as rendered in a Firefox browser. A developer would want to know how to add a button and would right click on a page and hit view page source (or equivalent). The code on the right panel would appear and the code for the button would be visible. The developer would simply copy the part of the code highlighted and put it in his own Webpage in order to show a button.

The Web was soon turned into a mess because of the lack of conformity introduced, especially with respect to standard representations in a browser. In an attempt to get out of the chaos that was happening, a solution was proposed in July 1994. The solution was to put specifications for a new version of HTML called HTML 2. All widely used HTML tags in the Web were collected and put in a draft document that served as the base of Tim Berners Lee's HTML 2. In addition to the draft, Berners-Lee wrote a Document Type Definition known as DTD for HTML 2. Document Type Definition documents are a kind of a specific description of the rules of the language that define what elements can be used and how they can be arranged and nested throughout the markup declaration. DTD's are mathematically precise. By definition, a Document Type Definition "defines the legal building blocks of any [SGML based language] document. It defines the document structure with a list of legal elements and attributes" (17).

## Failure of DTD

The chaos the Web was in was due to the lack of structure and rules in HTML, the introduction of the DTD were thought to solve this problem. By providing rules and a proper grammar for developers to follow, the earlier stated reasons for why the Web was becoming chaotic should not be valid. However, DTD declaration was optional in HTML documents and even declaration of the DTD would not change anything in a document rendering wise. Browsers were not built with internal validators or the ability to use validation tools to validate HTML documents with respect to the DTD declared so no effect was noticed.. Moreover DTD did not have powerful expressivity. DTD did not have any kind of rich type system but rather a simple one that cannot describe types properly like integers for example. DTDs could not precisely describe the number of valid elements possible.

There still were a relatively large number of invalid HTML documents. Many new browsers were coming up and still new browser developers were adding new tags to HTML and so, the previous problems persisted. During 1995 all kinds of new HTML tags and elements as well as attributes were showing up

## HTML 3.2

In January 1997, HTML 3.2 was ready along with cross industry specifications that were reviewed by all the members of all the organizations that were working with HTML as well as the major browser developing companies such as Netspace and Microsoft. HTML now had a stable specification approved by almost everyone who is concerned with HTML, Web browsers and the Web in general. HTML now had a richer syntax and much more expressive elements (18).

## HTML 4.0

In December 1997, HTML 4.0 was released. HTML 4.0 had all the features of the previous HTML versions, more multimedia features, scripting languages and style sheets. HTML 4.0 also aimed to improve the Web and clean up the chaos it was in. The attempt was through three basic improvements. The first was by organizing the code.

HTML 4.0 aimed to keep formatting code away from styling code to keep things clear and clean. The second was through disapproving any elements that were arbitrary and seemed to cause problems. The last and the most important was making document type declaration obligatory rather than optional. However, this attempt failed since browsers still did not support any validation of documents against the declared document type and the problem persisted with invalid documents (19)(20).

### *XHTML 1.0*

On January 26 2000, a new version of HTML that was to solve these problems was awaiting the Web. XHTML 1.0 was released as a new worldwide standard. Because of all the mess the Web was suffering from, the W3C decided to do a reformulation of HTML to make it similar to XML by applying the concepts and rules of XML on Web markup, especially draconian error handling, and thus came the naming XHTML. The three principles of HTML 4.0 discussed earlier were also present in XHTML 1.0 but with more additions taken from XML to aid XHTML 1.0 in improving the basic syntax (20). XHTML 1.0 not only gave Web developers the capability to write code that is up to standards but also obliged them to change their perspectives.0 This was primarily because of the draconian error handling that was introduced to XHTML 1.0 which was not present in earlier versions of HTML. Draconian error handling was basically treating any error as being a fatal one. As not abiding by the rules of the HTML DTDs was still not affecting the rendering of pages, it seemed that the only way to oblige developers to abide by rules was by making their pages fail.

### XHTML 1.0 Improvements and Requirements

As every version of HTML, XHTML 1.0 had several improvements. XHTML 1.0 was putting strict requirements for documents to be valid. XHTML 1.0 documents should have a DOCTYPE that showed it was an XHTML 1.0 document as well as the document type declaration. As every HTML document, an XHTML 1.0 document had the HTML tag <html> as the root element. XHTML 1.0 required that this element included the xmlns namespace as well as a value for it. As for the structure and syntax of the document, they had to follow the rules of XHTML and not the usual HTML. Moreover, it was

advised that XHTML 1.0 document be declared as an XML document but it was not obligatory like the requirements mentioned earlier (20)(21).

## XHTML Syntax

As for syntax concerns of XHTML, a number of things was introduced. Once the XHTML 1.0 document had all the required declarations, the influence of XML on the Web markup became obvious. The XHTML 1.0 syntax had case awareness which was not in earlier versions of HTML. The changes also included well formedness of tags and elements, as well as empty and non-empty elements and the quotation marks use in documents(21)(22).

## Failure of XHTML on the Web

Looking at XHTML, it seemed that with this version the mess that the Web was in could have been solved if it had been adopted. XHTML adoption unfortunately did not happen even though it became a standard. It was used along with HTML and browsers supported both. This was mainly precipitated by several basic factors listed below.

- HTTP and character encoding conflicts and the default content types
- Draconian error handling
- Poor tool support and limited popularity

### *HTTP and Character Encoding Conflicts and the Default Content Types*

One of the strengths of XML was that developers could use any encoding they wanted depending on what was appropriate. This was true for local storage but not over the HTTP. HTTP was not looking into the files to figure out the character encoding but rather used its own method. The problem was that the HTTP method did not always give the same character encoding as that declared in the xml file and the result was two different character encodings. According to the specifications, the HTTP encoding was the one to be used (23). Moreover, encoding declaration was optional in both XML and HTTP. This would result in having no encoding at all, having encoding in the file but not in the HTTP as well as the other way around or having them both. This became common in feeds. Publishing systems did not generate feeds every time a request was received but rather

created the feed file only once and stored it in the cache. What happened was that the Web server that was serving the feed did not know the encoding of the application asking for the feed and therefore ignored it. The server would serve the feed with no encoding in the HTTP header. In order to fix this issue, a set of rules were put to determine the encoding set by the Network Working Group under RFC3023. The problem was that almost all Web servers had a default configuration for sending any kind of static xml files which included feeds. The content-type was set to text/xml without any charset parameter by default. This resulted in a huge number of ill-formed documents which led to the second reason why XML failed in the Web: The draconian error handling (24).

### *Draconian Error Handling*

Postel's Law states the following "be conservative in what you do, be liberal in what you accept from others." The first part of the law applied to XML because developers were sending well formed documents but they were perceived as ill formed by the receiver. Due to the fact mentioned earlier, many originally well formed documents have changed to be ill formed as a result of the encoding not being correct. Draconian error handling would basically ignore all these documents. In order to comply with Postel's Law, the XML developers had to make sure the documents they produced could be valid under text/xml. The solution was using us-ascii for character encoding. This would result in file sizes to increase at least four times and up to eight times more, since characters would require four to eight bytes to be represented instead of one byte. For feed files that were sent many times a day, this was a huge increase in file size. Looking at the second part of Postel's law, XML seemed to ignore it. With draconian error handling, XML would only accept valid documents and just ignore any document with any sort of error. XHTML was based on XML and thus also adopted the draconian error handling. Applying these strict rules to Web markup was not a good idea. The Web was based on versions of HTML where well formedness was not an issue and brining XHTML into the game meant having to refactor the entire Web. But why didn't new Websites adopt

XHTML then? Simply because it was harder to get right. Everything had to be perfect to render; something most developers are not used to.

```
<html>
    <head>
    <body>
        <b><P> Non Well Formed Page </P>
        Broken Here</b>
    </head>
</html>
```

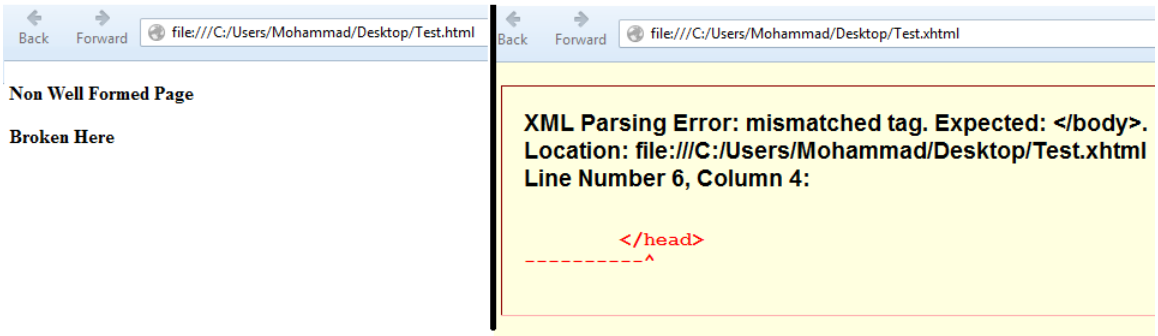



**Figure 6 Rendering HTML & XHTML**

In figure 6 we see ill formed HTML code rendered in a browser as HTML and XHTML. In the left panel, we see that when the file extension was .html in the address bar, the browser rendered the file correctly in the browser. On the right panel, the same file is being rendered but with a .xhtml extension. The browser fails to render and gives a parsing errors. The browser complains about expecting a closing body tag which is the cause of the ill formed code.

With all browsers supporting error handling, developers became inclined towards writing markup that displayed correctly on browsers rather than conforming to the rules. Unfortunately, browsers kept supporting HTML as well as XHTML rather than supporting only XHTML which was a standard and consequently, no one was obliged to shift to XHTML because an “easy going” alternative was available along with good tool support which brings us to the third reason why XHTML failed to assert itself (25)(24).

### *Poor Tool Support and Limited Popularity*

All the major software tools that were used by developers – such as Macromedia Dreamweaver, Adobe GoLive as well as Microsoft FrontPage – did not support XHTML. Developers were accustomed to these tools and were either not willing to switch to other new tools or had no choice but to stick to these tools because of the environment they worked in. Moreover the media did not play a good role in propagating XHTML and therefore, it did not gain a lot of popularity. Not a lot of people knew about XHTML; as for the ones who did, they were not able to notice how easy it would be to switch to it (24).

## *HTML5*

With no new major features and additions being added to XHTML, it was getting clearer that XHTML won't be solving the problems of the Web and a new solution was required. This time the solution would come by fixing the errors rather than trying to prevent them from happening. This is what HTML5 aims for; error handling.

### Mozilla/Opera Joint Position Paper

The Mozilla Foundation and Opera Software submitted a joint position paper in June 2004 during a workshop held by the W3C. The joint position paper had seven principles. The first was regarding backwards compatibility between different versions of HTML. After the rough transition from HTML4 to XHTML 1.0 this became an essential point. W3C was not providing any means of either compatibility with earlier versions or methods to migrate from earlier to newer versions.

Another thing the paper discussed was having specifications for error handling. This was never put in any of the earlier versions of HTML. The paper also discussed other things as well as features that are nice to have. Some of these features are related to SVG and JavaScript (26).

## Formation of WHATWG

On the second day of the workshop held by the W3C, the 51 attendees had to vote on the joint position paper. 8 people voted for the paper where as 14 voted against it (27). The WHATWG community was formed two days later and introduced to the public. The working group was accused of being unofficial and loose consisting of people with common interests. The WHATWG's main purpose was creating specifications for what was discussed in the joint position paper. The group consisted of people from Opera, Mozilla and Safari (28).The group was an invite only group and only people invited could be part of it. The mailing list was still available for the public to join and contribute. Note that Microsoft was invited to join the WHATWG community for being a leader with its Internet Explorer browser but it refused to join.

## WHATWG HTML5 Specifications

The first thing to notice is that HTML5 is not based on SGML or any context free grammar unlike all previous markup languages. With a grammar based markup, everything not recognized has to be rejected which is not the case anymore in HTML5. Adding to that, in it is possible to always find a derivation to a parse tree and thus give output no matter what the input is. HTML5 is designed more towards flexible parsing as well as compatibility especially for mobile devices (29). The principle design goal of XHTML is to reject erroneous documents so there will be fewer of them on the web. The main design goal of HTML5 is to make the results of parsing erroneous HTML documents unique across browsers causing applications to have a consistent behavior. HTML5 also allows the use of inline SVG and MathML inside text/html elements. A number of new elements were introduced as well. In addition to the new elements and new form control types, new attributes were added as well as global attributes that can be added to any valid element. As new elements were added, others were deprecated. HTML5 also stopped using DTDs. A switch to RELAX NG and Schematron was made to validate schemas. (30)

So why would HTML5 change the Web? Patrick H. Lauke, one of the developers of the Opera browser, says in an article to the .Net magazine in February 2011 that "the

HTML5 specification goes into a lot more detail — not only defining the various markup elements available to authors, but also describing exactly how markup itself should be parsed" (31). This way, every browser developer now has specifications for building a parser to be used in browsers so that all HTML5 documents can be rendered in the same way on different browsers and different platforms. Moreover, HTML5 has backwards compatibility. The path to travel from HTML 4.01or earlier versions and XHTML was clear and smooth. The document type has to be changed to HTML5 and everything should be fine. Documents should still validate and render (32)(33).

The most important thing in the HTML5 specifications is that error handling is well defined. HTML5 aims to be able to handle errors in order to produce a parse tree for every input. As a result HTML5 also promises to be able to give a unique rendering of HTML by parsing HTML in the same way. This allows all browsers to tackle errors in the same way which results in more consistency in rendering pages on different devices, platforms and browsers. Because of this feature, many mobile phones are now compatible with HTML5. According to Strategy Analytics the number of compatible mobile phones in 2013 will reach 1 billion devices up from 336 million by the beginning of 2012(34). This clearly shows that HTML5 is growing and it is growing fast even though it is still under development and not yet standardized.

The fact that HTML5 is aiming at dealing with the “tag soup” that has been generated by all previous versions of HTML is what makes HTML5 important and makes parsing it much more difficult than any previous version. Jeremy Keith in his book *HTML5 ‘For Web Designers’* says the following: “HTML5 is written for authors and implementers. It explains why the HTML5 specification is so big and why it seems to have been written with a level of detail…”(35).

## HTML5’s Bright Future

HTML5 is still not a standard and is a project in progress. A plan has been suggested by W3C, and by the end of 2014, it is expected to have a stable HTML5 recommendation. The plan also includes dates for releases of HTML5.1 and 5.2. A first working draft of

HTML5.2 is expected to be released in 2015 whereas 2016 is set as a milestone to release a recommendation of HTML5.1. (36)

## Percentage of Websites Using HTML5

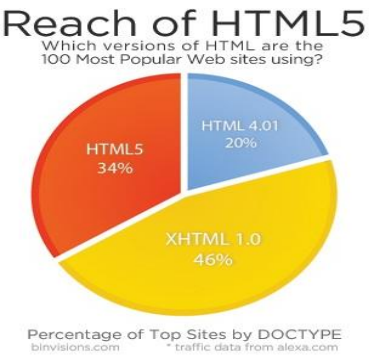



**Figure 7 Percentage of Websites Using Different Versions of HTML(62)**

Looking at how things are moving with HTML5, it is clear that it is going to be the next big thing. Lots of developers seem interested in HTML5. According to a survey by Bin Visions reflected in figure 7, 34% of the top 100 most trafficked sites in the world are already using HTML5. 46% are using XHTML 1.0 while the remaining 20% are using HTML 4.01. Out of these sites, social media sites had the highest percentage with 17% whereas technology mainstays or sites that offer downloads and support like Mozilla and Microsoft Websites, come second at 13%. These numbers were released on September 30th 2011 and the numbers are growing even more by the day. This clearly shows how fast HTML5 is spreading unlike any of the previous versions that did not spread so quickly especially XHTML which was not used a lot even after becoming an international standard. With HTML5 the Web awaits a promising future in the coming few years (37)(20).

# Parsing

## Parsing and Parsers

Parsing is a procedure whereby a sequence of characters of some kind of a language, either natural or computer, is analyzed with respect to the grammatical rules of the language being used. In computer science, parsing is the process in which the computer does the analysis on the string provided. It results in a parse tree returned by the computer being used. The parse tree holds syntactic information about the string parsed as well as other information such as the semantics. The program that does the parsing on the computer is called a parser. They are usually a partial structure of a bigger component. Interpreters and compilers rely on parsers. The parser in these components takes the input and creates a form of structure for the component to use.

The structure is usually a type of hierarchical structure that describes the relation between the characters being parsed; usually of a tree form(38).

## Parser Structure

A parser is made up of two parts. The first part is the lexical analyzer, also called a lexer. The second is the syntactical analyzer. A lexical analyzer is what performs the lexical analysis part of the parsing process. The lexical analyzer scans the input that the parser is given and produces a set of tokens that represent the input according to the lexical grammar of the programming language. Tokens are a kind of representation of the input according to some rules of tokenizing being followed that reflects the actual input. The syntactical analyzer has three major tasks. The first is analyzing the syntactical structure of the tokens being passed to it. The second task is checking the tokens for errors. Once these two bits are done, the analyzer goes to the third and most important step which is building the structure which as mentioned earlier is mostly a parse tree (39)(38).

In figure 8, we can see how the parser is represented by one structure with two components inside it. This parser takes some input string which is received by the lexical analyzer. The lexical analyzer processes the string, creates the tokens and passes them to the syntactic analyzer. The syntactic analyzer does the required processing and returns the parse tree. The role of the parser is done at this point. The parse tree can now be used by other components as input to carry out further processing and give certain output.

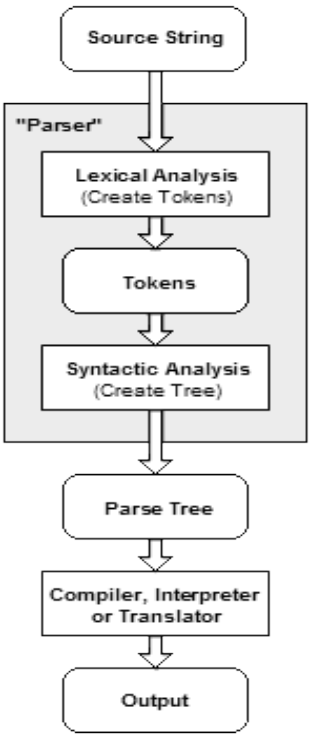


Figure 8 Parser Structure(63)

## Error Handling in Parsers

Error handling is one of the most important parts of the parsing procedure. Since language specifications usually do not include any specifications related to error handling, parser developers had to deal with the errors on their own. Different parsing mechanisms have different abilities in handling errors. Error handling can be categorized into three categories.

The first category is to just stop the parsing procedure. The second category is to try to go further in parsing and trying to find other parse errors. This way the programmer will be able to know more than one error after every parse. The third category is correcting errors. Error correction tries to correct the code being parsed as part of the parsing process. This, however, is very hard to achieve because of the fact that several input can be expected each with a different way to handle.

In HTML, error handling has been part of the browser developer's job until HTML5 arrived. Browsers have different ways of dealing with errors but the general aim is trying to render as much of the document as possible. In HTML5 error handling is part of the specifications and is no more part of the browser developer's job.

# HTML5 Parsing

## HTML5 Parsing Challenges

So why is HTML5 parsing a challenge? First of all, HTML5 promises to solve all the chaos of HTML; something that all previous versions of HTML tried to do but failed. Asking Ian Hickson, a member of the WHATWG and a major specifications writer for HTML5 about the difficulties he replied with the following "The main challenge was that none of the browsers agreed, and so we had to first figure out what was actually necessary for compatibility with deployed content. After that, it was mostly just a lot of work, and then a lot more work cleaning up the edges". The first thing was to try and convince all major browsers to go ahead into the idea of HTML5. This was discussed earlier in the joint position paper that did not get an agreement between the attendees of the W3C conformance.

Now let's look at the technical side and see the major challenges faced. The first and major issue is trying to deal with the tag soup phenomenon that has spread in the world of Web. How to deal with <a></a>? In all the earlier versions of HTML, no specifications for invalid markup were defined. XHTML took things to a better level by not allowing such invalid markup making this the most important feature of XHTML.

Still, this did not define a way to deal with invalid markup. Every Web browser had to find a way to deal with invalid input which resulted in having a unique input giving different DOM trees on different browsers. In HTML5 the same input should always result in the same DOM Tree. Let's take the following example and see how different browsers interpret it. (40)

```
<body>
  <p>This is a sample test document.</p>
  a <em> b <address> c </em> d </address> e
 </body>
```

**Windows Internet Explorer 6**

WIE6 would generate the below tree in figure 8 for the input above.

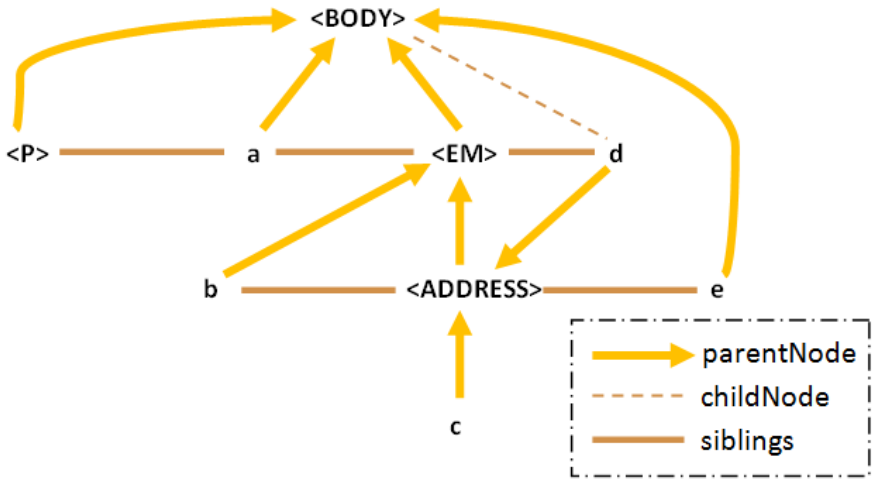



Figure 9 Dom Tree Generated by Internet Explorer 6 (40)

Notice that 'e', in figure 9, has been added as a sibling of the 'address' element and a child of the 'body' which is completely clear since it is outside both 'address' and 'em'. 'd' is set to have 'address' as a parent node the same as 'c'. What is done here is that all the elements before the opening tag of 'address' have 'em' as a parent. Anything after the opening tag of 'address' has 'address' as a parent. IE is somehow trying to follow the

order of the tags opened regardless of the order they were closed and assigning the parents this way.

**Mozilla 1.2**

Mozilla Firefox would generate the below tree in figure 9 for the input above

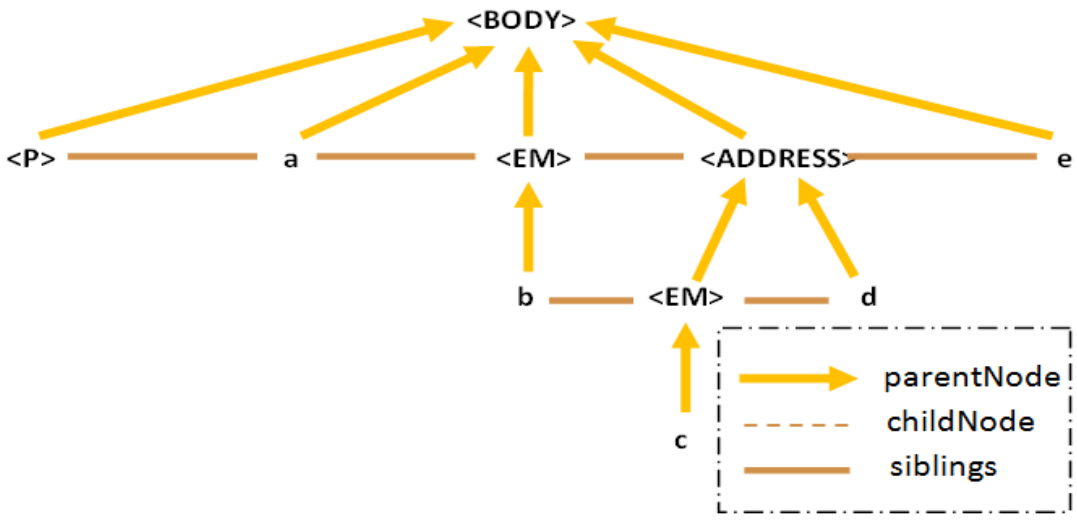



Figure 10 Dom Tree Generated by Mozilla Firefox (40)

The first thing to notice in the tree in figure 10 is that there are two 'em' elements present in the tree. Mozilla is closing the 'em' element when it reads the 'address'because an 'em' element cannot have an element 'address'. When reading the closing tag of 'em', Mozilla adds another 'em' element to the tree as a child of the 'address'node. This is different from the way IE was dealing with it and thus the different DOM tree at the end.

**Opera 7**

Opera 7 would generate the below tree in figure 10 for the input above

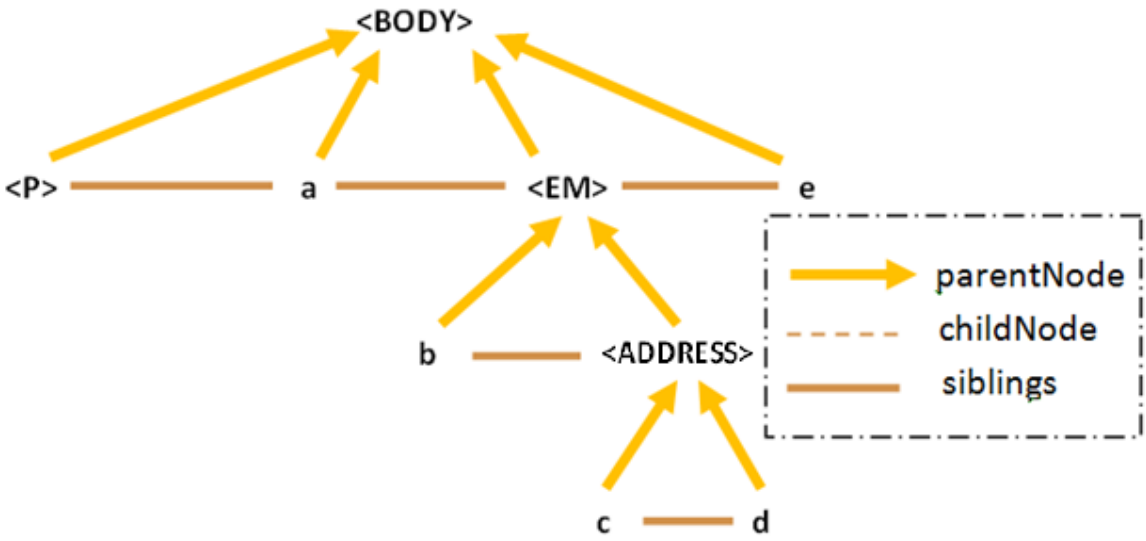



Figure 11 Dom Tree Generated by Opera (40)

What is being done here by Opera in figure 11 is leaving any unjustified closing tags to a time in parsing where the closing tag can be justified. In others words, any closing tags are not considered until all child elements have been closed. This solution does not take the DTD into consideration. The 'em' element is not closed until the 'address' element close tag has been processed. This explains why both 'c' and 'd' have 'address' as a parent.

With IE and Opera's approaches it is easy to deal with any invalidities in styling and scripting. Any changes done will affect all the elements which might not be the case in the tree generated by Mozilla because there are 2 EM elements. Moreover, the parse tree generated by Opera and Mozilla are much more stable. If we look at the tree generated by IE, we can notice that it would be possible to face an infinite loop trying to go over the elements of the tree because of the loop generated between 'd' 'address'

and ‘em’. After doing a complete loop and back to ‘em’. The sibling will take us back to ‘d’ and the loop continues. Mozilla’s tree on the other hand, makes it easy to interpret the result of the tree unlike in IE and Opera where things become more complicated.

In conclusion, we see how each approach has its pros and cons and how each one deals with the problems in a different way. Trying to find a unique solution for the tag soup is among the hardest challenges to be faced by HTML5 specifications writers.

## Error Handling in HTML5

Misnested tags are one of the most common mistakes in HTML. Take the following HTML code as an example :    test  . We can notice that the bold tag is closed before the italics tag. At the point when the parser reaches the italics open tags there will be 2 active formatting tags; the bold tag and italics tag. After reading the close tab tag the parser finds that there is a mistake the list of active formatting tags is updated to include only the italics tag. No change to the DOM tree is required at this point. After reading “test”, a reconstruction of the active tag element, in our case is italics, is required and an italics element is added to the DOM before adding the “test” text node to the tree.

Another example is   test1  test2  which is a little bit similar to the previous example. The bold tag is closed before the paragraph tag. When the closing tag of the bold element is parsed the open elements stack has four elements. Html, body, b and p. The active formatting elements stack has one element which is the bold tag. A new bold element will be created. All the children below the paragraph node will be moved to be below the new bold element. So the new bold element will have “test1” as a text node child. After reading the text “test2” after the closing b tag, the bold tag and all its children are moved to be children of the paragraph tag. The text node “test2” will be added as a child of the paragraph element becoming a sibling of the bold tag.

A third and final example is having nested tables. In the case of nested tables where we have a table tag inside another table, the parser will have to deal with this case as well. The inner table will be moved outside of the outer one. The result will be two separate

tables. This is in brief how the parser deals with this issue and how the browser renders it.

The examples given above are among the simplest error handling examples and still are complex to handle, so tackling complex errors is not an easy task requiring complicated error handling. This is exactly why the HTML5 specifications are fairly large compared to previous HTML versions. Implementing the error handling part of a parser is not an easy job to do but with the WHATWG specifications available, things became easier. Error handling now has a specification to be used in building parsers making the mission of building a parser easier.

## HTML5 Tokenizer Specifications

The tokenizer specifications are a precise description of how HTML5 tokenization takes place. The specifications provide an explanation of the tokenization process before providing the pseudo code that describes the tokenizing process. The pseudo code spans 1083 lines covering all the states of the tokenizer. Though the pseudo code is not very complex, it is at the same time not obvious to understand.

The tokenizer specifications of the parser are separated into 2 parts, an introductory part and the pseudo code part. The introductory part of the tokenizer explains briefly how the tokenizer works but focuses mainly on the way the tokenizer stage and the tree construction stage interact. The introduction also covers how to deal with parse errors and when parse errors can be thrown in the tree construction stage. Scripting is also mentioned and how executing scripts can result in dynamically adding markup to the document. The tokenizer resembles a state machine with states that cover the stages of the tokenization. The state machine has 68 states as well as 1 guide to tokenize character references. In each state the pseudo code explains how to deal with any input. There are usually the expected characters that result in doing something specific and the anything else branch that is general for any other kind of character that might be read by the tokenizer. Below is the pseudo code of the data state which is the start state of the state machine.

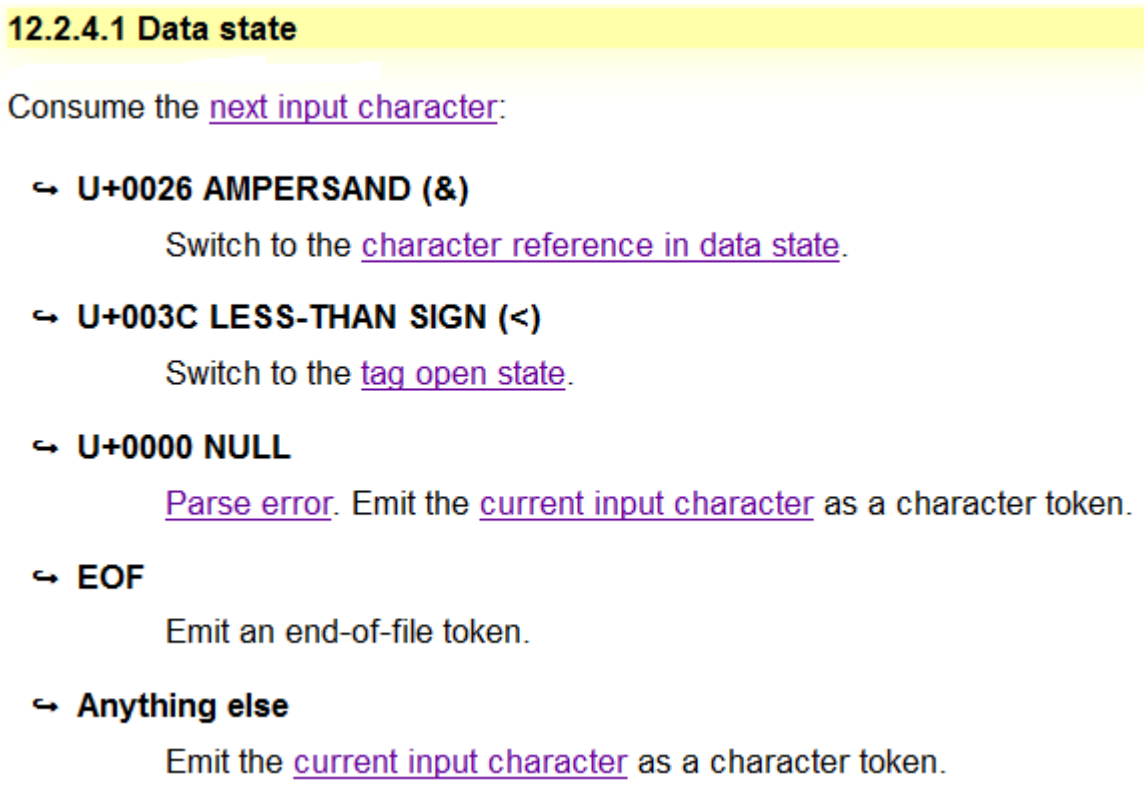



Figure 12 Tokenizer Data State(41)

The first thing to notice in figure 12 is that Unicode values are used to reference characters in order to prevent any ambiguities in the specifications. The initial step to do in all states is to read a character. In some states more characters have to be read in order to check for patterns like two consecutive dashes for comments or check for specific words like DOCTYPE, PUBLIC or [CDATA[. In the data state we have cases were we switch to another state like the case when reading an ampersand or less than sign. In the case of a null character a parse error is thrown. Parse errors have a specific way of dealing with them which are described in the specifications. We also emit tokens in the case of a null character or an end of file (EOF) character. In these two states parsing stops and no other states are called. In the anything else branch the character is emitted as a character token. In this case we do not see any switch to a state just like the case of the EOF and null characters. This means that we do not change state but go over the state again and consume another character.

In some cases in the tokenization state, a character can be ignored like spaces or tabs. The next character is then consumed in the same state. Other cases require reprocessing the character and changing state. The character reference in attribute value state is the only state that requires information about the state that lead to it. After doing some processing the character reference in attribute value state has to go back to the state that leads to it of which there are only 3 possible states.

## HTML5 Tree Construction Specifications

The tree construction specifications follow a similar format to that of the tokenizer. The tree construction is separated to five parts. The first part is about creating and inserting nodes. This is a combination between pseudo code and regular English explanation of how to create and insert nodes. Before inserting a node the proper position for inserting the node has to be chosen. The process depends on several variables.

The first one is the target which can either be specified implicitly, and in this case no further action is required, or the current node which is the bottommost node in the stack of open elements. The stack of open elements is a stack that grows downward and has all open elements that have been processed. The bottommost element resembles the top of the stack in the common stack structure. Next is the fact whether foster parenting is enabled or disabled. In specific cases where foster parenting is enabled and the target belongs to a specific set of elements more steps have to be taken that move elements around in the tree and repositions elements before returning the exact position to add a new element.

The second is creating the correct element to be added. The specifications define a specific element structure for every HTML element. The namespace of the element also plays a role in deciding what element is created. Attributes in the XMLNS namespace also require more things to be done before creating the element.

The last is inserting the element. In one case where the insertion location is the document itself and if the document cannot accept more child elements (if the document already has a child) then the element is just dropped. The element then has

to be added to the stack of open elements and the current node has to be assigned to the newly added element. This part also describes how to deal with some specific elements with MATHML attributes and SVG attributes where the name on the token are in lowercase where as in the element they have to be added as camel-case. A list of all the possible attributes is provided in the specifications.

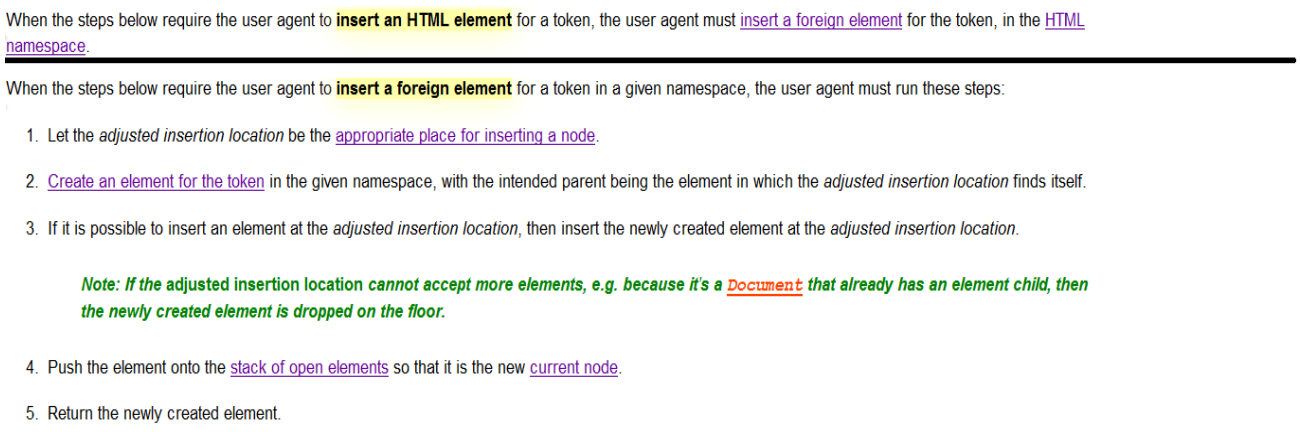
Inserting characters is also covered in this section. Inserting characters is similar to inserting other tokens. It requires updating the position as well as creating a character element. Similarly is inserting comments. The position has to be updated and a comment node is created and added to the correct position in the tree.

The second part of the introduction describes the interaction between the tree construction and the tokenizer. There are two algorithms, the generic raw text element parsing algorithm and the generic RCDATA element parsing algorithm that switch the tokenizer state to the RCDATA state. These algorithms can only be called after reading a start tag token.

The third part of the introduction explains how to close elements that have implied end tags. During parsing some end tags have to be generated and this happens by popping elements of the stack of open elements. There are conditions to what elements to be popped. In HTML, some elements can only be present as children of other elements. If the parent element is closed then all the child elements have to be closed since they cannot exist outside the scope of the parent.

The fourth part is about the rules of parsing the tokens into HTML content. The tree construction stage uses the same methodology in dealing with the tokens as the tokenizer in dealing with the characters. There are 23 different insertion modes that are equivalent to the states in the tokenization stage. The way of writing the pseudo code is the same as that of the tokenizer. An action is to be done depending on the type of token that is received. The actions are usually adding elements or throwing parse errors. Each insertion mode might switch to another insertion mode keep the same insertion mode change the tokenizer state or stop the parsing process.

The process is complicated. Unlike the tokenizer stage there is no text to explain how the tree construction works in general or what it represents. The pseudo code refers to so many algorithms that also might refer to other algorithm which makes things very complicated to understand. Let's take inserting an HTML element as an example



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

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

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

   ***Note: If the adjusted insertion location cannot accept more elements, e.g. because it's a `Document` that already has an element child, then the newly created element is dropped on the floor.***

4. Push the element onto the stack of open elements so that it is the new current node.
5. Return the newly created element.

**Figure 13 Insert an HTML Element**

Inserting an HTML element algorithm leads to inserting a foreign element in the HTML namespace. We can see the algorithm in the 2nd pane in figure 13. Inserting a foreign element requires adjusting the location and creating the element, each of which has its own complicated algorithm that also requires more algorithms or checking other variables.

The last part gives the rules for parsing tokens in foreign content. In this part scripting is covered. Different scripts have to be processed in different ways according to the namespace they fall in. Script elements in the SVG namespace need to be processed according to the SVG rules. Other than scripts, there are rules to parse some specific elements and rules to parse all other elements in general.

# Smalltalk

## Smalltalk Background

Smalltalk is a programming language that started development in 1969 and was first released in 1972. The first stable release of Smalltalk was the second version of it named Smalltalk-80 in 1980(10).

The Smalltalk language, like Java and C++, is an object oriented language. It is on the other hand dynamically typed and all type checking happens during run time rather than at compile time. It is also a reflective programming language that allows examining of the structures and values as well as properties and methods.

Smalltalk was started as a project for educational purposes and was designed in a way for constructionist learning, the method of learning by modeling things to be able to understand them. The project was started at the Learning Research Group(11). After the stable version in 1980, many variations of Smalltalk and Smalltalk like languages appeared. In 1998 ANSI Smalltalk was released as the new standard and stable version of Smalltalk(11).

## Smalltalk Influences

Smalltalk has had a lot of influence on the world of programming languages on many different fields. There are basically four major points of influence of Smalltalk on computer programming.

The first thing is through the syntax and semantics. The most famous are Ruby Python and Perl. Ruby and Python have taken some ideas from Smalltalk and implemented them. The object model of Perl is inspired from that of Smalltalk. Objective-C has both syntax and runtime behavior similar to that of Smalltalk. Other languages that are influenced by Smalltalk are S#, Script.NET, Self, and NewsPeak.

Second is that Smalltalk was used as a prototype for the message passing computation model. Message passing is the communication process which is most known to be used in object oriented programming as well as in parallel computation and communication

between processes. This allowed the ability to make processes or objects to synchronize as they have to wait to receive a message.

Third is the windows, icons, menus, pointer graphical user interface which was the inspiration to having windows in Operating System. The first desktop Macintosh had windows almost the same as those of Smalltalk. The two looked almost identical which is the reason why people believe that Smalltalk was the inspiration of “windowed” environments.

The forth and the last is the influence on visual programming tools. Many of these tools look similar to those of Smalltalk tools. The development environment of Smalltalk was used as a reference for creating these tools(42) (43)(44)(45) (46) .

## Syntax and Semantics

Smalltalk syntax has only 6 words that are predefined. The words are true, false, nil, self, super and thisContext. The programmer cannot change what these variables stand for. Smalltalk is a language that uses a message sender and receiver. The keywords self and super are used to point to the sender of a message where self is receiver of the message while super is for the superclass that defines this message.

Smalltalk uses English like syntax unlike other common programming languages. It uses a period to end a statement. The language has built in message sends, assignment, method returns as well as literal syntax. Message sends are based on 3 parts, a sender, a receiver and the message to be sent. Most Smalltalk statement consists of this structure. Others are just unary operations that only have a receiver and a message. Below is a table with sample Smalltalk code covering a few basics of the syntax and semantics of the language.

| | |
|---|---|
| $A | The character A |
| 'Hello Dissertation Paper' | A string |
| #( 1 2 3) | An array of integers |
| \|var\| | Defining a variable |
| \|var\|<br>var := $a. | Defining a variable and setting it to the character 'a' |
| 5 factorial | Sending the message factorial to value 5. Unary message example. |
| 2 raisedTo: 3 | The message raisedTo is sent to the value 2 with argument 3 |
| \|rect\|<br>rect := Rectangle new. | Assigning the variable rect to be a new rectangle Object |
| rect width: 10 height: 15 | Assigning the width and height of the rectangle object rect. |

Table 1 Sample Smalltalk Code

## Pharo Environment

Smalltalk uses a different environment than the familiar IDE (like Netbeans, Eclipse or Visual Studio). Smalltalk uses a windowed environment for development that runs images. It can be said that Smalltalk environments work as virtual machines. There are several environments for Smalltalk development each with little variations to the Smalltalk language. Among these are Pharo, Squeak, GNU Smalltalk, VisualWorks and Dolphin Smalltalk. Pharo is the most stable environment for Smalltalk development which is the main reason of choosing it over all the others.

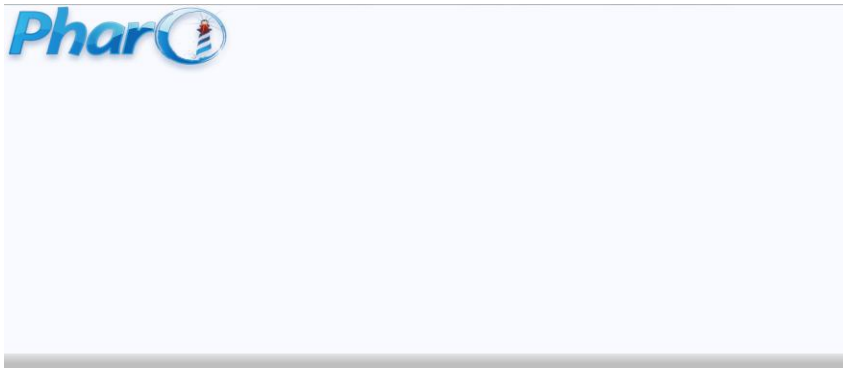


Figure 14 Pharo Environment New Image

In figure 14 we can see the Pharo environment when a clean image is loaded. It looks similar to a computer desktop that has no icons on it. Using the mouse it is possible to left and right click to open windows and choose from them.

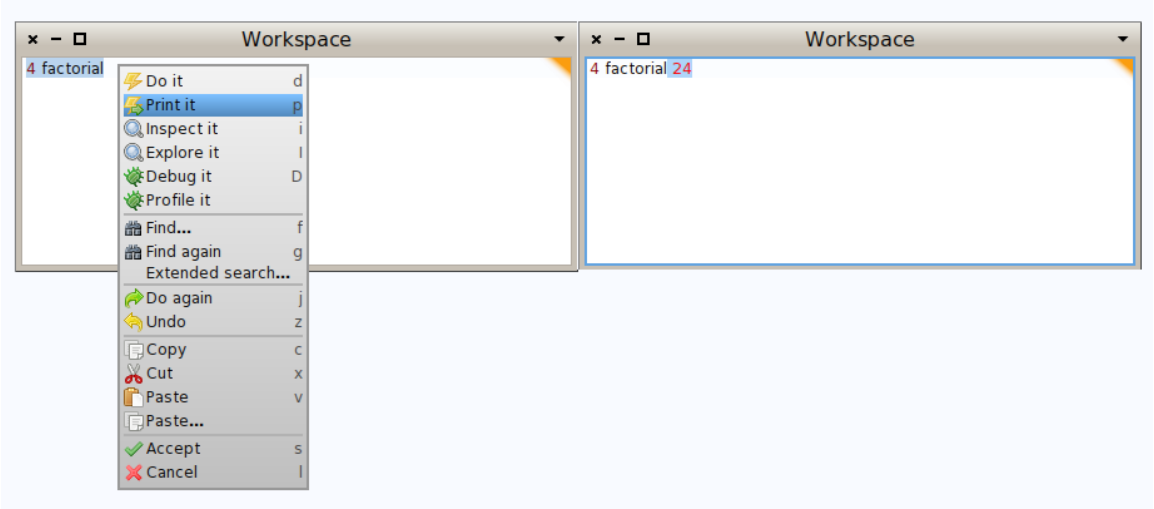



Figure 15 Workspace

By hitting right click and choosing workspace we get the window in Figure 15 above. The workspace is where any running of an application happens. We can see a very simple message being called which is the factorial that was mentioned in the table above. By highlighting the whole message and right clicking on it we get a pop up window with a set of actions to choose from. In this case the action “print it” was chosen which gave the result shown in the second window on the right. The message 4 factorial evaluates to 24. Choosing “do it” instead of print it would run the statement but would not print the result. An example to when to use “do it” is when there is a message that adds new

classes and packages or finds dependencies to the environment. Explore it and inspect it would open a window showing information about what is highlighted. These two actions are usually used with objects. The windows will show all the information about the variables each in a different way. All the object variables and their values can be seen and even inspected or explored themselves as well. “Debug it” will open a debugger window to go through the statements of the code one by one while being able to see all variables and objects

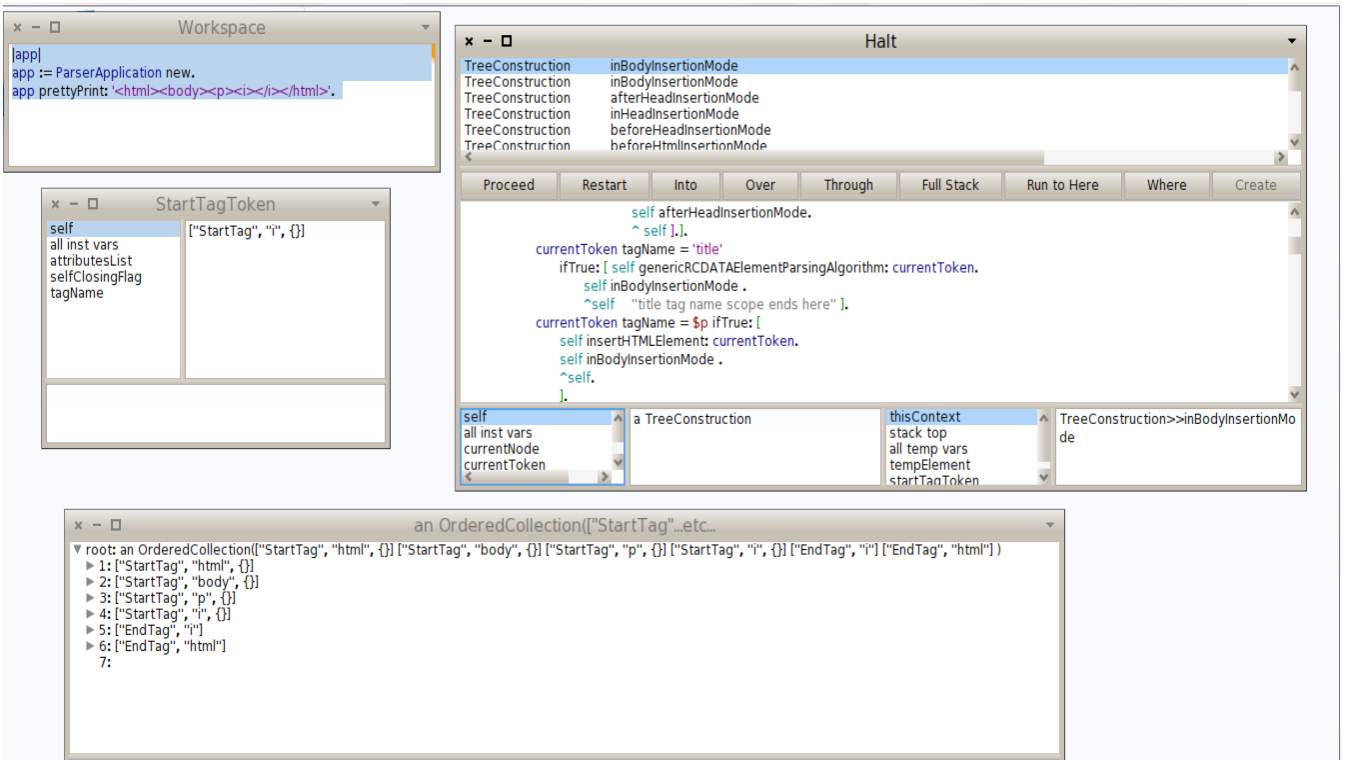



Figure 16 Debugger

In figure 16 we can see what is mentioned above. The window with title workspace is the operation that is being debugged. Next to it is a bigger window with title halt. This window is the debugger window. The actions of the debugger are the ones labeled proceed, restart, into, over, through, full stack, run to here, and where. Below it is the code being debugged. The 4 smaller boxes in the bottom show the class variables on the left and the method variables on the right. On the left, the window labeled StartagToken, is the window that is shown upon inspecting the currentToken variable.

We can see that the token is a start tag token. In the left part of the window are all the variables of the start tag token and pressing on each would show its value in the right window. Below it is another window labeled an OrderedCollection. This is the window shown upon exploring the variable tokens which is an array list of tokens passed by the tokenizer. We can see how the OrderedCollection is shown to use with all the elements inside it.

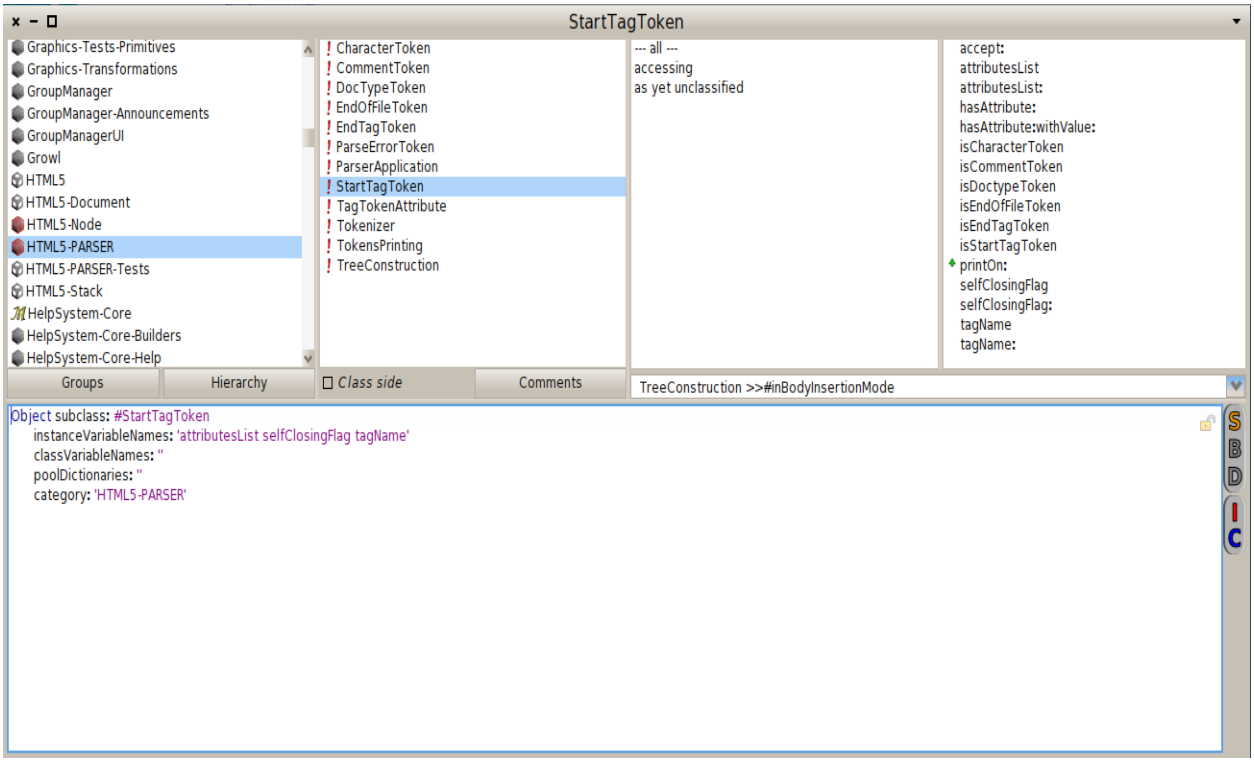



Figure 17 System Browser

In figure 17 we can see the system browser. The system browser is where all packages are stored and where all the Smalltalk code goes. The system browser is separated in 5 major sections, four in the top, and one in the bottom. The top left panel is for the packages. The one opened in the photo is that of the project called HTML5-Parser. Next to it is the set of classes of this package. This includes for example the set of token classes which can be seen in this part. The third panel has what is called categories in Smalltalk. This is a kind of classifying methods together under one name. By default there are three categories. First one is Category “all” which has all the methods of the

class. The second is category accessing which has all the accessor methods or what is more commonly known as setters and getters. The third category is the unclassified which has all the methods that are not put under a category except, of course; the “all” category. The forth panel is where the methods are. We can see the setters and getters of the StartTagToken class as well as different methods like printOn which defines how the class is printed when a print is called. The last panel in the bottom is where the code goes. Since in this figure the StarTagToken class is chosen what we see in this panel is the code for this class. The class name as well as instance variables, class variables, pool dictionaries as well as the package the class belongs to, which in this case is HTML5-Parser.

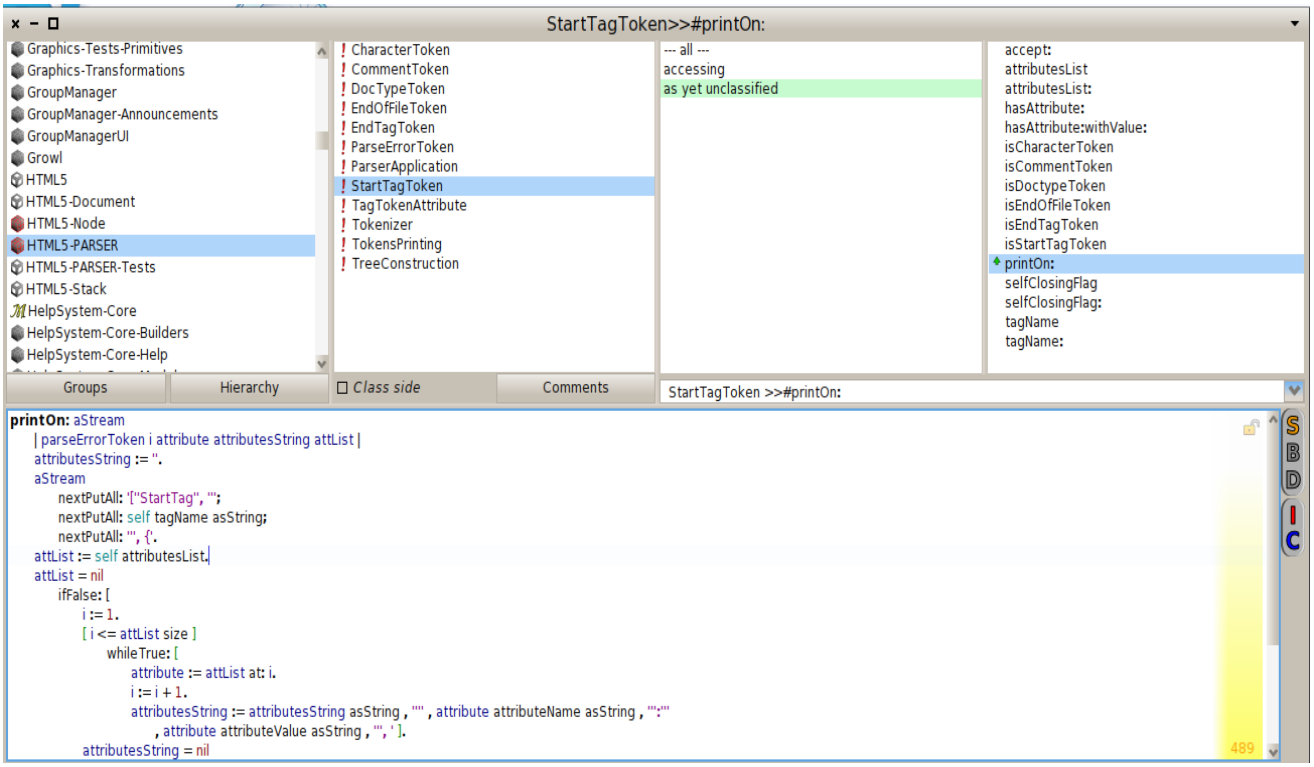



Figure 18 System Browser

In figure 18 we see the same window but with the printOn method chosen. The bottom window now shows the code for this method. Coding takes place in this panel.

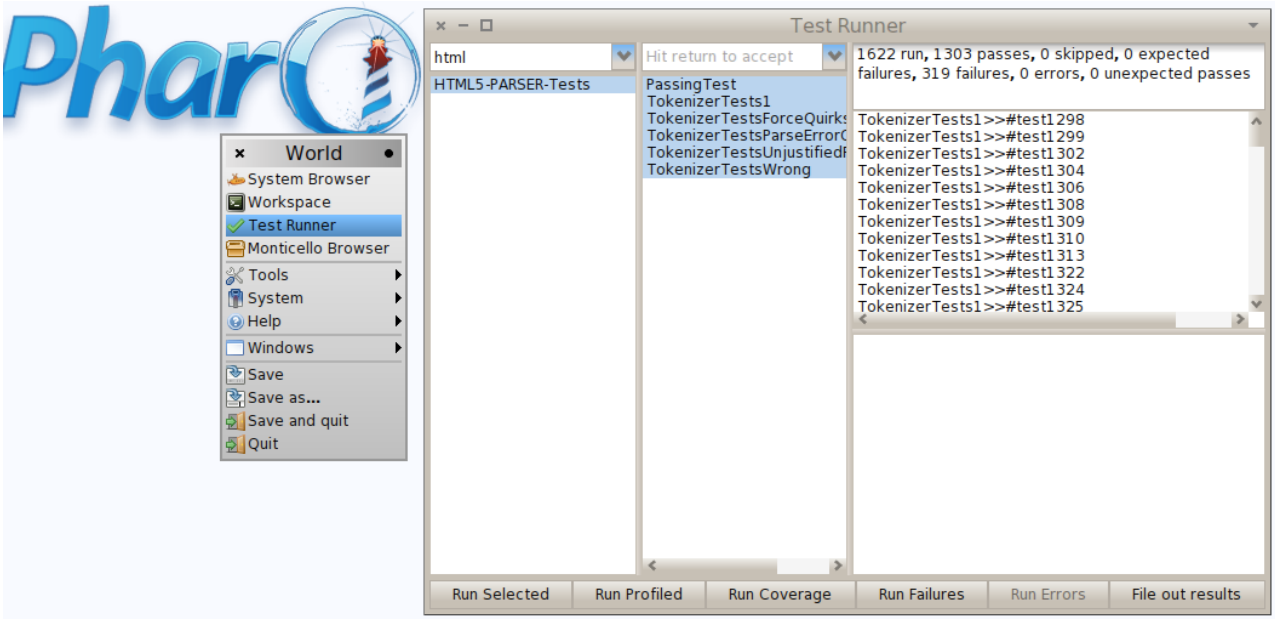



Figure 19 Testing Framework

Figure 19 shows the testing framework. Upon right clicking and choosing test runner from the window the test runner window opens showing all the tests inside this image of Smalltalk. In the image we can see the tests for the tokenizer. The left first panel shows all the testing packages. Next to it are all the categories for the tests. The figure shows a few categories for tests that separate tests between the problematic wrong ones and the passing ones. By hitting run selected all the tests are run and the results can be seen in the upper right most panel. 1622 run, 1303 passes and 319 failures. The failing tests can be debugged from this framework to check why the tests are failing.

## Why Smalltalk?

Since the project is mostly related to the implementation and since the pseudo-code for the product is available, the project is going to be more oriented towards programming and choosing a language is a primary thing in this project.

The language to be chosen should not have a HTML5 parser. It should also be a language that is in need for a HTML5 parser. A survey was held on many programming languages out of which only twenty three were kept as they seemed the most common.

| Language | HTML5 Parser |
|---|---|
| C# (C Sharp) | YES |
| C/C++ | Yes |
| Pascal | Probably no |
| ASP | YES |
| PHP | Probably no |
| Java | YES |
| Python | YES |
| Perl | YES |
| Ruby | YES |
| Tcl | Probably no |
| Caml, Ocalm | Yes |
| Scala | Probably no |
| NetRexx | Probably no |
| Cobol | No |
| Fortran | Probably no |
| Lisp | YES |
| Scheme | Probably no |
| Prolog | Probably no |
| Haskell | YES |
| Forth | probably no |
| Smalltalk | No |
| Modula | Probably no |
| Oberon | Probably no |

Table 2 Languages Survey Results

Out of the twenty three languages, eleven already have implemented parsers. Three of the languages with implemented parsers – C, C++ and C# – depend on an implementation written in Java and converted to the respective languages using the Google Web Toolkit. For the other twelve languages, no sources were found that talk about an implementation of a parser and this is why these entries are marked as probably no. Going over the languages a decision was made to go with Smalltalk for several reasons. The first reason is because the Smalltalk community is in need for a parser. Having a parser is of great importance for any community. Parsers are used in programming to read documents and produce parse trees as well as writing to files. Programmers will then be able to use these parse trees to manipulate them and thus changing the document. The parser can then be used to add elements and tags to the

documents and generate new ones dynamically. A simple example is a SAX or DOM parsers in Java which can be used to read and write to XML documents. These parsers can be used to read, manipulate and generate XML documents according to the preference of the programmer. Applications can also need to depend on parsers to both read and write HTML files. Furthermore, after getting in contact with the Smalltalk community, many people were interested in the project and some were even suggesting ways of building the parser. This would be really helpful in asking these members for code reviews of the project and get feedback from them in order to improve the quality of the code being delivered. Another reason to why choosing Smalltalk is because Smalltalk is a new language which is not familiar. Choosing C would have been an option since C depends on a JAVA based implementation of the parser and not a C based implementation. In addition the C community would prefer a C based implementation of the parser so it would also be of benefit of the community. This option was left out because C is not a completely new language to learn and benefit from unlike the case with Smalltalk. Moreover, a JAVA implementation is being used so an alternative is available. On the other hand, there is no alternative or work around available for the Smalltalk community.

More over Smalltalk is language used in so many Web related applications. Seaside is one of the most famous. Seaside is an application framework written in Smalltalk used to develop Web applications in Smalltalk. It has a component architecture which is used in order to build Web pages from trees or components. (12)(47) AIDAWeb is another framework similar to seaside that is also used to build complicated Web applications.(48) It is also used to build Web applications using Smalltalk. Other Web related frameworks written in Smalltalk are Iliad and Swazoo(49)(50). Iliad is a little bit different from Seaside and AIDAWeb by the fact that is has reusable widget that can be reused in building Web application. Swazoo is a Smalltalk Web server that has many frameworks and applications implemented on it like both AIDAWeb and Seaside.

Smalltalk has good support to the Web and is rich in frameworks that help build Web applications and thus the need for applications that can be helpful to this environment. A parser would be a very beneficial project for this community and especially to individuals working with the Web and using the above mentioned frameworks and Web servers.

# Parser Implementation

## Initial Steps

### *Familiarizing with HTML5 and Parsing*

HTML5 and parsing are completely new subjects and acquiring knowledge about them was a basic step before starting with the implementation. Reading about HTML5 on the WHATWG Website was a great source of valuable information about HTML5 and all the features it has. Udacity offers also a free online course about HTML and Web browsers given by Westley Weimer, a professor of Computer Science at the University of Virginia. The course was a magnificent source of information about parsing and how it happens in browsers. The course teaches about many aspects of HTML in general and HTML5 specifically and especially error handling. As this is the hardest aspect of building an HTML5 parser further research was done on error handling.

During the tree construction phase, the html parser has to deal with any errors found and fix them before creating the tree. All sorts of errors, from missing tags to missnested tags going through broken tags and all kinds of mistakes that could appear should be handled in the tree construction stage.

### *Understanding Pseudo Code*

The first step was familiarizing with the pseudo code on the WHATWG Website. The pseudo code is quite different from the usual “English like” pseudo code. The pseudo code was not trivial to understand. It then became clear that the pseudo code describes a state machine where each part describes the behavior of each state of the machine. The state machine is constituted from sixty nine states where the start state is called the

Data State. Each state starts by reading the next character. According to the character a decision is made. In general, states emit tokens or throw parse errors as well as moving into other states.

## *Java Implementation*

Along with the learning phase, an initial implementation phase was taking place. A Java implementation of the tokenizer was thought to be a good way to familiarize with the specifications and understand how the tokenizer works as a whole. The reason behind using Java as a start is because it is a familiar language to start with. Moreover doing an initial implementation with Java would certainly help familiarizing with the specs and to fully understanding them before the shift to a completely new language which would be much more difficult with both aspects of new language and complicated specifications. A design can also be put for the tokenizer and tested by the initial implementation. The design was quite simple where each state is represented as a method. Each method takes as parameter a character input and accordingly decides what to do next. The state might switch to another state by calling another method, throw a parse error or emit a token into an array list that will contain all the tokens produced. Since efficiency was not currently a concern the implementation started using this design. The implementation revealed many hidden aspects in the specifications that one cannot notice by just reading. The general idea of how the parser works became much clearer as well as the relation between the states. Two weeks into the process the specifications became pretty much clear and reading them and understanding them was no more a hard issue. Due to these facts, it was now possible to start the Smalltalk implementation.

# Smalltalk Implementation

## *Tokenizer Phase*

### Familiarizing With Smalltalk

The implementation phase required a training stage for Smalltalk. The training stage had different branches from choosing the best environment, to using the environment which is different than usual IDE's to the actual Smalltalk syntax and semantics.

The Pharo by Example book as well as video tutorials and built in tutorials inside the Pharo environment, revealed many features of the environment. Enough knowledge about writing, and debugging Smalltalk code as well as running tests to start the implementation was acquired. During the implementation phase many things became clear including many tools and frameworks. The Pharo environment has so many built in tools that developers can use. One of these features is a framework called Petit Parser .

### Petit Parser

Petit Parser is a built in framework inside Pharo that is made specifically to build parsers. For a quick read the framework seemed interesting especially with all the things it can offer(51). The first and most important reason is that it makes work more efficient and easier. Since the framework is built for building parsers then it has built in functionalities that had to be implemented. Moreover it is easier to maintain in the long run than the regular code. Since Petit Parser relies on a grammar, it is sufficient to do the changes to the grammar that is being used. On the other hand, in the case of code, the developer has to go over the code and do the changes in all the necessary places. So in other words, Petit Parser requires a grammar, and in my case it is a grammar for HTML5 which is nonexistent yet. As mentioned earlier, the HTML5 parser specifications are the first specifications not to rely on a grammar unlike all the previous versions. Trying to build a grammar for HTML5 did not seem a good idea especially since some claim that it is almost impossible to find a grammar for HTML5. Trevor Jim wrote in a page called "A grammar for HTML5"on his personal Website that the grammar for HTML5 can be

simply represented by HTML5 = .* where the dot represents any character and the star as the zero or more repetitions of it. This is because HTML5 can accept any input and has the specifications to deal with it. (52)

With Petit Parser out of the question coding continued. The learning curve was steep with lots of obstacles on the way to get a working tokenizer. Reasons ranged from the language which is completely new, to the environment that is not so easy deal with going through the specifications which in most cases turned out to be confusing. Other than the code of the tokenizer, a set of specific token types had to be built to represent each type of token. Since different tokens had different characteristics, a generic token type would not be able to handle all the token types. At the end the tokenizer implementation was complete and the next step was testing.

## Implementation Time Required

Progress was going according to plan during the implementation. All throughout the process, time was being recorded to have an estimate of the time consumed on doing the project as well as having estimates of time required to accomplish further goals in the project. The time was being recorded to finish every single state. The states can be grouped into similar states. Similar states almost have the same code but with different method calls or with some extra code. An example is the DOCTYPE System Identifier Single Quoted State and the DOCTYPE System Identifier Double Quoted State. The two states have the same code except that in one we check if the character is an apostrophe whereas in the other we check if it is a quotation mark. Because of this issue, some states take much less time than others. On average in a case similar to this one the first state would take between 2 to 3 hours. The one similar to it would take between 20 to 30 minutes. The fastest was a state done in 2 minutes which was a state similar to an already done state but with minor changes like the one mentioned above. The longest took around 21 hours in total. The state had a lot of backtracking and checking earlier tokens. The first implementation of this state was wrong and took around 8 hours. After figuring out the mistake, another implementation was added which is supposedly correct and took 13 hours to complete. Time to do things related to the project was

being monitored as well. Among these things are figuring out how to do things in Smalltalk, code refactoring or even researching to figure out ambiguities in the specifications. All of these things together totaled up to around 27 hours. In total getting the tokenizer part of the project done took around 200 hours.

## Testing

After the tokenizer implementation was done testing the tokenizer started. Testing is split into two testing types. First is manual testing to ensure the tokenizer is functioning and producing output. After ensuring the tokenizer runs the second stage will start which is using automated testing using SUnit framework inside the pharo environment.

The first step in order to get ready for testing is building a method that prints the tokens according to the WHATWG tokens printing standard. After this method was done, testing started.

### *Manual Testing*

The first stage of manual testing has resulted in finding quite a good number of mistakes. The mistakes and errors were a result of the lack of experience with Smalltalk and the Pharo environment in general. The structure of the language makes it easy to do mistakes. Unlike in Java or C++, forgetting a delimiter in Smalltalk does not cause any errors because the code will still be valid. Using manual testing most of these mistakes were revealed. Lots of refactoring took place at this stage with many aspects of Smalltalk getting clearer and eventually resulted in a running tokenizer. Now that the tokenizer is running, the next step of testing has to start: automated testing.

### *Automated Testing*

Automated testing will be the last part of the tokenizer implementation process before moving to the tree construction. Automated testing will be based on two kinds of test. The tests that are already provided by the HTML5 community on html5lib which provide specific tests to test certain states in order to insure covering as much cases as possible. Since the tokenizer represents a state machine which has a finite number of cases then testing every bit of the code is possible. As part of the automation building a method to

automatically read the test files is going to be considered. Since the number of test cases is going to be huge it is not a good idea to pass the files manually. This part was not done using Smalltalk. Java was used to implement a reader to read these files and generate a single output file that represents Smalltalk test cases code that can be filed in into Pharo. The reason behind not using Smalltalk is that it would be much more complicated and would take much more time especially that this would not be a contribution to the actual final result of the project. Even while using Java, this wasn't a very easy job because of the many different formats for the expected output for different token types. Different authors also have different ways of writing test cases which required having to modify the code to cope with the changes in the input type.

```
3  {"description":"Correct Doctype lowercase",
4  "input":"<!DOCTYPE html>",
5  "output":[["DOCTYPE", "html", null, null, true]]},
6
19 {"description":"Truncated doctype start",
20 "input":"<!DOC>",
21 "output":["ParseError", ["Comment", "DOC"]]},
22
43 {"description":"Start Tag w/attribute no quotes",
44 "input":"<h a=b>",
45 "output":[["StartTag", "h", {"a":"b"}]]},
46
47 {"description":"Start/End Tag",
48 "input":"<h></h>",
49 "output":[["StartTag", "h", {}], ["EndTag", "h"]]},
50
51 {"description":"Two unclosed start tags",
52 "input":"<p>One<p>Two",
53 "output":[["StartTag", "p", {}], ["Character", "One"], ["StartTag", "p", {}], ["Character", "Two"]]},
```

Figure 20 Tests on HTML5Lib

We can see in figure 20 different tests for different token types. Each token type has a different way of printing its output. Moreover, attributes also require adding them when present or leaving the brackets empty when not found.

SUnit was used for the automated testing. SUnit is a framework for testing in Smalltalk used for automated testing(53). Pharo has a built in tutorial about Sunit. The Pharo By Example book as well has a SUnit chapter that explains how to use it. SUnit is almost the same as JUnit which is familiar and not completely new. As a matter of fact, JUnit is actually derived from SUnit.

## Test Difficulties

After running the first phases of testing, more than one thousand five hundred tests where either failing or giving an error. Going through the tests revealed a tremendous range of coding mistakes. From Calling wrong methods or not calling them at all to forgetting to add tokens, as well as not incrementing the reader and lots of other mistakes that were due to the misunderstanding of the way Smalltalk works. After doing lots of refactoring to the code, things got a little bit better with fewer errors. By going over the failing test cases, it was obvious that some tests were failing because the output and the expected output where not exactly matching because of either extra/less spaces or extra/less brackets. A refactoring step for both the tests as well as the code to print the output was taken.

The tests were not all consistent. A simple example is that some tests had a space between a comma and a left curly bracket where as others did not. Some tests had left and right empty curly brackets '{}' to show an empty set of attributes where as others just left the field empty with a space.

After fixing all these problems, more progress on the testing part was happening. Further work revealed lots of mistakes in the tests due to several reasons. The major and most common problems only will be discussed.

### *Nil and Null*

The first thing to notice in the tests was that any null values are represented by 'null'. In Smalltalk any null values are represented by 'nil' so all of the null values should be changed to nil. This caused quite a small amount of tests to pass.

### *Character Unicode Values*

The second thing to notice was that characters were represented using Unicode values and not actual characters. The tokenizer would read the Unicode value as a set of character and thus resulting in wrong output which would be the set of characters representing the Unicode value. Another refactoring process took place to change all Unicode values to the respective character representation.

## *Character Token Representation*

After changing all the Unicode values, some tests related to characters where still failing. The reason was because any sequence of two or more characters was being represented by one character token. Here is one of the tests showing what was stated above.

{"description": "Valid Unicode character U+10000",

"input": "\uD800\uDC00",

"output": [["Character", "\uD800\uDC00"]]}

The correct output for this test should actually be two character tokens as follows

{"description": "Valid Unicode character U+10000",

"input": "\uD800\uDC00",

"output": [["Character", "\uD800\uDC00"]["Character", "\uDC00"]]}

All these tests had to be changed. The writer of the tests merged all characters following each other into one character token which is not true because the specifications ask to emit a character token for every character that is read.

## *Force Quirk Flag*

Doctype tokens have an attribute called Force Quirk Flag. According to the specifications the value is initialized to off when the token is created and set to the value on in some cases. The tests provided did not have any of the values as on or off. Alternatively they were set as true or false in the test cases. False was changed to off and true was changed to on. After doing this change only a few number of tests passed. Looking at the failing tests, it was obvious that the force quirk flag value was the only difference between the actual output and the expected output taken from the tests. Going over several tests there were around 400 failing tests because of this reason. One by one going over the specifications showed that most of the test cases were wrong. In some cases there were mistakes with my code where I don't set the force quirk flag and those have been corrected. After making sure that indeed there are mistakes in the test cases force quirk flag values, they were all changed to the correct values.

### *Parse Error Count and Unjustified Parse Errors*

Parse errors were also a major problem in the tests. There were basically 2 basic problems. The tests would either have a wrong number of parse errors or unjustified parser errors. Some tests would have 3 parse errors were as the output would only show 2 or 2 parse errors while the output has only 1. This was the case for many tests. Moreover, some tests would have parse errors even though the input would be error free. Another stage of checking the input against the specifications resulted also in finding mistakes in the code as well as mistakes in the tests. Parse errors were not added in some places but still some tests were still wrong and had to be changed.

### *Completely Wrong Output*

The last set of problems that occurred a lot with tests was related to tests with completely wrong output. Here is an example of one of these tests:

{"description":"<a a A>",

"input":"<a a A>",

"output":["ParseError", ["StartTag", "a", {"a":""}]]}

The output should actually be : '[["StartTag", "a", {"a":"", "a":""}]]' . This is also a case where an unjustified parse error is found. The input clearly has three a's whereas the output only has two only. There are around 20 to 30 tests with totally wrong output.

## Tests Feedback

After finishing the testing phase the writers of the tests have been contacted to ask about the conceptions. Unfortunately the reply was that they are not active anymore on the HTML5Lib and suggested to open a topic on the html5lib-discuss Google group(54). Until now there is still no reply from anyone in the working group about this topic. The question can be found here: https://groups.google.com/forum/#!searchin/html5lib-discuss/Tokenizer$20Tests$20Bugs

The characters are thought to be represented all in one character token just for simplicity reasons so that you don't get a huge amount of tokens. As for the parse errors, it is probably because these parse errors occur during the tree construction stage. Since the tokenizer and the tree construction actually work together, it is possible to have parse errors from the tree construction stage. This was true in some cases. The input would result in a parse error during the construction stage and would have one extra parse error in the token. This is not true for all cases though and probably not true but just a note.

## *Tree Construction*

### Implementation

Similar to the start of the tokenizer implementation stage, implementation started with reading the specifications. The specifications for the tree construction stage turned out to be really complicated. Understanding the general mechanism was not as easy as that of the tokenizer, though they specifications represent a state machine. The tree construction machine is represented by 23 insertion modes for HTML elements as well as rules for dealing with foreign elements. These 23 modes also depend on other very complicated algorithms. These algorithms are related to creating nodes, updating the correct location to add nodes, adding closed tags to elements with no closed tags as well as rearranging the nodes and changing their location and many others.

As it was obvious that having an overall view of how the tree construction works in general, a decision was made to try and implement it step by step by completing a set of elements that can be parsed by the parser. As the first step, the basic HTML elements were chosen consisting of the most common elements of HTML. After finishing this set of elements more elements can be added until all the elements of the HTML are covered.

The same methodology used in the tokenizer stage is to be used in the tree construction stage. Each insertion mode would be represented as a separate method that takes a token and does some computation on it as required by the specifications. Before this

could start a few things had to be done. First of all an HTML document had to be used. Since Smalltalk already has packages for XML, a new HTMLDocument class was added as a subclass of XMLDocument class. The HTMLDocument class has all the required variables that are to be used during the tree construction stage. The tree construction stage also depends on a several number of nodes specific to HTML elements. Each node represents an HTML element. Because there are so many nodes, the initial implementation will feature a generic element that has all the general attributes required for the basic elements to be covered.

Going over the states one by one, all the cases that cover the required HTML elements mentioned earlier were implemented. This means two things. First of all the states that can be reached by chosen set of elements are the ones implemented. For example the "in table" insertion mode can only be reached after reading a table element which is currently not part of the elements covered so it is left out. In other insertion modes, some tokens that are also not part of the set specified can be found but are not covered. An example is receiving a comment token from the tokenizer. Comments are not part of the set provided so it is not covered in insertion modes that are reachable by the elements of that set.

## Testing

Tree construction testing started as manual testing. During the process of implementation test would be run to check the functionality and for any possible bugs. After a fair amount of work has been done in the tree construction part, more work was put into testing. The SUnit framework was also used for this purpose. Tree construction has 5316 tests that cover the elements html, head, body and p. The expected output for the DOM tree is taken from an online application called HTML5 Live Dom Viewer which can be accessed here http://livedom.validator.nu/. The tests revealed an enormous number of bugs in the code especially in the basic functions of the tree construction stage. More manual testing was done to check for the correctness of the application in dealing with the rest of the elements covered in the project. The test suite also includes more than 100 tests for the parts that are not covered yet. These tests can be used later

on when the project is continued in order to check for the coverage of elements. These tests alone are not enough but can give an indication of what elements haven't been covered yet. More testing should be done to cover more cases for every element covered.

## Code Statistics

As most of the project is about coding, coding statistics provide good evidence to how good or bad the job was done. The parser is not complete and thus most of the code is that of the tokenizer. Using the file out method in Pharo to generate a text file of the code of the whole project showed that there is around 7500 lines of code for the parser. This includes the code for the tokenizer and tree construction states, as well as all the helper methods and classes required for both phases of parsing.

| | |
|---|---|
| Tokenizer pseudo code | 1083 lines |
| Tokenizer pseudo code transliteration | 4078 lines |
| Tokenizer structures and helper methods code | 750 lines |
| Tokenizer tests code | 13191 lines |
| Tokenizer test count | 1622 tests |
| Tree construction code transliteration | 876 lines |
| Tree Construction structures and helper methods code | 469 lines |
| Tree Construction test code | 68540 lines |
| Tree construction test count | 5214 tests |

Table 3 Code Statistics

### *Tokenizer*

The pseudo code provided for the tokenizer; i.e. that of the 68 states, cover 1083 lines of code on the WHATWG Website. The Smalltalk code written for this phase covers 4078 lines of code. This is almost four times more than the provided pseudo code. The graphs of Figure 1 Figure 2 Figure 3 and Figure 4 in appendix 1 show the exact number of lines of code for every state as well as the lines for the pseudo code as bar graphs. Added to those are around 750 lines of code for the tokens structure and other helper functions such as tokens printing code.

### *Tokenizer Testing*

Tokenizer testing spans a big amount of coding with 13191 lines of code for 1622 different tests. Test code was generated, as mentioned earlier; using a program written in Java to read test input taken from the HTML5Lib and give Smalltalk SUnit code to be imported into the SUnit framework.

### *Tree Construction*

Tree construction code cannot be compared in the same way since it is not totally complete. The code generated for the tree construction insertion modes covered 876 lines of code. Helper methods and other functions cover 476 lines of code. All in all the tree construction code covers 1279 lines of code.

### *Tree Construction Test*

Tree construction test code has the biggest amount of lines of code with 68540 lines of code covering 5214 tests. The tests cover the elements html, head, body, and p closing and opening tags. The tests were used to check the functionality of the parser as a whole as well as the helper methods used by the tree construction. Helper methods responsible for updating the location to add an element and inserting elements are the major parts of the parser as they are used in all states. The tests represent all possible permutations of the elements up to strings of size 4 elements as well as some other tests to cover other parts of the tree construction stage.

### Parser Coverage

The parser being submitted is not a complete parser but covers a partial set of elements. What has been covered are the elements html, head, Body, a, p, b, and i as well as doctype elements comments and characters. Though this is true, the implementation should be able to deal with all elements. The parser checks elements against a list of uncovered elements and if the element is among those not covered, the element is simply ignored and the token after it is processed.

## Project Application

A simple application is built that uses the parser that shows how the parser can be used. The application uses the Glamour framework which is a framework that is used to build applications. The application is quite simple.

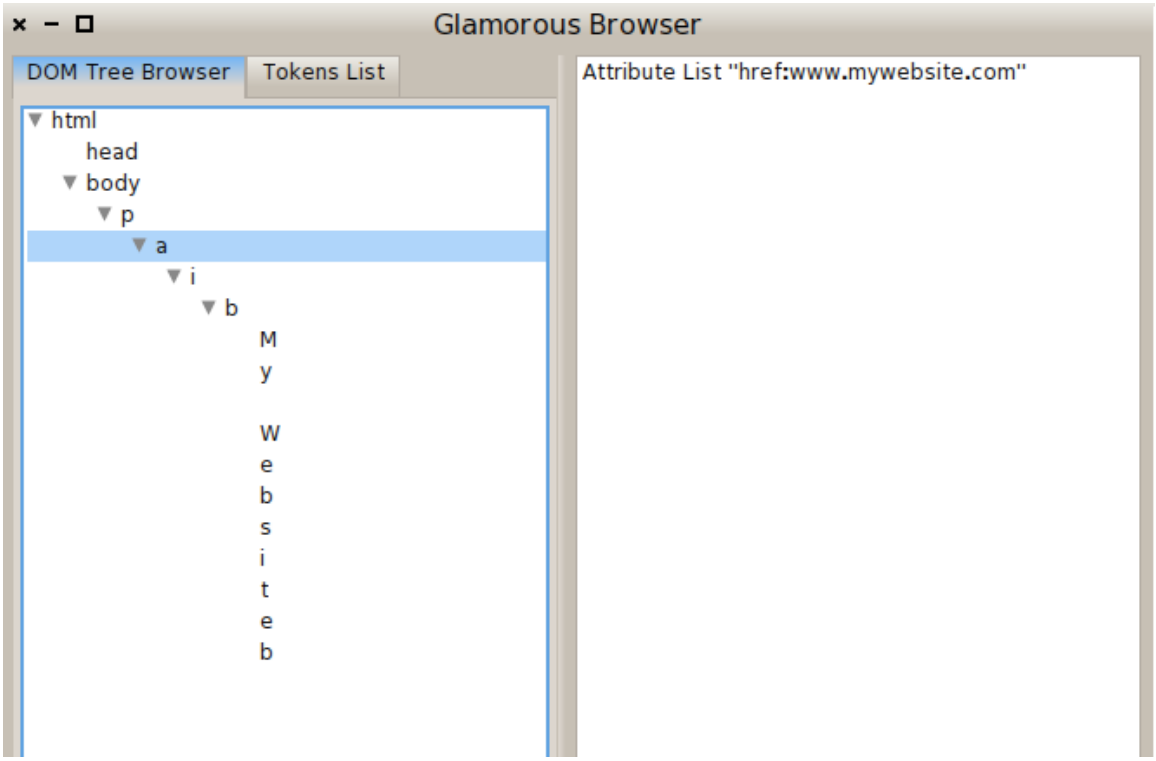



Figure 21 Dom Tree of String <html><head><body> <a href="www.myWebsite.com"><b>My Website<b></a>

The window is split into two panels like in figure 21 above. The panel on the left shows the tree structure of the DOM tree that can be expanded or contracted. When an element is selected the second panel shows the variables of the element like the attributes it has for example.

The window also has another tab. That tab displays the list of tokens that resulted from the tokenization stage of parsing which can be seen in figure 22. This way the user can see the tokens produced and the DOM tree they evaluate to.

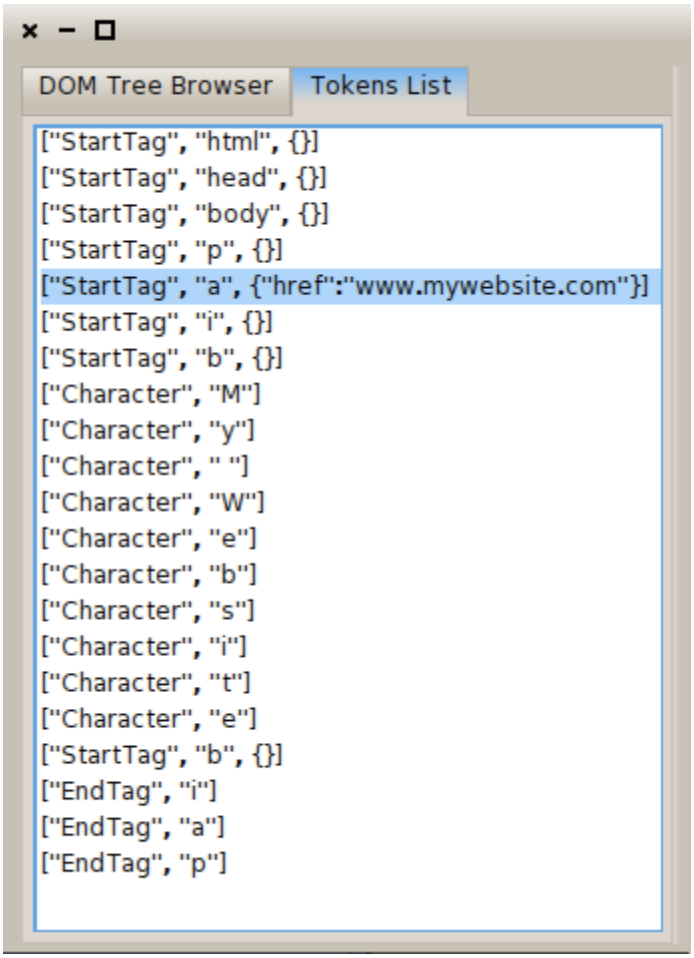



Figure 22 Tokens of String <html><head><body> <a href="www.myWebsite.com"><b>My Website<b></a>

# Specifications Feedback

## Tokenizer Specifications

The journey with the specifications starts with understanding them. It was not an obvious to understand how the tokenizer actually works. At first it might not seem to be the most efficient and simplest way to explain the specifications but as soon a general idea of what is happening is formed it becomes quite obvious. The tokenizer stage does not require lots of complexity in the implementation which makes understanding how the overall tokenizer works possible. The specifications follow the same pattern and same kinds of steps to deal with similar input in several states.

Even though all what is mentioned above is true, there could be some improvement to some minor confusing things in the specifications. In some conditions where the same state has to be preserved, the specifications do not reflect that. The pseudo-code would just show how to process the input. It might be possible to do things this way if it is true that not stating a change in state would always mean staying in the same state but it is not true. In some cases when a null or an end of file character is read, tokenization has to stop. So in some cases not putting a state change might mean staying in the same state or stopping the tokenization process. This is something to be aware of during the transliteration process. It was a frequent mistake found in the tokenizer code where going back to the same state had to occur rather than termination.

Another something to point out is consistency. Even though the specifications follow the same pattern, it sometimes has different text for the same procedure. A very simple one is reading an end of file character in the script data escaped state and script data escaped dash state. In the first one the specifications say the following *"Switch to the data state. Parse error. Reconsume the EOF character."* Where as in the second, it would say *"Parse error. Switch to the data state. Reconsume the EOF character.".* There are several cases similar to this one. Note that both have to result in the same output even though the steps are not the same which leads us to the last point.

Usually specifications and pseudo-code follow the order of execution for the algorithm. In the tokenizer specifications this is not true. We noticed in the earlier example that in one case the switch to data state was before throwing the parse error. Changing the state should be the last thing to do after all the processing is done. An example is the character reference in data state shown in figure 23.

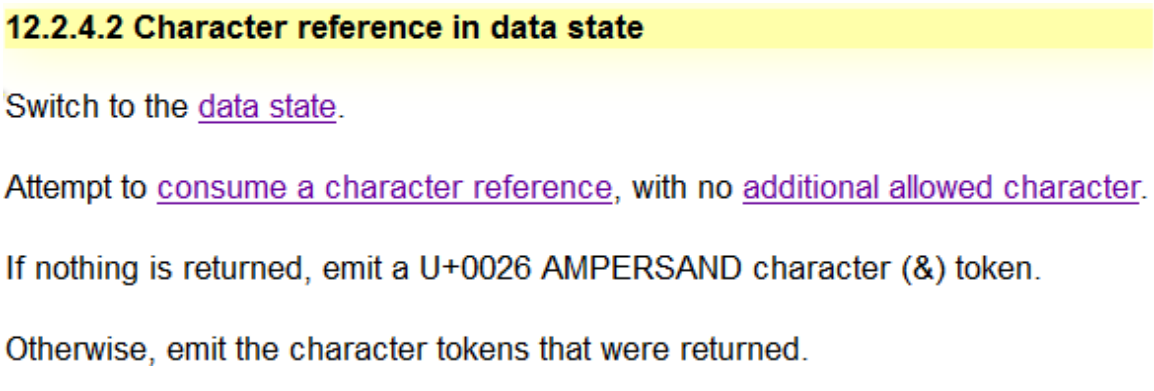



Figure 23 Tokenizer Character Reference in Data State(55)

From a first look, a reader would get an impression that switching state has to be done first and then do the processing part which is actually wrong. Moreover this is not the case with all states. In some cases changing the state comes last rather than first which makes things even more confusing.

## Tree Construction Specifications

Just as in the tokenization stage, the first step was understanding the specifications. This turned out to be almost an impossible thing to do looking at the complexity of the tree construction algorithm. Unlike the tokenizer stage, the tree construction stage depends on many algorithms, objects and conditions which make it so hard to have a general idea of how it works. The introduction does not provide an explanation about how the tree construction stage works but rather directly starts describing the algorithms to adding different kinds of elements and algorithms that going to be used during this stage. Lots of pages are hyperlinked in order to show pseudo-code or explain a structure.

The first thing to notice in this part is the description of the stack of open elements being a stack that grows downwards. The bottommost element in the stack is the last element added to the stack where as the upper most is the element added first. So it is basically a regular stack and during the implementation it becomes clear that indeed it is a regular stack structure. It is quite confusing to call it a stack that grows downward. A

stack can be thought to grow in any direction as long as the top of the stack is the last element added.

The structure of writing the specifications is the same as that of the tokenization stage but not very neat. One would expect to see inside an insertion mode all cases with the same token type after each other like all the start tag tokens after each other followed by end tag tokens and so on. In the specifications it looks more like a random permutation where the tokens are interchanged. This is not a very serious problem but annoying when having to do the transliteration.

The pseudo-code explains almost everything needed, even though it might be in a complicated way; the specifications do not cover what a marker element is. In many states the process requires looking for marker elements that are not defined in the specs. The specifications have hyperlinks on every bit that has an explanation or a definition either on the same page or on some other page except a marker element.

The pseudo-code also assumes some knowledge about the domain of the Web and HTML. It is not written for any programmer to read and translate into code. On the other hand it is also not possible to explain everything because that would probably be enormously big and too detailed. The specifications cover enough details to explain how the implementation has to happen.

# Project Feedback

## Project Limitations

Encoding is the first limitation in the project that is not covered. Covering encoding first of all is a difficult task to achieve. Algorithms that deal with encoding are fairly complicated with an encoder and decoder for each encoding type. Another thing is that covering encoding would not add to the value or benefit to the parser. Covering encoding is a challenge by itself and this is why it was not covered.

Scripting is the second limitation. Dealing with scripting is a complicated task that would require a JavaScript implementation. Moreover scripting will require in some cases to change the state of the tokenizer meaning a connection for communication between the tokenizer and tree construction should be established and that is also very difficult to achieve. The most important thing is that the parser is not being implemented to be used inside a browser but for applications. Applications that use parser do not require scripting and adding it would also not be of great value to the goal of the project.

Other than the technical limitations there were many other limitations that did not allow building a complete parser. First of all is the time constraint. Implementation of the tokenizer took more time than expected which left less time to build the tree construction part of the parser. Second is the fact that the tree construction stage is much more complex than that of the tokenizer. Even with better knowledge of the Smalltalk language and how to deal with it, implementation was on a slower pace than that of the tokenizer.

## Project Difficulties

Due to many different factors, many difficulties were faced during the process of building the parser. The first reason is the lack of knowledge in the field of HTML. The specifications are written for people who have knowledge in the field and understanding why things are they way they are was taking a lot of time and effort. The second is the Smalltalk language. The Smalltalk language is completely new. Having experience with languages like Java and C++ makes it harder to understand how Smalltalk works. Many aspects of the language were confusing especially with the sender and receiver way of calling methods. The testing phase of the tokenizer revealed many bugs in the code and by fixing them many new aspects of the language became clearer. Many things could have been done in a much easier way and in less time in the tokenizer stage. The third is related to the specifications. The specifications are not so easy to understand especially with little knowledge in the field of the Web. The specifications for the tokenizer become obvious after understanding the structure of the tokenizer but this is not the case for the tree construction stage. With every new state

covered things become harder and more complicated. Algorithms are complicated and depend on many factors as well as other algorithms which make getting things right without lots of testing hard. The last is the little experience in the field of software engineering. Being a fresh graduate with very little experience in software engineering in general and in programming in big projects, building a HTML5 parser was a very challenging project. The programming experience was limited to building algorithms to do a certain job where in most cases a stub is given and the final output is clear and quite simple compared to this project.

Moreover, James Graham claims that a HTML5 parser can be completed in weeks.(56) After working on the project, the claim seems to be most likely unachievable. Let us build some assumptions. The tokenizer pseudo code spans 1083 lines of code. Assuming each line of pseudo code transliterates into 1 line of actual code then the tokenizer would require 1083 lines of code. According to David Veksler programmers write around 10-12 lines of code per day that go into the final product at the end regardless of the skill of the programmer.(57) Let us assume that since the pseudo code is there and the programmer only has to do a transliteration to a certain language a programmer can write 5 times more. That would be 50-60 lines of code per day. Doing simple division we get that a programmer needs between 18 to 22 days to complete the tokenizer which is almost 3 weeks to build the easy bit of the parser. Note that the assumptions assume a 1 to 1 mapping between the pseudo code and the actual code as well as not taking into consideration building all the structures required for the tokenizer. From this it becomes obvious that it is highly improbable that a parser can be built in two weeks.

## What is Next?

The first thing to do in order to complete the project would be refactoring the tokenizer. The tokenizer code can be refactored to cover less lines of code as it is in many places redundant. This would be a good way to start familiarizing with the Smalltalk framework in general and with the way the tokenizer works. The second step is checking the states that are covered. Going over each state one by one and check the specifications for the missing parts. The missing parts are flagged in the code and should be easy to spot. This

will also give a good idea about the architecture of the code and how it functions and works together. The tests can then be used to insure nothing is broken. Filling the missing parts also includes coding up the method for throwing a parse error. This can be a first step in actual implementation. The method calls for throwing errors are in the code; the body of the method has to be implemented. When all the above is done, then all the states currently covered are complete.

Now in order to cover the rest of the states, the best way is to check all the states that are reachable from the states covered. In other words, assuming states A, B and C are covered and E, F, G, H, and I are still not covered, look at the states that can be reached from any of the states A, B, C even if some other state looks simpler and would look like a good state to start with as a beginning. Many errors show up during the implementation and many ambiguities become clear and a state that is not reachable from any of the covered states means it cannot be tested and this is where my advice comes from. Now when the state is chosen, the next thing to think about is what to start implementing first. Moving by type of tokens is would be a good idea. Cover comment and doctype tokens first. Then start covering start tag tokens element by element. Then jump to end tag tokens and so on. Once the state is complete add tests to the framework that cover the cases in the covered state. Fix any bugs and make sure the state is doing the intended job. When the cycle is done, move forward into another state and repeat the same process until all the states are covered.

## Project Code

### *Tokenizer Code*

Looking at the tokenizer code now, it is obvious that much better can be done. The tokenizer code was the first part to be implemented and the language was still not so clear. The first thing is that variables that are used in most of the states can be added as global variables and initialized in the initialize method. The tokenizer code creates new variables in initializes them in every state. So in all states, the first few lines are just initializing variables. The second thing is adding helper methods to replace redundant code. The tokenizer code for example does not use a method to create tokens and add

them to the list of tokens. Every time a new token has to be added a new token is created and the variables of the token are added one by one. One method for every token type can be added to replace lots of lines of code in the tokenizer.

### *Tree Construction Code*

Unlike the code in the tokenizer stage, the code is much better. With the language becoming more familiar and many of its aspects becoming familiar the quality of the code improved. The two points mentioned in the previous part do not appear in the tree construction code. The tree construction probably does not require refactoring as most functions are embedded into methods no matter how small they are.

### *Test Code and Framework*

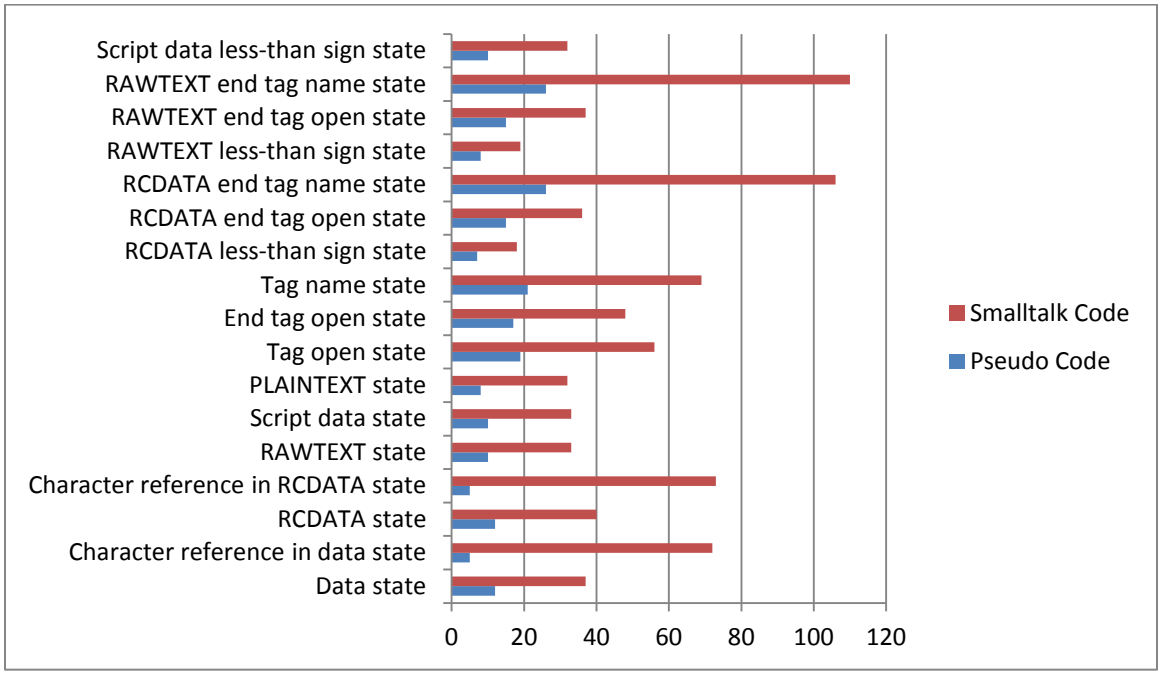
Testing code and the testing framework are also of good quality. The fact that the test code for both the tokenizer and the tree construction depends on comparing strings makes things a little bit harder but it ensures the output is correct. The tree construction tests for example compare a string representation of the DOM tree. The string shows the hierarchy of the nodes by using new line and tab characters. Sometimes the test would fail because of an added space at the end of the string which might seem annoying but on the other hand having a strict test is always better. Assume we use a minification of this output where you get a string sequence of the elements in the correct order. Though this test would be testing the correct order of elements, it fails showing the hierarchy of elements which is very important.

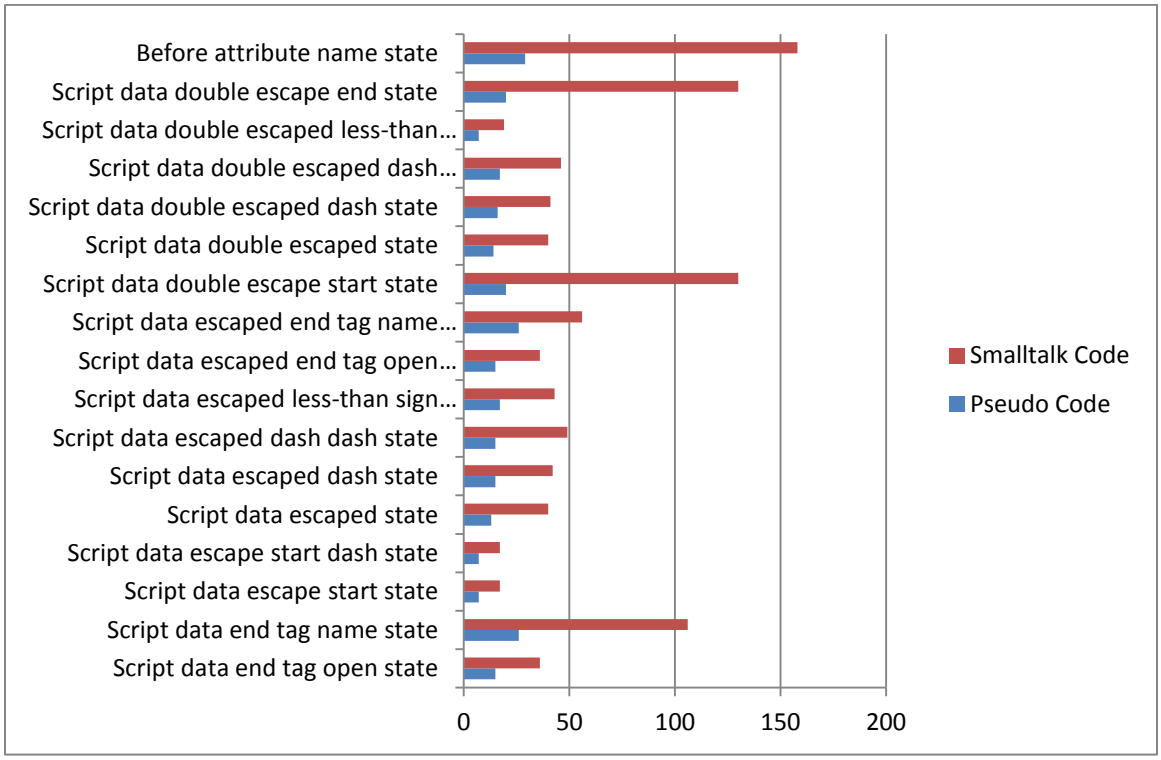
SUnit on the other hand is also an excellent framework for testing. It is very useful for keeping track of passing/failing test between one test run and another. It is also a fast framework with really good performance. Running around 8000 tests would take around 5 seconds on a normal personal computer which is quite impressing knowing the amount of computation required to finish each test.

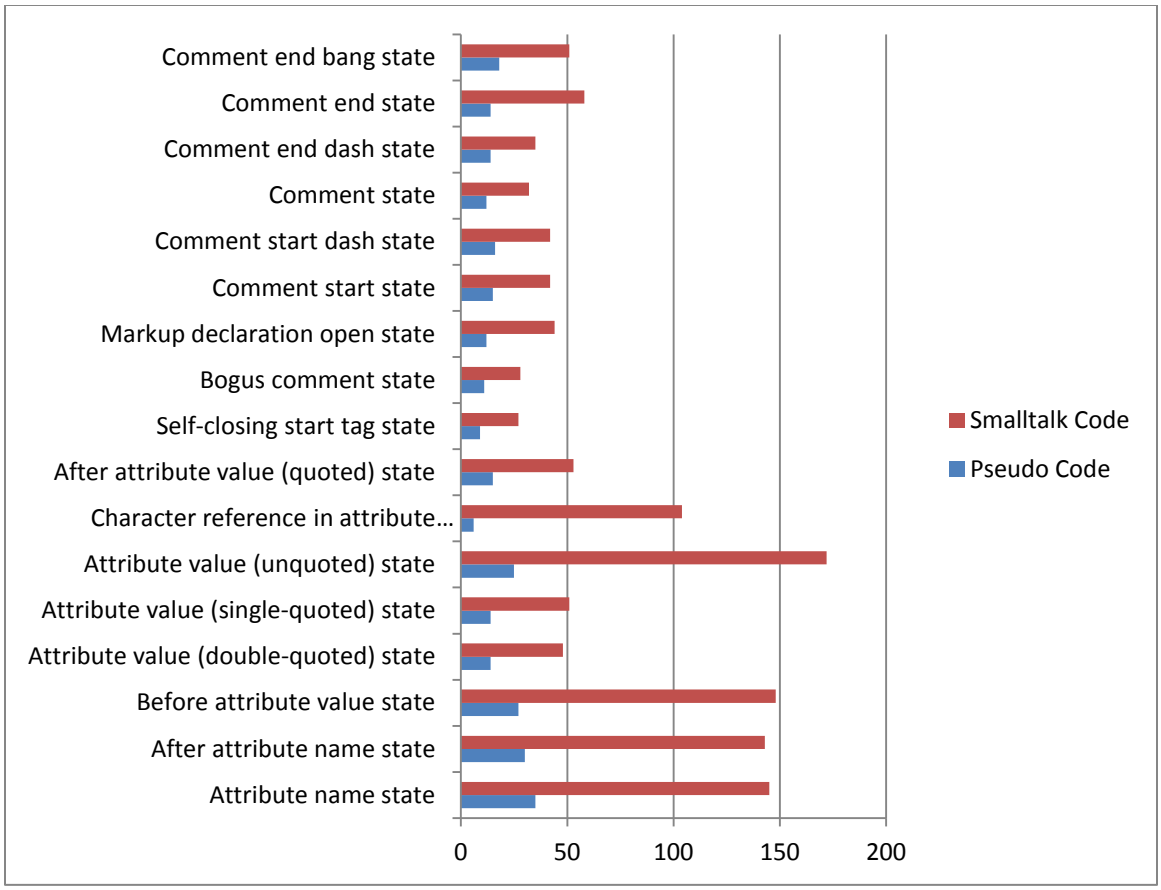
For those reasons keeping the test code and the framework would be something I recommend as well as using the test to check if anything has been broken when any changes are done.

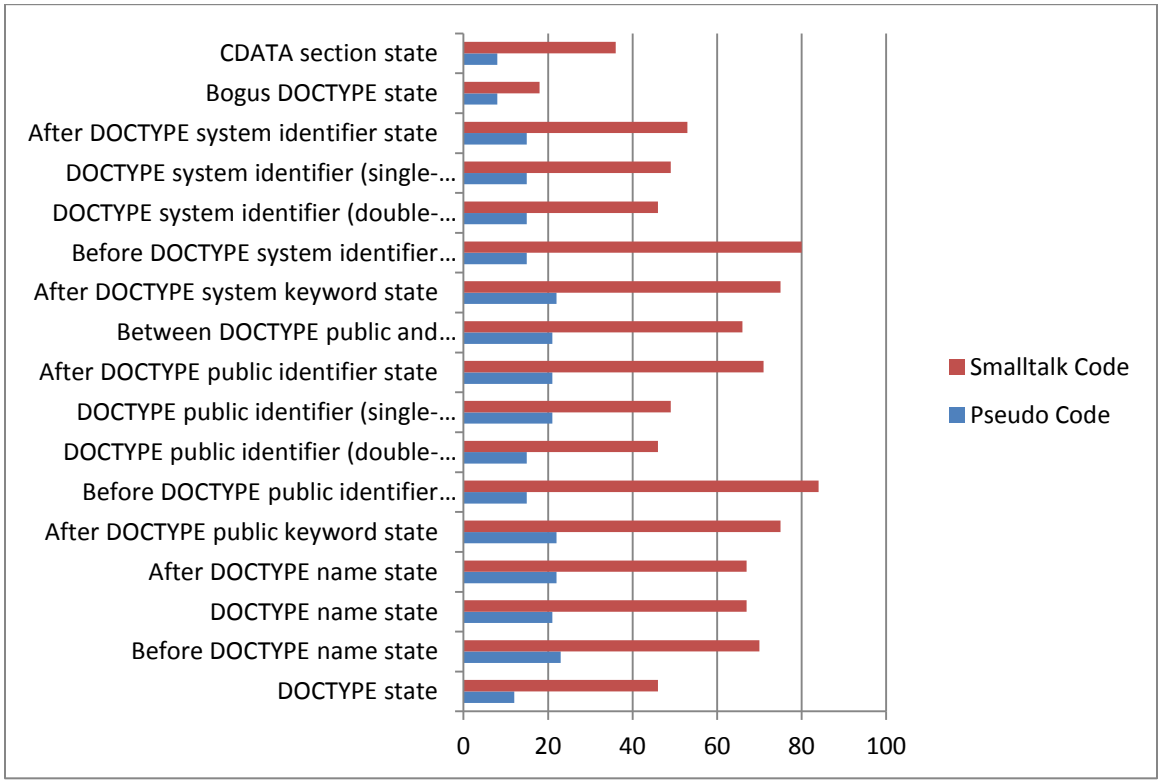
# Appendix 1

**Tokenizer Code Stats Charts**




Comment end bang state
Comment end state
Comment end dash state
Comment state
Comment start dash state
Comment start state
Markup declaration open state
Bogus comment state
Self-closing start tag state
After attribute value (quoted) state
Character reference in attribute...
Attribute value (unquoted) state
Attribute value (single-quoted) state
Attribute value (double-quoted) state
Before attribute value state
After attribute name state
Attribute name state
0
50
100
150
200
Smalltalk Code
Pseudo Code


CDATA section state
Bogus DOCTYPE state
After DOCTYPE system identifier state
DOCTYPE system identifier (single-...
DOCTYPE system identifier (double-...
Before DOCTYPE system identifier...
After DOCTYPE system keyword state
Between DOCTYPE public and...
After DOCTYPE public identifier state
DOCTYPE public identifier (single-...
DOCTYPE public identifier (double-...
Before DOCTYPE public identifier...
After DOCTYPE public keyword state
After DOCTYPE name state
DOCTYPE name state
Before DOCTYPE name state
DOCTYPE state
0
20
40
60
80
100
Smalltalk Code
Pseudo Code

1. **HTML5WorkingGroup.** Plan 2014. *W3C.* [Online]

2. **Hong, Dan, Chen, Shan and Shen, Vincent.** *An Experimental Study on Validation Problems with Existing HTML Webpages.* Hong Kong : s.n., 2005.

3. **Hickson, Ian.** *Several messages about HTML5.* 2007.

4. **Crockford, Douglas.** JSMin The JavaScript Minifier. [Online] 2003. http://www.crockford.com/javascript/jsmin.html.

5. **Symantec.** *Symantec Internet Security Threat Report.* California : Symantec Corporation, 2008.

6. **Brady, Pádraic.** Regex HTML Sanitisation: Off With Its Head! *Astrum Futura.* [Online] March 2011. http://blog.astrumfutura.com/2011/03/regex-html-sanitisation-off-with-its-head/.

7. **Bates, Daniel, Barth, Adam and Jackson, Collin.** *Regular Expressions Considered Harmful in Client-Side XSS Filters.* Raleigh, North Carolina : IW3C2, 2010.

8. TextLint. [Online] http://textlint.lukas-renggli.ch/.

9. **Girba, Tudar.** *Moose: The Book.* 4th. s.l. : Pier, 2011.

10. **Kay, Alan.** The Early History of Smalltalk. [Online] April 11, 1993.

11. **Community, Smalltalk.** ANSI Smalltalk Standard. [Online] 1998. http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html.

12. About Seaside. [Online] http://www.seaside.st/about.

13. **TexUsersGroup.** Tex Users Group Website. [Online] http://www.tug.org/.

14. *SCRIPT, An On-Line Manuscript Processing System.* **Madnick, Suart and Moulton, Allen.** 1968, IEEE Transactions on Engineering Writing and Speech.

15. **Goldfarb, Charles F.** The Roots of SGML -- A Personal Recollection. *SGML Source.* [Online] 1996.

16. **Sahuguet, Arnaud.** *Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask.* Penn Database Research Group,, University of Pennsylvania. Glasgow : s.n.

17. **W3Schools.** *Introduction to DTD.*

18. **W3C.** *HTML 4.0 Specification.* 1997.

19. **Watson, Dennis G.** Brief History of Document Markup. *University of Florida.* [Online] June 20, 2005. http://chnm.gmu.edu/digitalhistory/links/pdf/chapter3/3.19a.pdf.

20. **WebStandardsOrganization.** Common Ideas Between HTML and XHTML. [Online] 2003.

21. —. HTML Versus XHTML. [Online] October 2003.

22. **W3Schools.** HTML - XHTML. [Online] http://www.w3schools.com/html/html_xhtml.asp.

23. **W3C.** Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online] November 26, 2008. http://www.w3.org/TR/REC-xml/#sec-guessing-with-ext-info.

24. **Pilgrim, Mark.** XML on the Web Has Failed. *XML.* [Online] 2004.

25. **W3C.** *Draconian Error Handling.* February 9, 2010.

26. **The Mozilla Foundation Opera Software, ASA.** Position Paper for the W3C Workshop on Web Applications and Compound Documents. [Online] 2004. http://www.w3.org/2004/04/webapps-cdf-ws/papers/opera.html.

27. **W3C.** W3C Workshop on Web Applications and Compound Documents. [Online] June 2, 2004. http://www.w3.org/2004/04/webapps-cdf-ws/minutes-20040602.html.

28. **WHATWG.** WHAT open mailing list announcement. [Online] 2004. http://www.whatwg.org/news/start.

29. *Smart Phones Application development using HTML5 and related technologies: A tradeoff between cost and quality.* **Hasan, Yousuf, et al.** May 2012, IJCSI International Journal of Computer Science Issues, p. 7.

30. **Smith, Michael.** *HTML5 Overview.*

31. **Carson, Ryan.** Pro HTML5 tips for 2011. *Net Magazine.* February 2011, p. 41.

32. **van Kesteren, Anne and Pieters, Simon.** HTML5 differences from HTML4. *W3C.* [Online] April 5, 2011. http://www.w3.org/TR/2011/WD-html5-diff-20110405/.

33. **Roggio, Armando.** What Makes HTML5 Special. *Practical E-Commerce.* [Online] April 4, 2011.

34. **Shah, Neil.** One Billion HTML5 Phones to be Sold Worldwide in 2013. *Strategy Analytics.* [Online] December 7, 2011. http://www.strategyanalytics.com/default.aspx?mod=pressreleaseviewer&a0=5145.

35. **Keith, Jeremy.** *HTML5 For Web Designers.* New York, New York : Jeffrey Zeldman, 2010.

36. **W3C.** Plan 2014. *W3C.* [Online] http://dev.w3.org/html5/decision-policy/html5-2014-plan.html#introduction.

37. **Maine, Kurt.** Percentage of Websites Using HTML 5. *Bin Visions.* [Online] September 30, 2011. http://www.binvisions.com/articles/how-many-percentage-web-sites-using-html5/.

38. **Grune, Dick and Jacobs, Ceriel J.H.** Introduction To Parsing. *Parsing Techniques: A Practical Guide.* s.l. : Springer, 2008.

39. **Pandey, Raju.** Syntactic Analysis. *University of California, Davis.* [Online]

40. **Hickson, Ian.** Tag Soup: How UAs handle <x> <y> </x> </y> . [Online] November 21, 2002. http://ln.hixie.ch/?start=1037910467&order=-1&count=1.

41. **WhatWG.** Tokenization. *whatwg.* [Online] http://www.whatwg.org/specs/web-apps/current-work/multipage/tokenization.html#data-state.

42. Mainstream Influence Of Functional Languages. [Online] 2009. http://c2.com/cgi/wiki?MainstreamInfluenceOfFunctionalLanguages.

43. About Ruby. [Online] https://www.ruby-lang.org/en/about/.

44. **Bojčić, Krešimir.** Perl 6 The Holy Grail or Utopia? [Online] February 24, 2013. http://kresimirbojcic.com/2013/02/24/perl-6-the-holy-grail-or-utopia.html.

45. **Liu, Chamond.** *Smalltalk, Objects, and Design.* 1st. Lincoln : toExcel iUniverse, 2010.

46. **Savona, Matt.** *Smalltalk's Influence on Modern Programming.* 2008.

47. **Tate, Bruce.** Crossing borders: Continuations, Web development, and Java programming. [Online] 2005. http://www.ibm.com/developerworks/java/library/j-cb03216/.

48. AIDAweb . [Online] http://www.aidaweb.si/.

49. The Smalltalk web framework for developing modern applications with ease. [Online] http://www.iliadproject.org/.

50. Swazoo Smalltalk Web server. [Online] http://www.swazoo.org/.

51. Petit Parser. [Online] http://www.moosetechnology.org/tools/petitparser.

52. **Jim, Trevor.** A grammar for HTML5. [Online] 2012. http://trevorjim.com/a-grammar-for-html5/.

53. **SUnit, Camp Smalltalk.** SUnit gets a new home! [Online] http://sunit.sourceforge.net/.

54. **Yang, Edward Z.** *HTML5 Tokenizer Tests Bugs.* 2013.

55. **WhatWG.** WhatWG. *Tokenization.* [Online] http://www.whatwg.org/specs/web-apps/current-work/multipage/tokenization.html#character-reference-in-data-state.

56. **Graham, James.** *The non-polyglot elephant in the room.* 2013.

57. **Veksler, David.** *Some lesser-known truths about programming.* Alabama : Ludwig von Mises Institute, 2010.

58. **Hickson, Ian.** *Error handling and Web language design.* 2004.

59. **Lemone, Karen.** Compilers Programming Language Translation. [Online] 2003.

60. **UniversityCollegeDublinComputerScienceDept.** Compiler Construction Lecture. *University College Dublin.* [Online] 2008.

61. **Hickson, Ian.** Tag Soup: How UAs handle. *hixie.ch.* [Online] November 21, 2002. http://ln.hixie.ch/?start=1037910467&order=-1&count=1.

62. **BinVisions.** Percentage of Web sites Using HTML5. *BinVisions.* [Online] Sept 30, 2011. http://www.binvisions.com/articles/how-many-percentage-web-sites-using-html5/.

63. **Wikipedia.** Parsing. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Parsing.