A TESTING STRATEGY FOR HTML5 PARSERS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2015

By
José Armando Zamudio Barrera
School of Computer Science

Contents

Word Count: [18106]

List of Tables

List of Figures

Abstract

As an effort to maximise the interoperability among HTML parsers, both W3C and WHATWG provide a detailed parsing algorithm in their last HTML specification. The implementation and therefore testing of this algorithm may represent a complex task. Up to now, far too little attention has been paid to ensure the compliance of a parsers in relation to the specification. Furthermore, a considerably number of parsers rely on the HTML5lib test suite. This might not be enough to ensure the compliance and thus the interoperability. This dissertation presents an approach to test the level of compliance among several HTML5 parsers. In addition, it validates whether the HTML5lib test suite is enough to guarantee a consistent behaviour among HTML5 parsers implementations. Moreover, the developed testing framework may help to find compliance violations that might not be covered by the HTML5lib test suite. In order to achieve the above, a new parser implementation following the W3C specification was developed in Java and tested against the HTML5Lib test suite for the purpose of understanding the specification and validate its complexity. Secondly, a test framework based on N-version diversity was developed to compare the outputs of a selected group of parsers and determine the most likely correct output by plurality voting. Finally the level of compliance of a parser is determined by its percentage of convergence in relation to the other parsers. Data for this study was collected from the Common Crawl corpus. From a set of selected HTML5 parsers, this study found that in general there is a high probability of convergence, and that this probability is higher if the parsers in context passed the HTML5Lib test suite. The main reason of disagreements was because of specification differences. Moreover, the test framework proved to be useful for finding bugs in the parsers. In conclusion, the test framework developed in this study showed a close approximation of the level of compliance of HTML5 parsers and proved that HTML5Lib test suite does ensure a consistent behaviour among HTML5 parsers, ensuring the compliance with reference to WHATWG specification.

Declaration

No portion of the work referred to in this 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.

Copyright

Acknowledgements

To my project supervisor, Dr. Bijan Parsia, I owe my gratitude for his constant advice, guidance and support throughout the course of this project.

I would also like to express my gratitude to the Mexican National Council for Science and Technology (CONACyT) for awarding me a scholarship and financial benefits that have enabled me commence and complete my MSc studies at The University of Manchester.

Dedication

To my parents, for believing in me and pushing me forward to achieve my goals.

To my tutor Kristian, for your guidance and care.

To my friend, Casey, for your friendship, support and laughter.

To my university colleagues.

To Ana, for your patience, understanding and love.

Glossary

AUT Application Under Test. 7, 25

AWS Amazon Web Services. 54, 55

DOM Document Object Model. 15, 20

DTD Document Type Definition. 21

HTML Hypertext Markup Language. 15, 19

JAXP Java API for XML Processing. 42, 45

JSON JavaScript Object Notation. 35

S3 Simple Storage Service. 53

W3C World Wide Web Consortium. 15, 19

WARC Web ARChive. 39, 54

WHATWG Web Hypertext Application Technology Working Group. 15, 20

XHTML eXtensible Hypertext Markup Language. 19

Chapter 1

Introduction

Over the years Web Technologies have continually evolved in order to enhance communication over the Internet. Groups such as the World Wide Web Consortium (W3C) and Web Hypertext Application Technology Working Group (WHATWG) develop standards to support such communications. One of the most important standards is the Hypertext Markup Language (HTML) specification. Ian Hickson (main editor of the HTML specification) states that the main objective of the specification is " to ensure interoperability, so that authors get the same results on every product that supports the technology" [2]. In order to achieve this goal, the HTML technology has had several changes through history from its creation to its current version: HTML5.

The focus of this study is the parsing section. This section defines the rules to parse an HTML5 document into a Document Object Model (DOM) tree. To ensure interoperability, parser implementations must exactly follow these rules. However, the pseudocode algorithm provided by the specification is a complex combination of state machines. This makes it difficult for implementers to test and ensure the compliance with traditional functional testing methods.

Currently neither W3C nor WHATWG provide a reliable tool or mechanism to validate the conformance to the specification. Hence parser developers resort to test their output with test suites, validators or even by comparing their outputs against another parsers/browsers. A widely used test suite is called *texthtml5lib-tests* which provides a large set of tests with an accessible and platform independent format [3]. Nevertheless this test suite might be not enough to guarantee the conformance against the specification due to the dynamic behaviour of the web and infinity possibility of input cases. Thus a complementary test method is needed.

Additionally to HTML5lib test suites, several attempts have been made to test the

HTML5 parsers. Minamide and Mori[4] achieved to generate automatic test cases, however the specification coverage was limited due to complications with the translation of the algorithm to a grammar, and may not generalise to web content. One possible solution test this complex scenario is with Manolache and Kourie [5] approach to generate approximate test oracles based on on N-version diversity. These studies influenced the method proposed in this study and the design of a testing framework for HTML5 parsers.

The following sections of this chapter show the aim and objectives of the project as well as hypotheses and scope. Finally, the chapter ends with an outline of the structure of this dissertation.

1.1 Aim

Develop a testing framework based on N-version diversity to validate the compliance of HTML5 parsers.

1.2 Objectives

1. Develop a HTML5 compliant parser for the purpose of understanding the architecture and operation of the HTML5 parsing algorithm

2. Pass the HTML5Lib test suite with the developed parser.

3. Develop a test framework based on N-version diversity to validate the compliance of HTML5 parsers.

4. Expose the level of convergence of HTML5 parsers in relation to a set of test cases.

5. Identify and analyse the cause of disagreements.

6. Find compliance violations in the parsers in testing.

7. Find missing tests in HTML5lib test suite.

8. Find bugs in the specification.

1.3 Scope

The scope of the project is limited to the parsing section of the HTML5 specification, and as part of the plan and estimation, it was decided to exclude the encoding and scripting execution sections. Therefore the implementation and test framework was designed to only work with UTF-8 encoding and without script support.

1.4 Team organization

The work presented in this thesis was done by a team, which consisted of three MSc. students. The team worked together to develop the HTML5 compliant parser presented in Chapter 3. Similarly, parsers installation tasks and core components of the test framework were divided between the team. Then the decision was took to work individually in order to achieve different goals. Anaya designed and developed a Tracer from the HTML5 parser [1] whereas I was responsible for obtaining a sample from the Common Crawl data set in order to obtain an approximation of level of convergence and thus of compliance of selected parsers. The latter is shown in Chapter 6. The Appendix A shows a more detailed table listing the activities and work done by each member. Additionally, in the same Appendix a second table shows a log of activities done by myself.

1.5 Dissertation outline

The dissertation is composed of nine chapters. Chapter 1 is this introduction. Chapter 2 begins with the HTML history and looks at how HTML evolved to become what it is nowadays. Additionally it gives a brief introduction to the parsing algorithm and finishes describing related work to testing theory. Chapter 3 describes the design, building and testing of a parser implementation. Similarly, Chapter 4 describes the design and building of the test framework. Chapter 5 shows a detailed survey of existing HTML5 parsers and lists which were selected for evaluation. Chapter 6 introduces to Common Crawl data set and describes the method used to obtain a sample for the test framework. Chapter 7 shows the settings, results and discussion of the experiments realized in the order they were conducted. Chapter 8 presents the overall results and key findings of the study. The last Chapter 9 provides a concise summary of the research, the key findings, implications and future work.

1.6 Terminology

The terms **HTML parser specification** and **specification** refers to the W3C section 8.2 of Parsing HTML documents of the W3C HTML5 recommendation [6], unless indicated otherwise.

The term **MScParser** is used to refer to the parser developed in this project.

Chapter 2

Background and theory

2.1 Introduction to HTML5

This section gives an introduction to HTML5. The first subsection describes the evolution of Hypertext Markup Language (HTML) and explains the reason why it was created. The second subsection details the concept called *draconian error handling*, the reason why HTML5 did not adopt it and its relation to the HTML5 parsing algorithm.

2.1.1 HTML Historical background

The following is a brief description on the history of HTML and the introduction to terms related to this dissertation. According to W3C, HTML is defined as "the Webs core language for creating documents and applications for everyone to use, anywhere" [7]. Nowadays HTML is widely used in the world wide web and it has become a key resource in our daily lives.

HTML has been evolving since its creation by Berners-Lee [8] with the World Wide Web. The first versions were managed by CERN and then by TIEF [9]. With the creation of the World Wide Web Consortium (W3C) in October 1994 [8] the HTML language evolved until HTML4 was published in 1997. Then the consortium decided to work on the XML equivalent in 1998, called eXtensible Hypertext Markup Language (XHTML) [9]. XHTML was the result of W3C decision to move the web towards XML technologies, hoping it would "become a more flexible, machine-readable, stricter, and more extensible way to mark up documents and other data"[10].

One of the characteristics of XHTML is its draconian error handling. This term

refers to the requirement that all well-formedness errors be treated as fatal errors[11]. A well-formedness error is generally understood in this context to mean a violation of a XML rule. The main aim for adopting this XML characteristic into HTML technology was to facilitate developers to write processing programs. This characteristic forced people to be careful while writing HTML documents because any minimal error led to break down the entire web page. The next version W3C started working on called XHTML2, in addition to the draconian error handling, it did not have backwards compatibility at all [9].

Web Hypertext Application Technology Working Group (WHATWG) was born and HTML5 with it as a result of the bad reception of XHTML2. XHTML2 was not being used to its draconian error handling and lack of backwards compatibility[10]. Consequently, an independent group emerged in 2004 called Web Hypertext Application Technology Working Group (WHATWG) concerned about W3C direction. W3C continue working on XHTML2 whereas WHATWG decided to provide their own HTML specification, updating it according to the demand of the applications, users and vendors, while at the same time maintaining backwards compatibility and flexible error handling. In 2007 W3C stopped working on XHTML2 and formed a working group with WHATWG to develop the HTML5 standard[9] although they still provide separate specifications nowadays.

As mentioned above,The W3C and WHATWG are the two organisations that provide standards for HTML. The field of research of this dissertation is the validation of the conformance to the HTML5 parsing, therefore, the section of most interest is *Parsing HTML documents*[1] which defines rules to parse HTML5 documents. This section provides a detailed algorithm to generate a Document Object Model (DOM) tree from the parsing of a HTML5 document or an arbitrary string. The main difference between the W3C and WHATWG versions is that WHATWG specification is a *living standard* which means that does not have versions [12] and is in constant evolution. As a result it can be updated at any time. One possible implication of this is that it makes it difficult to be up to date to the specification.

2.1.2 HTML versus the draconian error handling

As described in the previous subsection, HTML was reborn as HTML5 to be less draconian [11] and more tolerant to errors. The motivation for using a Draconian error

[1] W3C (Section 8.2): `http://www.w3.org/TR/html5/syntax.html#parsing` ,
WHATWG (Section 12.2): `https://html.spec.whatwg.org/multipage/syntax.html#parsing`

handling policy was to allow developers to write simple programs without worrying about the error handling "for all sorts of sloppy end-user practices"[13]. As Bray notes: "The circumstances that produce the Draconian behavior - fatal errors - are all failures to attain the condition of well-formedness. Well-formedness doesn't cost much; the tags have to be balanced, the entities have to be declared, and attributes have to be quoted"[13]. Thus, it was believed that developers were going to follow this measure by writing HTML free of errors. However, history showed that this did not happen. Opposite to XML technologies, HTML was designed to accept and handle several syntax errors [14] and not well formed documents. The motivation can be resumed with Ian Hickson observation: "Authors will write invalid content regardless" [2].

The error tolerance feature of HTML5 is probably one of the reasons why the specification could not be easily described in Document Type Definition (DTD), XML Schema, RelaxNG or other grammar-based schema languages. In other words, HTML5 could not be an XML technology since HTML documents do not meet with the well-formedness constraints given in the XML specification [15]. Therefore, instead of providing a schema, HTML5 is described in form of a pseudo code that explicitly states error recovery rules [2].

What follows is an outline of the HTML5 Parsing Algorithm part of the HTML5 specification introduced in this section.

2.2 HTML5 Parsing Algorithm

The HTML5 Parsing Algorithm defines the rules to generate a DOM tree from a text/html resource[6]. The term DOM is defined by W3C[16] as "a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents". This section moves on to describe the architecture of the parsing algorithm in which a DOM tree is generated from a HTML document. The second part list some differences between the parsing sections of W3C and WHATWG found in the course of this project.

By and large the parser architecture is composed by a byte stream decoder, input stream pre-processor, tokenizer and the tree constructor. Its work flow is shown in the figure 2.1. Although there is a stage called Script execution, the rules to execute a script are in another section of the specification. Moreover, a script engine must be included in the implementation to actually run a script. As our scope currently is limited to parsing without script execution these rules are not covered.

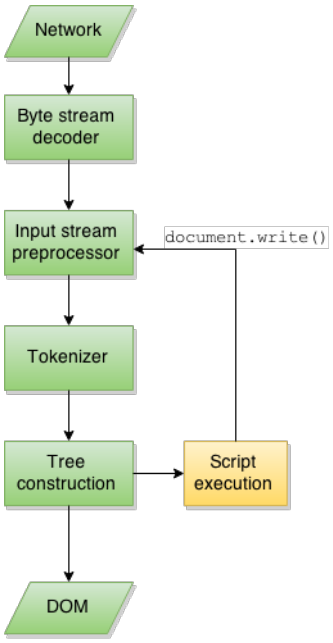


Figure 2.1: HTML5 parser flow diagram. Taken from W3C Recommendation.

The core components in the HTML5 parser are the tokenizer and tree constructor. Basically the tokenizer is responsible for doing the lexical analysis. It receives a stream of input characters and creates tokens. Subsequently the tokens are taken by the tree constructor, which may insert a node into the resulting DOM tree. Another important aspect of the parser is that the construction stage is re-entrant "meaning that while the tree construction stage is handling one token, the tokenizer might be resumed, causing further tokens to be emitted and processed before the first token's processing is complete"[6]. Additionally the specification provides algorithms for encoding detection and a stream preprocessor that prepares the input stream i.e. filtering some invalid Unicode characters.

Both tokenizer and tree constructor components are expressed as deterministic state machines. A state machine, finite state machine or finite state automaton is a structure consisting of a finite set of states (including a start state), a finite set of input symbols called alphabet, a transition function that tells which state to move depending of the input symbol and current state of the machine, and a set of final states. The tokenizer consists of 68 states, the set of Unicode code points [17] is the alphabet and the transitions are defined in every state. On the other hand, the tree constructor consists of 23 states, called insertion modes, tokens are the alphabet and the transitions are also described in every insertion mode.

The HTML5 parser algorithm is based on the Top-Down parsing. A Top-Down parsing reconstruct a parse tree from its root downwards [18]. In the HTML5 parser algorithm, the tree constructor creates a tree by adding as root the element *html* no matter if it comes or not in the input data. Then as it receives more tokens, these are added to the three in a top-down direction. However, due to its error correction capability, nodes could be added in an upper level in the tree. Thus it would be incorrect to affirm that the parsing technique is purely top-down, rather it is top-down derivation.

The HTML5 parser features a robust error handling capability. When a syntax error is encountered, the parser can modify its internal state to continue processing the rest of the input. This is know as error recovery [18]. Furthermore, its capable to correct the input to be syntactically correct, by adding, removing or changing Unicode characters in the input stream or tokens that feed the tree constructor.

Another special feature of the parser is the capability to modify the parser tree by adding, editing or removing tokens as a result of an execution of a script. The parser algorithm allows the execution of scripts while parsing, therefore the construction of the tree does not depend only of the tokens generated by the input document.

Finally, the HTML5 parser could not be easily described by a context-free grammar and thus apply existing parsing techniques. The reasons are the next characteristics:

1. Error handling and correction. A parser tree is always constructed even if the input contains errors.

2. Execution of scripts while parsing. The parser tree is constructed with tokens from the input stream and from the result of scripts execution.

So far this chapter has focused on HTML5 specification and parsing algorithm. The following section will discuss methods for testing HTML parsing and related literature review.

2.3 Testing methods

2.3.1 Functional testing

This section describes and discusses the different methods that attempt to address the aim of this dissertation. These methods are based on functional testing or black box testing as they are intended to test any implementation of the parsing algorithm. According to Borba *et al.* “Functional testing involves two main steps: to identify the

functionalities the product implementation should perform; and to create test cases that are capable of verifying whether such functionalities are fulfilled correctly according to the product specification."[19].

The following are two services that test browsers compliance with HTML and web technologies. The acid3 browser test consists in a test suite for evaluating HTML4, XHTML 1.0, DOM, javascript and CSS technologies[2]. However HTML5 is not included [20]. A web site called html5test [21] provides a score for the browsers, and validates if supports HTML5 tokenizer and tree construction, though methodology information is not provided.

A currently active HTML5 test suite maintained by HTML5lib developers contains more than 3000 test cases to test the parsing algorithm. The HTML5lib test suite[3] consists in two groups; tokenizer tests and tree construction tests. It is commonly used by parsers implementations e.g. cl-html5-parser [22], Ragnarok [23], even WebKit developers contributed with more than 250 test cases [24]. Although these tests were originally created for the HTML5Lib parser, they were designed to be used in any platform[25].

Holler *et al.*[26] used a Fuzz testing approach to test script interpreters, i.e. Javascript and PHP, by generating random code using a grammar and previous failing programs. This approach is difficult to apply to the HTML5 parser as there is no grammar yet that represents the complex algorithm. Nevertheless an alternative to this method is to develop an algorithm-based software to generate random test data.

Minamide and Mori[4] achieved to generate HTML documents to test the compatibility across browsers and the consistency of the specification with their *reachability analyzer* based on a translation of the algorithm into a conditional pushdown system. However, this method has two limitations. Firstly, the translation only covers part of the tree constructor, 24 HTML elements and 9 insertion modes. Secondly, formatting element behaviour is excluded. Minamide and Mori stated that the formatting element was excluded "because of difficulties with the destructive manipulation of the stack". In spite of this, the produced conditional push down system consisted in 487 states that evidences how complex can be an equivalent grammar for the full algorithm.

[2]http://acid3.acidtests.org/

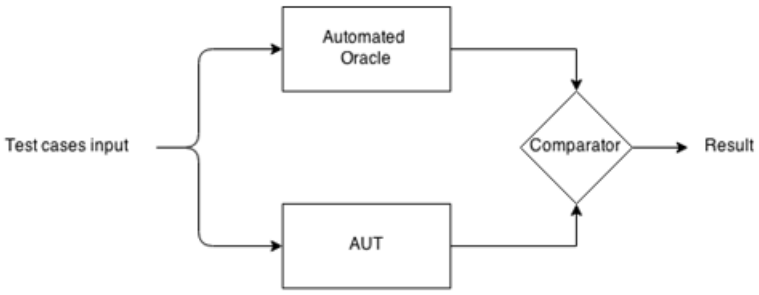


Figure 2.2: Automated Oracle test process. A test case is processed by an Automated Oracle that produces an expected result. The result is compared with the AUT result in order to find any possible failure.

2.3.2 Oracle testing

An oracle test is defined by Shahamiri *et al.* [27] as "a reliable source of expected outputs". The oracle test "must provide correct outputs for any input specified in software specifications"[27]. Even though these outputs could be generated manually (human oracles), it is recommended to use an Automated Oracle instead to ensure the testing quality while reducing testing costs[27].

Shahamiri *et al.* [27] describe a general oracle process with the following activities:

1. Generate expected outputs.
2. Save the generated outputs.
3. Execute the test cases.
4. Compare expected and actual outputs.
5. Decide if there is a fault or not.

By using an Automated Oracle in the process, as shown in 2.2, the correct expected results can be automatically generated for a set of inputs. If the Application Under Test (AUT) generates a different output then this is a possible failure. However, as Shahamiri *et al.* [27] highlights, the Automated oracle should behave like the software under test in order to provide a reliable oracle.

Shahamiri *et al.* [27] realized a comparison analysis in automated oracle studies. The ones relevant to this dissertation are described below.

Manolache and Kourie [5] proposed a N-Version Diverse Systems and M-Model Program Testing. These methods relies on independent implementations of the AUT.

N-Version Diverse Systems method requires whole software implementation whereas the M-Model Program Testing method focus on specific functions of the software to be tested. Shahamiri *et al.*[27] consider this costly and not highly reliable. Additionally as Manolache and Kourie [5] warn, a key concern of N-Version Diverse Systems are the "*common-mode failures* or *coincident errors*" which "occur when versions erroneously agree on outputs".

The second method is using a decision table. The decision table consist on a list of a combination of input conditions and their respective output. As Shahamiri *et al.* [27] mention, a limitation is the generation of manual outputs. Furthermore, it may not be convenient if the combination of inputs are very large and complex.

The third study is an Info Fuzzy Network (IFN) Regression Tester. By using Artificial Intelligence methods, Last *et al.* [28] simulated the AUT behaviour to be a test oracle. A previous version of the UAT receives random inputs and the outputs are used to train the IFN model in order to be used as a automated oracle. This method depends on a reliable previous version of the UAT and therefore is only applicable to regression testing.

2.4 Summary

This chapter has reviewed the key aspects that surrounds HTML5 and its parsing algorithm. It showed how error handling was a key concern in HTML5 and thus in the design of the parsing algorithm.

Furthermore, testing methods studies relevant to this study were described. As commented before, the selected method for the testing framework is based on N-Version Diverse Systems. Chapter 4 details why this method was selected and how was implemented.

The chapter that follows moves on to describe the methods for the design and implementation of a HTML5 parser.

Chapter 3

HTML5 parser implementation

In order to fully understand the HTML5 parser specification functionality and concepts as well as the implications and challenges, it was decided that the best approach was to develop our own implementation.

The main requirement for the implementation is to be compliant with the specification. The scope includes the tokenization and tree construction stages, and consequently, the testing is focused on these components. Therefore, decoder and scripting stages of the parsing architecture are not covered. These activities were not added to the project plan due time limitation and therefore were left for future work.

3.1 Design

3.1.1 Overview

As described earlier, the parser is composed of a decoder, pre-processor, tokenizer and tree constructor. The latter is capable of calling a script engine to run scripts. The input is a stream of data and the output a DOM tree. The Appendix B shows the overall design of the parser. Although both decoder and script execution steps were not implemented a brief explanation will be given in order to have the overall picture of the parser.

The role of the stream decoder is to receive a stream of data and converted into a stream of Unicode code points depending on the input character encoding. A Unicode code point can be a Unicode scalar value or an isolated surrogate code point . The length of a Unicode code point is a byte unless is a pair of a high surrogate followed by a low surrogate which must be treated as a single code point, even though the length is

two bytes. In order to determine the character encoding the specification provides an algorithm called ”encoding sniffing algorithm”. In this dissertation the term character refers to a Unicode code point for the sake of readability.

The stream of characters must be preprocessed before passed to the tokenizer stage. The pre-processor validates invalid Unicode characters or isolated surrogates by raising a parse error and prepares the stream of characters, i.e. by removing 'CR' (carriage return) characters for the next stage the tokenizer.

The next step is to pass the stream of characters to the tokenizer stage. The tokenizer will attempt to create tokens from the stream. The tokenizer may create several tokens; however, only when the algorithm indicate to emit a token, this is sent to the tree constructor.

The tree constructor may receive one or more tokens. Similar to the tokenizer, the tree constructor will attempt to create nodes and append them to the DOM tree from the tokens emitted by the tokenizer. When finishing processing tokens, the control is returned to the tokenizer unless the tree constructor triggers the stop parsing procedure.

When the stop parsing procedure is triggered, the parser run the steps indicated in the algorithm and then the DOM tree is returned.

As the HTML5 parser accept invalid documents, any parse error is clearly indicated in the algorithm and these do not trigger the parser to stop. In this design any error is added to a list or errors and optionally displayed in the log.

A parser context was designed in the parser life cycle to hold several elements that are shared among the components described above. It contains the states of the tokenizer and tree constructor objects, i.e. current tokenizer state and current insertion mode. The reason is that one object can alter the state of the other one when some conditions are met. The tree in Figure 3.1 shows the different types of elements that form part of the parser context. When the parser is created all these elements are initialized according to the specification. The reason to create this context object was due to the high dependency between the tokenizer and tree constructor. With this design it was easier to access these properties by passing the context as an attribute to the parser components. The downside is the larger size of the parser context class.

Additionally, the specification defines a considerable amount of inner algorithms or procedures which may be run from different sources of the parser, specifically from the different insertion modes. To tackle the algorithms were implemented in individual static classes as shown in Figure 3.2. This design allowed us to divide the coding tasks between the team members and decrease the coding process difficulty. Moreover,

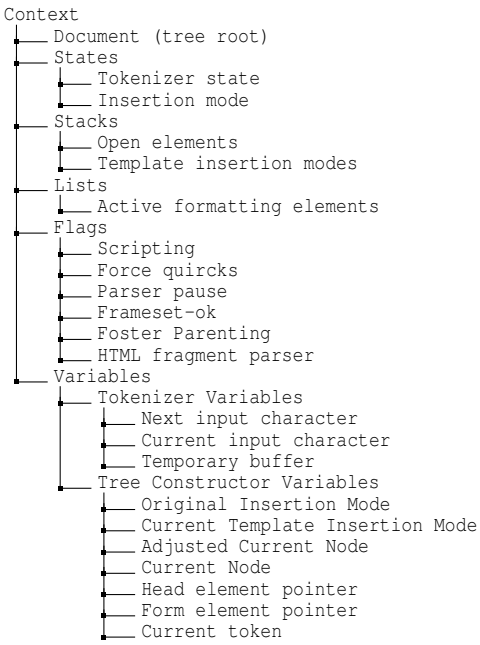


Figure 3.1: Parser Context. Lives within the parser life cycle and contains elements shared between the parser objects.

this design helps to facilitate the algorithm transliteration while keeping high cohesion between parser components.

3.1.2 State design pattern

The adoption of the state pattern design was very helpful in the implementation due to the fact that both tokenizer and tree constructor represents finite state machines. This subsection will show the characteristics of such a design and the justification of why this pattern was selected.

The state pattern design is one of the behavioural patterns for object-oriented software defined by Gamma et al. [29] that allows an object to change its behaviour depending on its state.

The Figure 3.3 shows an abstract class diagram for the state design pattern. The context class is the interface that a client call. This context has an abstract state representing the current state that is implemented by concrete states. Such states define different behaviours and transitions depending of the input data. In addition, the current state could be changed by the context or a particular state.

The benefits of using this design pattern are:

1. All behaviour of a state, in this case the steps described for each state, are put in

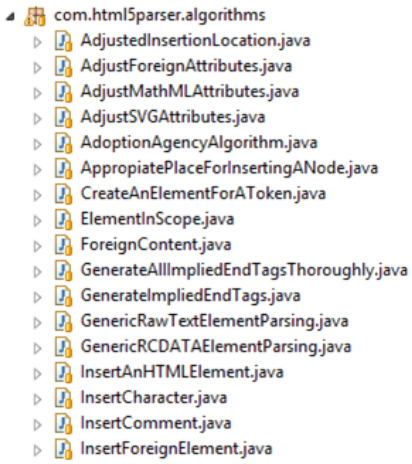


Figure 3.2: Java classes representing different algorithms

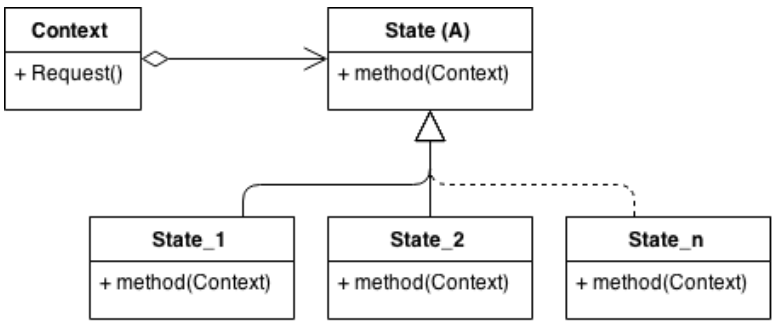


Figure 3.3: State design pattern class diagram

a particular class, increasing the low coupling and high cohesion between states.

2. The states are easy to maintain and easy to add more without changing the context or any other class.

3. A large amount of conditional statements is represented by few lines in the context.

3.1.3 Tokenizer

The tokenizer is the stage that receives characters and generate tokens. This is defined in the specification as a state machine.

Ideally the tokenizer can be modelled as:

$(\Sigma, S, s_0, \delta, F)$

Where:

Σ is the set of characters

S is the set of 68 tokenizer states

s_0 is the initial tokenizer state: Data state

δ is the transition function $\delta : S\,x\,\Sigma \rightarrow S$.

F is the set of final states {Data state, RCDATA state, RAWTEXT state, Script data state, PLAINTEXT state}

The set of characters are all those possible values from the Unicode standard database [17].

The tokenizer states are the 68 states defined in the specification. The specification is confusing in the sense that the last subsection *Tokenizing character references* is stated as a state though after analyzing every state, there is no transition towards it, therefore it was considered as an algorithm that attempts to return character tokens from a numeric or named reference.

The transitions are defined in each tokenizer state, which receives a character and as result it may create and/or emit a token, may switch the state and may update an element from the context. However, one limitation in the design was that the tokenizer can only read and process one character at the time, making it difficult to implement some states that look ahead several characters before doing a transition.

The set of final states are basically those that emit an end of file token. An end of file token may be emitted as a consequence of consuming the EOF character. The EOF character is a conceptual character that represents the end of the stream or as the specification defines, a lack of any further characters. In our design has a value -1.

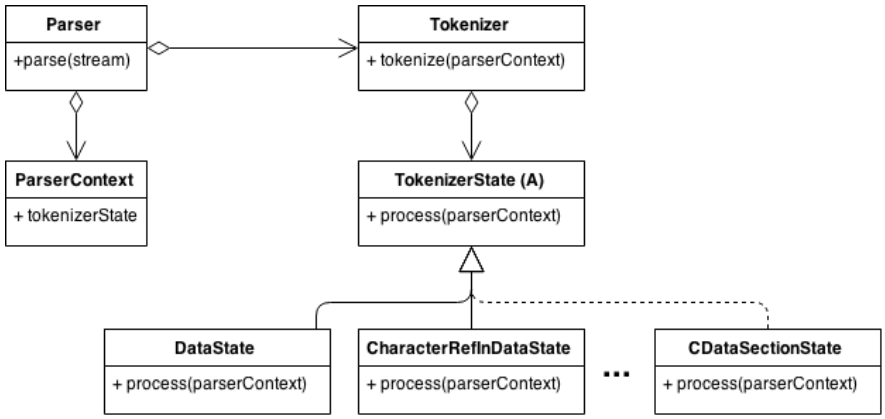


Figure 3.4: Class diagram of tokenizer and related classes

The Figure 3.4 shows the design of the tokenizer. In order to tokenize a stream, the class *Tokenizer* calls the method process of the current *TokenizerState*. This state is retrieved from the *ParserContext*. As we can see, the TokenizerState is able to access and update any public element from the parser Context. That is, can read the next input character from the stream, do the transition to another state or remain in the same and access to other variable if indicated in the specification.

3.1.4 Tree constructor

When tokens are emitted by the tokenizer, the tree constructor immediately process them. Each token goes through an algorithm called tree construction dispatcher that basically push the token to the below state machine if it is HTML content; otherwise it is considered as a foreign content and a different list of steps must be followed described in the *rules for parsing tokens in foreign content* section.

The design of the tree constructor is very similar to the tokenizer with the difference that the input are the tokens generated by the tokenizer and the output is the updated DOM tree. Additionally the machine output, i.e. the DOM tree, depends on the *stack of open elements* state, the *list of active formatting elements* and the *stack of template insertion modes*. The former also affects the behaviour, i.e. the transitions.

Although it is not entirely correct due to it excludes the data structures mentioned before, for the sake of comprehensibility, the tree constructor may be modelled as:

$(\Sigma, S, s_0, \delta, F)$

Where:

Σ is the set of tokens described in the previous section.

S is the set of 23 insertion modes

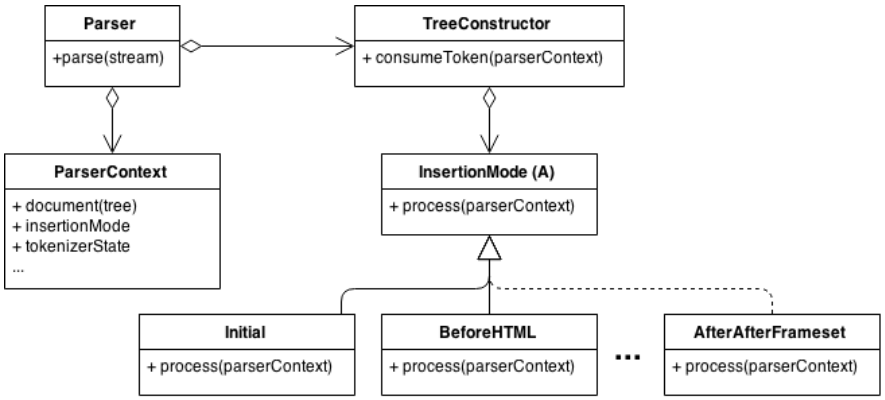


Figure 3.5: Class diagram of TreeConstructor and related classes

s_0 is the initial insertion mode: Initial

δ is the transition function $\delta : S\,x\,\Sigma \rightarrow S$.

F is the set of final states {In Body, In Template, After Body, In Frameset, After Frameset, After after body, After after frameset }

The transitions are defined within each insertion mode which receives a token and as a result it may update the DOM tree, may switch the insertion mode and may update an element from the context (even the tokenizer state). Some insertion modes, depending on the token being processed, run steps from other insertions. Finally most of the algorithms are extensively reused across the insertion modes.

The set of final states are those that indicates to run the stop parsing procedure. After this, the parser returns the DOM tree generated from the parser context object.

The Figure 3.5 shows the design of the tree constructor. The class *TreeConstructor* provides a method *consumeToken* which calls the method process of the current *InsertionMode*. This state is retrieved from the *ParserContext*. Similar to the tokenizer design, the *TreeConstructor* has access to the elements of the *ParserContext*.

The capability to fix mis-nested content is an important feature of the tree constructor. The *adoption agency algorithm*, is an algorithm that is executed by the tree constructor when some conditions are met. Such algorithm can change the DOM tree structure by moving a node to a different parent. For instance in Figure 3.6, when after parsing the character '2', the current tree has two children in body node, being character '2' child of node 'p'. However when the end tag 'b' is parsed, the *adoption agency algorithm* is run and the character '2' is detached from the node 'p' and moved into the node 'b'. Finally 'b' is appended as child of 'p'. This was the main limitation in the Minamide and Mori[4] attempt to generate a conditional push down system.

Input	Tree after parsing '2'	Output
1 23	<html> <head /> <body> 1 2 </body> </html>	<html> <head /> <body> 1 23 </body> </html>

Figure 3.6: A misnested tag example

Input	Output
<h1>This contains errors	<html> <head /> <body> This contains errors </body> </html> Parser errors: Unexpected token: end_of_file value: null at com.html5parser.insertionModes.InBody@5d624da6

Figure 3.7: Parsing example showing a parse error. The closing tag h1 is missing, instead an EOF was encountered.

3.1.5 Error handling

The specification indicates when a parser error is raised by just stating the sentence *This is a parse error*. The specification then describes how to handle the error, however does not define a name or type of error. It is up to the developers to decide the message error. In this implementation it was decided to add them into a stack of parser errors inside the context object, thereby they can be displayed in the log as shown in Figure 3.7 . Due to this, the parser context object is almost in every class of our application.

3.2 Building

Java was the programming language chosen for developing the parser. As the main requirement of the implementation was to learn and understand the specification, and secondly, to be compliant with the specification, there was no restriction concerning which language to be used. The following are the reasons why Java was chosen:

1. High level of experience of team members.

2. Portability, i.e. the binary can be run on any device that has installed a Java Virtual Machine.

3. Object-oriented paradigm. It is possible to take advantage of its encapsulation, polymorphism, and inheritance features. This allows the application of the state pattern previously described.

4. Garbage collection feature; this helps to avoid memory leaks.

5. It is supported by a large community.

No external libraries were used apart from the native Java libraries. Github was used as repository for the code. The source code can be accessed through the following URL: `https://github.com/HTML5MSc/HTML5Parser`.

The development activities were divided between the team following Agile techniques such as working in sprints, pair programming, sprint and product backlog, stand up meetings and sprint retrospectives. Trello[1] was a helpful tool to organize and keep track of the progress.

3.3 Testing

Prior to the development of the test framework, the parser implementation was testes with the HTML5Lib suite, in order to fix errors and bugs. The HTML5Lib test suite can be retrieved from their Git repository as well as detailed information about the test formats[3]. JUnit[2], a Java framework to write tests, was used to develop a test runner to retrieve the files from the repository and run the tests. What follows is a brief description of the test formats.

3.3.1 Tokenizer

The format of the tokenizer tests use a JavaScript Object Notation (JSON) notation. Such format is shown in the Figure 3.1.

[1] https://trello.com/
[2] http://junit.org/

```
{"tests": [
    {"description": "Test description",
    "input": "input_string",
    "output": [expected_output_tokens],
    "initialStates": [initial_states],
    "lastStartTag": last_start_tag ,
    "ignoreErrorOrder": ignore_error_order
    }
]}
```

Listing 3.1: Tokenizer test format using JSON.

The *description* field is only a description of the test. The *input* is the string to be parsed. The test may contain a list of *initialStates*, meaning the test has to be executed with each of these states. The *lastStartTag* means the tag name of the last start tag to have been emitted from the tokenizer. The *ignoreErrorOrder* is a Boolean that indicates whether the order of errors are validated. The *output* is a list of expected tokens. The format of valid tokens are shown in Listing 3.2. When a *StartTag* has a third value *true*, means that is a self-closing tag. The string *ParseError* indicates that a parse error was raised. Our test runner just compares the number of these incidences and not the value or reason of error. The disadvantage is that there is not possible to validate the location or reason of error, only the number of them.

```
["DOCTYPE", name, public_id, system_id, correctness]
["StartTag", name, {attributes}*, true*]
["StartTag", name, {attributes}]
["EndTag", name]
["Comment", data]
["Character", data]
"ParseError"
```

Listing 3.2: Expected tokens test format.

3.3.2 Tree builder

The HTML5Lib tree construction tests are more like testing all the parser than only the tree constructor. That is, the input is a string instead of a list of tokens. Therefore these tests depend on the correct functionality of the tokenizer. The format consists of any number of tests separated by two lines and a single line before the end of the file. The test may contain several options, however, the minimum are:

- #data: The input string
- #errors: Expected errors

- #document: The expected DOM tree

The #document is the DOM tree serializing each node in one line. Each line must start with the character pipe ('|') followed by two spaces per level of depth. The Listing 3.3 shows the test for the example of misnested tags seen in Figure 3.6. A detailed specification can be found in the HTML5lib tests online respository[3].

```
#data
<b>1<p>2</b>3</p>
#errors
Error
Error
#document
| <html>
|   <head>
|   <body>
|     <b>
|       "1"
|     <p>
|       <b>
|         "2"
|       "3"
```

Listing 3.3: Tree Constructor test example.

3.4 Summary

This chapter has described the methods used to develop a HTML5 parser by following the specification parsing algorithm. The state design pattern was used to implement the Tokenizer and Tree Contructore components of the parser. The language selected was Java because of its features and the previous experience the team has. The HTML5Lib test suite along with JUnit were used to evaluate the parser and fix outstanding errors. Moreover, challenges and implications encountered throughout the implementation were also discussed. Similarly, the next chapter describes the procedures and methods used for the test framework implementation.

[3] https://github.com/html5lib/html5lib-tests/tree/master/tree-construction

Chapter 4

Test Framework

This chapter moves on to the design and implementation of the test framework. Based on N-version diversity systems, a test framework was build to measure the level of convergence among parsers and thus have an approximation of their level of compliance with the specification. The first section describes the design of the framework, its architecture, parser adapters and the Comparator and plurality algorithm responsible to generate the results. The second section describes how these components were built. Finally, a brief description of other components of the framework is given.

4.1 Design

4.1.1 Architecture

Our approach was influenced by the N-version diversity, known for being used in fault-tolerant systems, similar to Manolache and Kourie [5] M-mp testing strategy. The design described in this section compares the results of independent developed HTML5 parsers. Then the plurality agreement is selected as the correct result.

Below are described the advantages and disadvantages of this method.

Advantages

- Can take any web page or any piece of HTML as test input. Therefore, using existing HTML from the Web saves the effort to create test cases.
- Can find disagreements among parsers that lead to a parser bug or even a specification bug.

- A disagreement may represent a new test case to add to the test suite after analysing the test.

Disadvantages

- A successful result does not mean that it fully conforms to the specification. Moreover it might be a bug in all winner implementations. This is a *common-mode failure*, previously defined in Chapter 2.

- A disagreement still has to be manually analysed to know what is the reason and probable error.

A diagram of the whole architecture of the framework is shown in Figure 4.1. An overview to its components is given below.

The **input** can be an HTML document as a string, from a URL, a file or a set of documents, e.g. from a Web ARChive (WARC) file. A WARC file is a format to archive HTML responses. A more detailed description is given in Chapter 6

The **input preprocessing** prepares the input for the parsers. For example, if the input is a test of cases, they must follow a certain file structure in order to be processed by the parsers.

Then the input is parsed by a set of **parsers** that are part of the test. As every parser may serialise the output tree differently, a custom **adapter** must be used to serialise the output with a common format, hence the outputs can be compared.

The next step is the execution of the **comparator** which compares the tree outputs and finds the most likely correct output by using a plurality voting algorithm. The output is a report file in XML and a set of files containing the disagreements.

Finally, a **Web application UI** presents the report and disagreements in a human readable interface. Additionally, a **tracer** can be enabled in *MScParser* implementation to show which parts of the specification algorithm were executed by running a particular test.

4.1.2 Adapters

As was pointed out in the previous subsection, an adapter must be created in order to include a parser in the framework. Every adapter must serialise the same output format to be comparable. In order to have an adapter working in the framework, it must comply with the following specification.

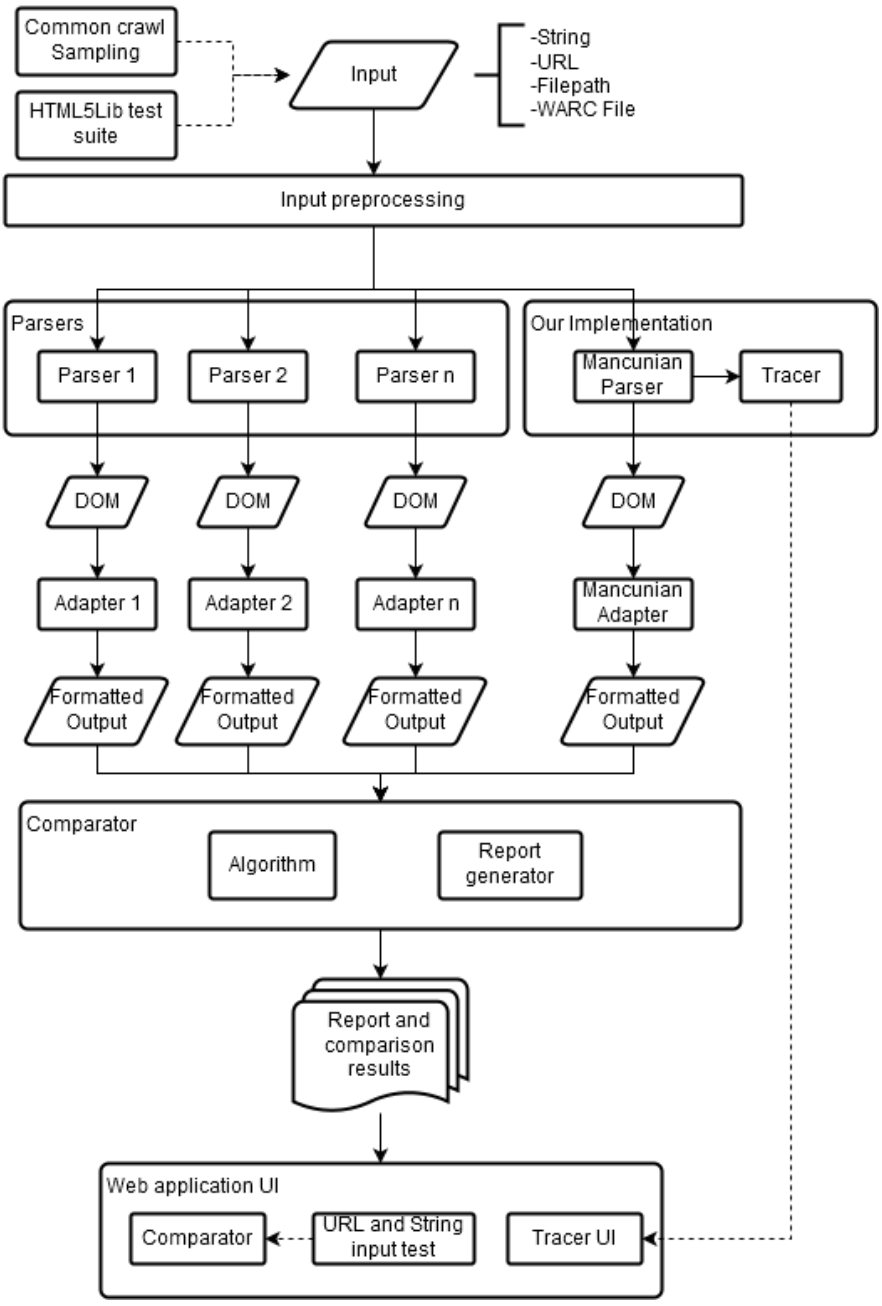


Figure 4.1: Test framework architecture. Designed along with Anaya[1]

The adapter must provide three services, parse a string, parse a file and parse a document from an URL. It is recommended to follow the next execution syntax:

1. To parse a string: -s [string]
2. To parse a file: -f [path to file]
3. To parse a document from an URL: -u [url]

Additionally, the adapter must encode and decode in UTF-8 due to the project scope. It must serialise according to the HTML5lib test suite format, as described in section 3.3.2. And the *scripting flag*[1] must be enabled. These guidelines must be followed to ensure the same output across parser adapters. It was decided to delimit the scope by only testing the behaviour with the *scripting flag* set as true. The reason was that several parsers, e.g. Parse 5, HTML5Lib and Jsoup, did not implement this behaviour.

It was decided that the best format to serialise the output trees was the HTML5Lib format. This format is widely used and it is simple, reliable and language independent. However, a limitation is that it cannot be parsed again as HTML. This limitation could be solved by serialising again to HTML, although this approach was rejected because it was considered that it would require greater effort for the designing and implementation.

4.1.3 Comparator and plurality agreement

The comparator is the component responsible for comparing the outputs trees generated by each parser from a single HTML input and for selecting the most likely correct output based on plurality voting. A plurality is the element that have more votes, or in this case, more agreements. Thus majority differs from plurality, by the fact the element must have more than the half of total votes[30]. Therefore a plurality may not be unique. Although if this case is presented then, in this design, is considered no plurality.

A plurality voting was chosen over majority voting because of the following scenario (third scenario in Table 4.1). Assuming we have two compliant parsers that produce the same output and three not compliant parsers that produce unique different outputs, then there is a plurality but not a majority. Therefore a plurality voting results

[1] `http://www.w3.org/TR/html5/syntax.html#scripting-flag`

Scenario	Majority	Plurality
All trees are equal	Yes	Yes
The largest group of identical trees represents more than the half of total trees	Yes	Yes
The largest group of identical trees does not represent more than the half of total trees	No	Yes
There is not a unique largest group of identical trees	No	No
All trees are different	No	No

Table 4.1: Majority VS Plurality. Possible scenarios when comparing output trees

in two correct parsers whereas in majority zero correct parsers, even if one agreement was given.

4.2 Building

The building consisted in the following:

1. **Adapters**. Each adapter had to be developed in the same language as each parsers.

2. **Parser Runner**. The adapters could be run manually, however an small bash script was developed to run them all with the same input. The output are the trees generated from the parsers organized as described in the following Subsection 4.2.2.

3. **Comparator**. This program compares and creates a report with the results in a XML file. We started developing the comparator in bash, however several limitations were encountered. First, the team did not have experience with this language thus productivity was slow. Second, the decision to use a XML file to store the results led to use Java, because the team had a better knowledge of the Java API for XML Processing (JAXP) library.

Same as with the parser implementation, Github was used as repository. Source code can be found in `https://github.com/HTML5MSc/HTML5ParserComparator`. Maven[2] can be used to build the application and retrieve all dependencies.

What follows is a description of the above software components.

[2]https://maven.apache.org/

4.2.1 Parser adapters implementations

Every adapter must produce the HTML5lib test format output as described in Subsection 3.3.2 in order to compare them and find the disagreements. The tree format is described in detail in the HTML5lib test suite on-line repository[3].

The development of an adapter can be quite challenging. It depends on the language and implementer. For example, some implementations had the code already for building the tree with the required output format, a reason why this format was chosen. However with other parsers, including ours, it was necessary to code it. Moreover, even if the implementation had already a serializer, i.e. html5lib and parse5, some fixes were done in order to obtain the desired output.

As an example, a piece of code of the HTML5lib adapter is shown in Listing 4.1. The testSerializer (line 12) was reused to give the HTML5lib test format, however, a *convertTreeDump* method was developed to fix some extra spaces generated. Observe the use of UTF-8 encoding.

```
1  p = html5parser.HTMLParser(tree=html5lib.getTreeBuilder("dom"), namespaceHTMLElements
       =False)
2
3  if len(sys.argv) == 3:
4  if sys.argv[1] == '-f':
5          with open(sys.argv[2], "rb") as f:
6                  document = p.parse(f,encoding="utf-8")
7  if sys.argv[1] == '-s':
8          document = p.parse(sys.argv[2])
9  if sys.argv[1] == '-u':
10         with closing(urlopen(sys.argv[2])) as f:
11                 document = p.parse(f)
12 output = p.tree.testSerializer(document)
13
14 print convertTreeDump(output.encode('utf-8'))
```

Listing 4.1: Html5Lib adapter piece of code showing how the parser must be created in order to obtain the output format required for the test framework.

4.2.2 Preparing the input

Single test

In order to compare one test, the files containing the output trees of the parsers in testing must be placed inside a folder with the name of the test. Each of these files

[3] https://github.com/html5lib/html5lib-tests/tree/master/tree-construction

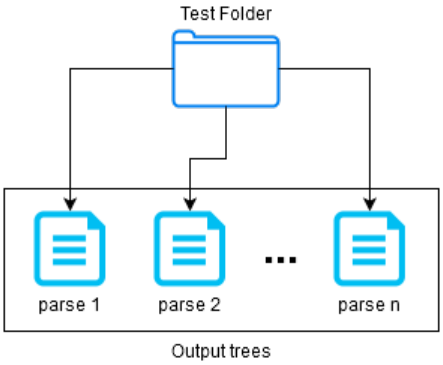


Figure 4.2: Single test. A folder contains the output trees from different parsers.

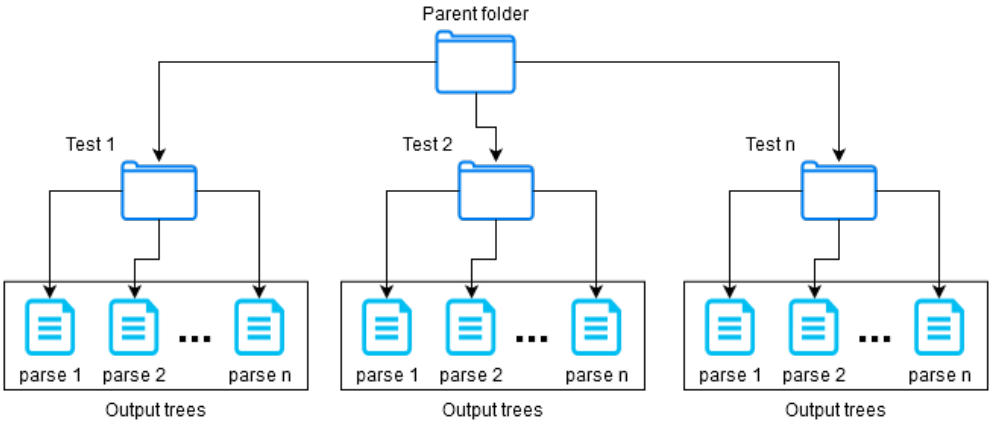


Figure 4.3: Multiple test. A folder contains sub folders that represent the tests.

should have as name the name of the parser that generates it. The Figure 4.2 illustrates the above.

Multiple tests

In order to compare a set of tests, they must follow the single test file structure and be inside a parent directory. The Figure 4.3 illustrates this file organization. The output of the parser runner follows this structure.

4.2.3 Comparator

A Java application was developed to implement the comparator. This comparator is able to generate a report from a single test or a set of tests.

The first step was to compare and group the parsers with identical trees, i.e. agreements. Then the parsers that are plurality are selected according to the plurality voting

```
1 +67382,24,"
2 |                      ;+67408,25,>
3 |                       ;-67472,1,>;+67473,26,
4 |                        ";-67526,4,</a>;+113993,1,";+113995,7,|     ";|
```

Figure 4.4: Disagreement file example. This file store the differences against the most likely correct tree.

algorithm. Finally the disagreements and the plurality result are added into a report XML file which is next described.

Report

The XML technology let us organize tests and the results of the parsers in a tree structure. This format will also let any language with XML capabilities to read, query and/or process the report. The Java library Java API for XML Processing (JAXP) was used to create and process the report file.

The first approach was to add the parser outputs and result of the comparator in the report. However, a problem was found when parsing a large amount of data. The JAXP library have a memory size limit for storing the DOM tree. The comparator stopped working when reaching this limit, because of the big size of the report file. This led us to store the parser outputs outside the XML file, in external binary files. Thus, the XML file provides the paths to these files.

Moreover, in order to reduce even more the disk usage and avoid identical files because of identical parser outputs, it was decided to store only the most likely correct input and the differences with the other outputs. The figure 4.4 shows an example. Concatenated blocks indicate the location of the difference, the length of difference and the difference. The upside is an increase in computability whereas the downside is that a processor for this type of file has to be developed in order to show the differences in a human readable output.

4.3 Other framework features

4.3.1 Web Interface

A web application user interface (UI) was developed to visualize the contents of a report in a more human readable way. Furthermore it simplifies the navigation through the disagreements and provides different utilities to improve the user experience and

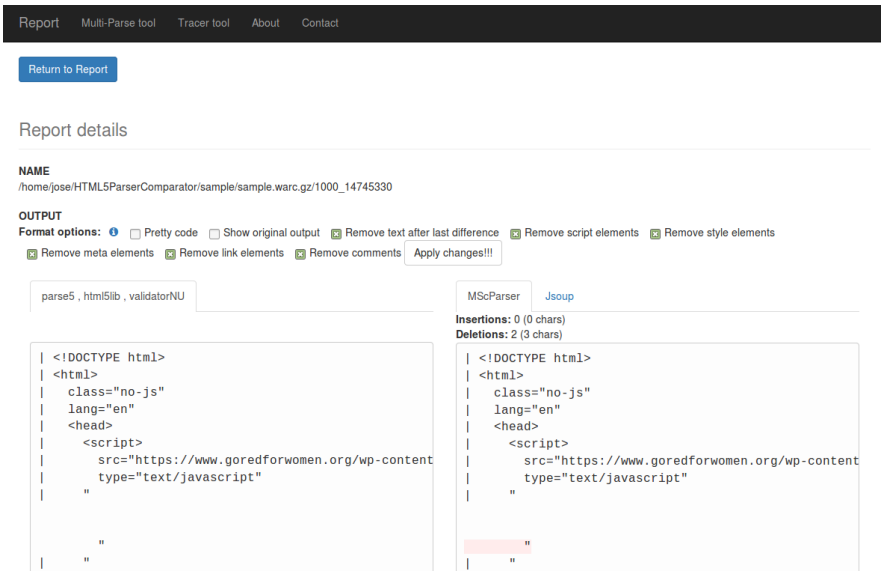


Figure 4.5: Web application UI screen shot. This image shows two output trees and a difference in red color.

analysis of the results. The Figure 4.5 illustrates a screen shot of the web application showing a difference between two trees.

4.3.2 Tracer

A tracer was developed in by Anaya using the parser implemented in this study described in section 3. This tracer provides a log of the different sections touched by parsing an HTML document and can be used to find and trace bugs in a parser. More information can be found in Anaya's work [1].

4.4 Summary

The test framework developed is based on N-version diversity systems and consists of a input preprocessing, parsers to be evaluated along with their custom adapter, the Comparator, the Web Application and Tracer.

This chapter described the design and implementation of these components. Parsers adapters must follow a defined specification in order to generate the outputs trees with the HTML5Lib format. Then, the Comparator compares these output trees and decides which parsers are most likely to have the correct output by following a plurality algorithm. Moreover, it was explained how single and multiple tests can be done, and how

they have to be organised in the file system.

In order to execute the test framework, a group of parsers and a set of input test cases have to be selected. The next chapter shows a survey of HTML5 parsers and lists the parsers selected whereas Chapter 6 describes the method for sampling the Common Crawl data to obtain a set of test cases.

Chapter 5

HTML5 parsers survey

The purpose of this chapter is to present a survey of HTML5 parsers. This research helped to choose which parsers could be part of the test framework. For this study, the parsers that used and passed the HTML5Lib test suite had more priority because a disagreement would represent a missing test for the test suite or a potential bug in the specification.

The most known implementations, and probably the most important, are those used by the HTML browsers. The parser is part of their layout engines. For example Chrome uses Blink[1], Safari use Webkit[2], Internet Explorer uses Trident[3] and Firefox uses Gecko [4]. Additionally, Mozilla is currently working on a new engine called Servo. While all rendering engines above were coded in C++, Servo is being coded in Rust language looking to exploit its parallelism power and increase performance considerably [31].

In addition to be used in browsers, an HTML5 parser can also be used as a:

- Validator. i.e validating documents against the HTML specification.
- Checker. i.e fixing documents in order to conform to the HTML specification.
- Sanitizer. i.e. removing some data from the HTML document to avoid cross-site scripting attacks.
- Minifier. i.e. reducing the size of a document while preserving the same functionality.

[1] http://www.chromium.org/blink
[2] https://www.webkit.org/blog/1273/the-html5-parsing-algorithm/
[3] https://msdn.microsoft.com/en-us/library/aa741312%28v=vs.85%29.aspx
[4] https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/HTML5_Parser

Parser	Language	Html5lib tests[1]	Compliant with	Last release version	Last Update
AngleSharp	C#	Yes	W3C[8]	0.9.0	26/08/2015
parse5	javascript	Yes	WHATWG	1.5.0	24/06/2015
Html5lib	python	Yes	WHATWG	1.0b7	07/07/2015
jsoup	Java	No	WHATWG	1.8.3	02/08/2015
validator.nu	Java	Yes[2]	WHATWG	1.4	29/05/2015[3]
cl-html5-parser[4]	Common Lisp	Yes	WHATWG	Unknown	17/07/2014
html5ever[5]	Rust	Yes[6]	WHATWG	Not released yet	31/08/2015[7]
tagsoup	Haskell	No	No specified	0.13.3	01/10/2014
html [3]	Dart	Yes	WHATWG	0.12.1+2	06/07/2015
gumbo	C99	Yes	WHATWG	0.10.1	30/04/2015
Hubbub	C	Yes	WHATWG	0.3.1	08/03/2015
html5-php	Php	No	W3C	2.1.2	07/06/2015
html	Go	Yes	WHATWG	Unknown	28/07/2015
HTML::HTML5::Parser	perl	Yes	No specified	0.301	08/07/2013

[1] The value "No" means that there is no information provided nor tests included in the source code that indicates the use of HTML5lib test suite.
[2] Assumed from the acknowledgements section of their page.
[3] This is the date of the last update of the source code, since the binay file have not updated. The binary file last update in maven is 05/06/2012.
[4] Html5lib port.
[5] Currently under development.
[6] Currently passes all tokenizer tests and most of tree builder.
[7] This is the date of the last update of the source code, since this parser is under development.
[8] W3C compliant with some WHATWG extensions.

Table 5.1: HTML5 parser survey

- Scraper. i.e. extracting relevant data.

Table 5.1 shows an overview of a list of some HTML5 parsers that can be found on the web. The search was done mainly using the Google search engine and in GitHub repositories. Some of their websites had links to another parsers. Those parsers that explicitly state that they parse HTML5 were included here. It can be seen that mostly all implementations of the WHATWG specification but jsoup use the HTML5lib test suite. On the other hand W3C implementations use their own test suite. A more detailed table can be found in Appendix C.

Additionally to the parsers presented in the previous table, the following list shows those parsers that are port of gumbo parser[32].

- C++: gumbo-query by lazytiger
- Ruby:
  - ruby-gumbo by Nicolas Martyanoff
  - nokogumbo by Sam Ruby
- Node.js: node-gumbo-parser by Karl Westin

- D: gumbo-d by Christopher Bertels
- Lua: lua-gumbo by Craig Barnes
- Objective-C:
  - ObjectiveGumbo by Programming Thomas
  - OCGumbo by TracyYih
- C#: GumboBindings by Vladimir Zotov
- PHP: GumboPHP by Paul Preece
- Perl: HTML::Gumbo by Ruslan Zakirov
- Julia: Gumbo.jl by James Porter
- C/Libxml: gumbo-libxml by Jonathan Tang

The parsers selected for this experiment are following. Parse5, validator.nu and html5lib were chosen because they claim to pass all HTML5lib[33][34][3]. Furthermore, parse5 developers has a faster performance according to their developers[33]. The benefit comes when parsing a large set of HTML documents as input test, reducing the time of the experiment. On the other hand, validator.nu was also chosen because a port in C++ is used in Gecko engine[34]. The benefit is that we could have a similar behaviour of Mozilla Firefox, one of the major browsers. Jsoup was chosen because it appears to be widely used in the Java community[35]. Anglesharp was chosen by one of the members of the team because he has experience with the C# language. Finally our implementation was also included as a evaluation and also to prove how compliant is a new parser that passes all HTML5lib test suite.

1. parse5
2. validator.nu
3. html5lib
4. jsoup
5. AngleSharp
6. MScParser

Although it would be very interesting to add the browsers parsers, the reason why they were not included was because they are not standalone. Therefore, it represents a considerable effort to extract the parser from the browser engine. According to the dissertation plan, it was considered not feasible to include such an activity.

5.1 Summary

A survey of HTML5 parsers with relevant information to this study was presented in this section. The parsers selected to be in the execution of the test framework are *parse5*, *validatorNU*, *html5lib*, *jsoup*, *AngleSharp* and the parser developed in this project *MScParser*.

Chapter 6

Common crawl data set as source of test cases

Different sources of HTML documents were considered as test cases. First, manually created tests. This method would be similar to the HTML5Lib test suite, therefore instead we used this suite to avoid duplicating work already done. Second, automatically created random HTML documents. After doing an small HTML5 random generator it was found that it is not a trivial task. It requires a very good knowledge of all elements of the specification in order to have the maximum coverage. Moreover, a big effort would be required to create a robust algorithm capable to generate useful HTML code, instead of generating a bunch of useless random text. Furthermore, Minamide and Mori[4] work has shown the difficulty to produce a grammar for the generation of random test cases. Third, use the web as a practically unlimited source of real test cases. Although the ideal testing would be a full coverage of the web, such process would require an enormous amount of resources. A simpler solution would be testing the free to use Common crawl data set. Common crawl have been storing a very large amount of web resources from the web. Even processing the whole Common crawl data set represents a difficult challenge. Therefore it was decided to obtain a uniform random sample representative of the Common crawl corpus.

On the other hand, another method analysed was the use of a web sampling technique. Baykan et al.[36] investigated and compared different web sampling techniques based on random walks. This method may provide more reliability for the sample. Again, Common crawl approach was chosen over a web sampling technique because it is simpler to obtain random records from the common crawl than implementing one of the algorithms for web sampling.

In addition to the html5lib test suite, a random sample of HTML documents was extracted from the Common crawl data set to be the input set of test cases for the test framework. What follows is a description of the Common crawl data set and the Common crawl index, a useful project for the sampling process.

6.1 Common crawl introduction

Common crawl is a non-profit organization dedicated to providing a copy of the internet at no cost[37]. Common crawl stores HTTP responses obtained with a Nutch-based web crawler that makes use of the Apache Hadoop project. Before blekko[1] was acquired by IBM in March 2015, Common crawl used a list of URL's provided by blekko URL ranking information. Nowadays it appears that Common crawl is using a static list of URL's primarily composed of the blekko sourced data[38]. The data is stored on Amazon Simple Storage Service (S3) as part of the Amazon Public Datasets program and every month a new crawl archive is released.

Amazon S3 uses the following concepts[39]:

1. *Bucket*. Is the container for objects.
2. *Object*. Consist of object data and metadata.
3. *Key*. Unique identifier for an object within a bucket.
4. *Region*. The physically location of servers where buckets and therefore data is stored.

The common crawl dataset is stored in the bucket *aws-publicdatasets* which is in the region *US East (Virginia) region*. An example of a particular key is "common-crawl/crawl-data/CC-MAIN-2015-22/segments/1432207930895.96/warc/ CC-MAIN-20150521113210-00139-ip-10-180-206-219.ec2.internal.warc.gz".

Advantages

1. It stores a very large amount of HTML pages, i.e. June 2015 archive is over 131TB in size and holds more than 1.67 billion webpages[2].

[1] Web search engine similar to Google.
[2] http://blog.commoncrawl.org/2015/07/june-2015-crawl-archive-available/

2. The data is free via Http or S3.

3. There are free frameworks to use, e.g. IIPC's Web Archive Commons library[3] and Amazon Web Services (AWS) SDK[4], that facilitate the access and processing of the data.

4. A complementary project called Common Crawl Index[40], provides an index for the common crawl data set. Useful for the sampling process.

Disadvantages

1. May not be a good representation of the whole web. It has been difficult to find more information about the crawling process, therefore the constitution of the monthly crawl archive is uncertain.

2. It could be that the data set is biased to English web pages[38].

Despite of the weaknesses, Common crawl was chosen as a sample source due to is a public data set free to use and along with IIPC Web Archive Commons and Amazon web services, it provides a powerful and simple work environment for working with large data.

6.2 Common crawl corpus description

The Common crawl corpus consists in three formats which are:

- Web ARChive (WARC) Format. This format is the raw crawl data which includes the HTTP response (WARC-Type: response), how the information was requested (WARC-Type: request) and metadata of the crawl process itself (WARC-Type: metadata).

- WAT format. This includes important metadata about the records stored in the WARC format. i.e. if the response is HTML, the metadata includes the links listed on the page.

- WET format. This includes only extracted plaintext.

[3] https://github.com/iipc/webarchive-commons
[4] https://aws.amazon.com/es/tools/

HTML documents are in the WARC files, specifically in the WARC-Type: response section. Though not all responses return an HTML document, for example the Picture shows an image response 6.1. WARC files are compressed gzip files. Every monthly crawl contains a collection of WARC files, of 1 gigabyte average. Each WARC file is a concatenation of several records.

```
WARC/0.16
WARC-Type: response
WARC-Target-URI: http://www.archive.org/images/logoc.jpg
WARC-Date: 2006-09-19T17:20:24Z
WARC-Block-Digest: sha1:UZY6ND6CCHXETFVJD2MSS7ZENMWF7KQ2
WARC-Payload-Digest: sha1:CCHXETFVJD2MUZY6ND6SS7ZENMWF7KQ2
WARC-IP-Address: 207.241.233.58
WARC-Record-ID: <urn:uuid:92283950-ef2f-4d72-b224-f54c6ec90bb0>
Content-Type: application/http;msgtype=response
WARC-Identified-Payload-Type: image/jpeg
Content-Length: 1902

HTTP/1.1 200 OK
Date: Tue, 19 Sep 2006 17:18:40 GMT
Server: Apache/2.0.54 (Ubuntu)
Last-Modified: Mon, 16 Jun 2003 22:28:51 GMT
ETag: "3e45-67e-2ed02ec0"
Accept-Ranges: bytes
Content-Length: 1662
Connection: close
Content-Type: image/jpeg

[image/jpeg binary data here]
```

Listing 6.1: WARC response example

6.3 Common crawl Index

As was pointed out earlier, the Common crawl Index[40] is useful for the sampling process. The common crawl index provides the location of WARC records of the Common Crawl data set, mostly used for querying URL's. The common crawl Index is generated with each monthly common crawl archive(collection). Moreover is also stored on S3, in the same bucket as the Common crawl data set, and is free to use, though currently an Amazon Web Services (AWS)[5] account is required to access it.

The common crawl Index format is called ZipNum Sharded[41]. Such a format divides the complete index into several blocks called shards of commonly 3000 CDX records. These shards are spread over multiple compressed files (part-file). For each of

[5]https://aws.amazon.com/

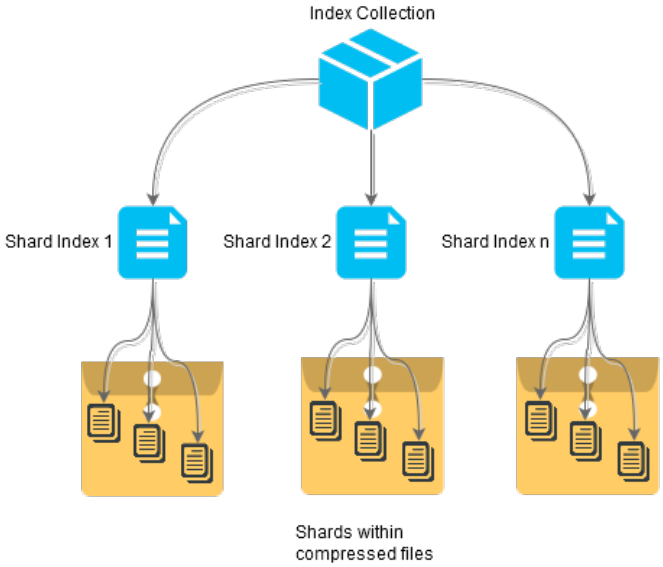


Figure 6.1: Common crawl index structure. The index is constituted by several second index files that contains the location of the compressed shards

```
Shard Record
├── URL search key
├── URL timestamp
├── Part-file index
├── Offset where the shard begins
│   └── Length of shard
com,aol,discover)/aoldesktop97 20150529135708 cdx-00015.gz 221669808 211634
```

Figure 6.2: Format and example of a Shard index record. Extracted from the Common crawl index.

these part-files exist a shard index that contains necessary data to locate every shard. The picture 6.1 shows the structure of the ZipNum Sharded CDX format.

Every line in the ”second index” represents a shard. The Figure 6.2 shows the format and an example.

A CDX record represents a WARC record of the Common crawl data. It contains the necessary information, i.e. container file, offset and record length, to retrieve the record. The Figure 6.3 shows the format and an example.

Moreover, as can be seen in the examples shown, the URL field is transformed to make it easier for lexicographic searching. Common crawl index reverses sub-domains,e.g. example.com to com,example,)/ to allow for searching by domain, then sub-domains[40].

```
CDX Record
├── URL search key
├── URL timestamp
└── JSON format
    ├── Original url
    ├── MIME
    ├── Response status
    ├── Digest
    ├── Length in container WARC
    ├── Offset where the record begins in container WARC
    └── Filename/key of the container WARC
```

```
com,jqueryui,bugs)/ticket/5963 20150529000543 {"url": "http://bugs.jqueryui.com/
    ticket/5963", "mime": "text/html", "status": "200", "digest": "
    FG5BIQ4YF4PY6734OK6I4DP4LMK6YRTM", "length": "4532", "offset": "32425753", "
    filename": "common-crawl/crawl-data/CC-MAIN-2015-22/segments/1432207929803.61/
    warc/CC-MAIN-20150521113209-00328-ip-10-180-206-219.ec2.internal.warc.gz"}
```

Figure 6.3: Format and example of a Common crawl CDX index record. Extracted from the Common crawl index.

6.4 Random sample algorithm

The Common Crawl Index was used to extract a number of random. Once the sample of indexes is created, then it is used to retrieve the Common crawl records.

The Listing 6.2 shows how the index sample is built from a Common crawl Index collection. Starts by randomly distributing the numbers of records to be retrieved per shard index file. Then for every file, the similar procedure is done assigning how many records will be retrieved per shard. For example if an index file was assigned to retrieve 10 records, these 10 records will be randomly distributed to its shards, and then the records will be taken randomly from the shards. The outcome of the algorithm is a plain text file with the random index records.

```
1  SET i=0
2  INIT all shardIndexFile.recordsToRetrieve = 0 //shard index files number of records
       to retrieve to zero
3  WHILE i <= sampleSize
4          SET shardIndexFile = pick a random shard Index file from the Index Collection
5          INCREMENT shardIndexFile.recordsToRetrieve by one
6          INCREMENT i by one
7  ENDWHILE
8  FOR each shardIndexFile in indexCollection
9          SET i=0
10         READ shards from shardIndexFile
11         INIT all shard.recordsToRetrieve = 0 in shards
12         WHILE i <= shardIndexFile.recordsToRetrieve
13                 SET shard = pick a random shard from the shard Index file
14                 INCREMENT shard.recordsToRetrieve by one
15                 INCREMENT i by one
16         ENDWHILE
17         FOR each shard in shardIndexFile
```

```
18              SET records = READ random records from shard. The number of records to
                    retrieve is the value of shard.recordsToRetrieve.
19              ADD records to sample
20          END FOR
21  END FOR
22  WRITE sample to sample.txt
```

Listing 6.2: Sampling algorithm

6.4.1 Random Shard Index File

The following describes the method for obtaining a random **shard index file** from an Index Collection. Such a method is executed in the line 4 of the Listing 6.2. According to how the Common crawl Index is built, the number of shard index files is variable. Therefore the first step is to count the number of shard index files of the collection and then get a random number between 1 and the total number of files. In the code shown in Listing 6.3 the structure *shardFiles* was previously filled with the paths of the shard files. The number *random* is a pseudo random generated number between the values of *min* and *max* that is used as a random index for the array structure *shardFiles*. The return object is the Path of the shard index file.

```
Path getRandomShardFile(List shardFiles) {
        int min = 0;
        int max = shardFiles.size() - 1;
        int random = min + (int) (Math.random() * (max - min +1));
        return shardFiles.get(random);
}
```

Listing 6.3: Random shard file selection

6.4.2 Random Shard

What follows is a description of the method for obtaining a random **shard** from an Index Collection. Such a method is executed in the line 13 of the Listing 6.2. The algorithm is similar to Listing 6.3 though the input is a list of the shard offsets of a Shard index file. This structure is previously filled by reading the Shard index file, as seen in line 10 of Listing 6.2. The offset is used as key for the shards. The offset is later used to read the shard index record (Figure 6.2). The Listing 6.4 shows the Java implementation of the previously described algorithm.

```
Shard getRandomShard(List shardOffsets, Path shardFile) {
        int min = 0;
```

```
            int max = shardOffsets.size() - 1;
            int random = min + (int) (Math.random() * (max - min +1));
            return readShard(shardOffsets.get(random), shardFile);
        }
```

Listing 6.4: Random shard selection

6.4.3 Random CDX Index records

The extraction of the CDX index records is done after selecting randomly the shards and the number of index records to retrieve from each. In order to retrieve a random set of index records from a shard, as seen in line 18 of Listing 6.2, the shard offset and shard length are used to read a shard in the compressed shard file. These values are obtained from the shard index record (Figure 6.2). The Listing 6.5 shows the algorithm to obtain a sample of records from a shard. Observe that the set *randomLines* is filled with a random sample of numbers that represent the lines of the shard. Every line is a CDX index record. And this sample does not repeat records. The Listing 6.6 shows the algorithm based on the Fisher-Yates shuffle algorithm for sampling a range of numbers that may represent a number of record. In addition the records are filtered by its *mime*, which is the content type. This may return less records than expected for the sample.

```
SET randomLines with a random sample from the set of numbers from 1 to 3000
READ shard from IndexFile from Offset to Offset + ShardLength
SET lineCounter = 1
FOR each record in shard
        IF  randomLines contains lineCounter THEN
                IF record.mime == text/html      THEN
                        ADD record to records
                END IF
        END IF
        INCREMENT lineCounter by one
END FOR
```

Listing 6.5: Random CDX records from a shard.

```
// Similar to Fisher-Yates shuffle
        public static Integer[] sampleInteger(int sampleSize, int maxValue) {
                // Create the universe
                int[] array = new int[maxValue];
                for (int i = 0; i < maxValue; i++) {
                        array[i] = i + 1;
                }

                // Random the sampleSize locations of the array
                Random r = new Random();
                for (int i = sampleSize - 1; i >= 0; i--) {
```

```
                int index = r.nextInt(array.length);// random position of all
                    the array
                // swap
                int tmp = array[index];// save the random position value
                array[index] = array[i];// swap the values
                array[i] = tmp;
        }

        // Cut the sample locations only
        Integer[] sample = new Integer[sampleSize];
        for (int i = 0; i < sampleSize; i++) {
                sample[i] = array[i];
        }

        return sample;
    }
```

Listing 6.6: Sample from a range of numbers. Used for sampling CDX records from a Shard.

6.5 Sampling method

The purpose of this section is to describe the method followed to obtain a sample from the Common Crawl data set. Currently, the process only allows to obtain a sample of a monthly collection. The Figure 6.4 shows an overview of this method.

The following details the procedure with reference to the Common crawl data set of May 2015 (CC-MAIN-2015-22), therefore some numbers may differ if the procedure is executed with another collection. Both sampling and WARC building processes were run on a Amazon EC2 instance to improve the speed by taking advantage of the data transfer speed between S3 and EC2 Amazon infrastructures. An Amazon Web Services account is required in order to access to the Common crawl Index and create the sample.

The language used to develop the necessary tools for sampling the Common crawl is Java. The reason is that IIPC provides a library called Web Archive Commons for processing compressed WARC files. The AWS SDK[6] was also used to access, read and retrieve data from amazon S3.

The first step was to download the set of shard index files from the Common Crawl Index. There are 300 files with a total size of 81.3 MB. In which are distributed 671970 shards with approximately 3000 records each. Thus this collection of Common Crawl

[6] https://aws.amazon.com/es/tools/

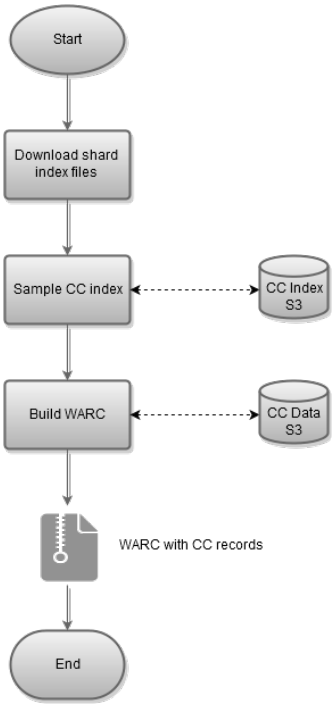


Figure 6.4: Sampling method. The sample of indexes is created first. Then the WARC file is built.

Index and therefore Common crawl data set have approximately 2,91 billion records.

The second step is to **sample the Common crawl index** as described in section 6.3. A Java tool was developed to generate a CDX file containing the sample. This software receives as input data the sample size, the location of the shard index files previously downloaded and the S3 key of the compressed shards. Additionally it requires a file *profileConfigFile.properties* which contains the AWS user account credentials for accessing to the Common crawl Index. The Figure 6.5 shows part of the sample file.

The third step is the **building of the WARC file**. As part of the Java tool, a service was developed to build a compressed WARC file. The input is the previously generated index CDX file and the file name of the output WARC. Each CDX record of the CDX file contains the key, offset and content length, as seen in Figure 6.3, data necessary to retrieve the record from the Common crawl data set. All records from the sample are retrieved and written in a new WARC file.

It was decided to separate the generation of the sample and the WARC building to let modify the index sample according to what is required. For example, to filter even more the data by top level domain or domain.

A Java application was developed to provide the above services:

```
com,asianscientist)/2011/10/features/ric-obarry-the-cove-acres-resorts-world-sentosa-free-captured-dolphins-102011 20150528220239 {"url": "http
com,artistlink,api)/?mtv_id=1882444 20150522143250 {"url": "http://api.artistlink.com/?mtv_id=1882444", "mime": "text/html", "status": "200", "
com,art)/gallery/id--a20-b1822/marcel-duchamp-abstract-posters.htm?ui=1b60e37b310343229f2701d1590f6e01 20150528144216 {"url": "http://www.art.c
com,realself)/user/75853 20150522122606 {"url": "http://www.realself.com/user/75853", "mime": "text/html", "status": "200", "digest": "HYGAXEUD
com,realself)/question/safe-my-breast-imlants-removed-in-my-doctors-office 20150522121357 {"url": "http://www.realself.com/question/safe-my-bre
com,recchiuti)/108.html?area=01;id=efh4fa86 20150523134545 {"url": "http://www.recchiuti.com/108.html?area=01;id=eFh4Fa86", "mime": "text/html"
com,realityfanforum,forum)/index.php/topic,25773.msg625732.html 20150522160345 {"url": "http://forum.realityfanforum.com/index.php/topic,25773.
com,road-results)/racer/43326 20150525053406 {"url": "http://www.road-results.com/racer/43326", "mime": "text/html", "status": "200", "digest":
com,righthondaparts)/auto-parts/2002/honda/s2000/s2000-trim/6-speed-manual-engine/chassis-cat 20150524172352 {"url": "http://www.righthondapart
com,salisburypost)/article/20130315/sp01/130319785/-1/sp01111712-nc-secession-petition/china-grove-man-critical-after-wreck 20150528142256 {"ur
com,sail-world)/cabbage-tree-island-yacht-race-warm-up-for-rshyr-entries/90810 20150524081436 {"url": "http://www.sail-world.com/Cabbage-Tree-I
com,rotoworld)/articles/gol/47935/275/d%c3%a9j%c3%a0-vu-all-over-again 20150527162744 {"url": "http://www.rotoworld.com/articles/gol/47935/275/
com,rugsusa)/rugsusa/control/newsearch?brand_name=decor%20rugs&cid=10339&price_range=200-299 20150524141059 {"url": "http://www.rugsusa.com/rug
com,ruffian)/index.php?categoryid=1079&page=categoryproducts&store25=fbi35f4kaj0b2lgd2u2ij5l8o5&storeid=25&subcatid=&subsubcatid= 2015052316284
com,rubylane)/shop/teesantiqueorchard/ilist/,cs=vintage+collectibles:angel+firurine 20150529170501 {"url": "http://www.rubylane.com/shop/teesan
com,schuberth,blog)/en/tag/scuderia-ferrari 20150526133946 {"url": "http://blog.schuberth.com/en/tag/scuderia-ferrari/", "mime": "text/html", "
com,scienceblogs)/dispatches/2009/11/25/aclu-defends-christian-student 20150522180053 {"url": "http://scienceblogs.com/dispatches/2009/11/25/ac
com,beautybar)/b/fiafini/brand=borghese/type=body+lotions 20150528203223 {"url": "http://www.beautybar.com/b/fiafini/Brand=Borghese/Type=Body+L
com,bestbuy)/site/searchpage.jsp?_dyncharset=iso-8859-1&_dynsessconf=&cp=1&fs=saas&id=pcat17071&iht=y&keys=keys&ks=960&list=n&nrp=15&qp=&saas=s
com,benetton,blog)/mexico/tag/juegos 20150522173854 {"url": "http://blog.benetton.com/mexico/tag/juegos/", "mime": "text/html", "status": "200"
com,bikerkiss)/m/yiduheshang/2325923/2002_honda_vtx_1800_c.html 20150528061945 {"url": "http://www.bikerkiss.com/m/yiduheshang/2325923/2002_Hon
com,bikereg)/confirmed/21815 20150522100449 {"url": "https://www.bikereg.com/Confirmed/21815", "mime": "text/html", "status": "200", "digest":
com,autosport,forums)/topic/177009-help-iding-a-rolls-royce-and-fire-station-number?mode=threaded&pid=6015003 20150527180402 {"url": "http://fo
com,automd)/shops/pearl-auto-body-glass_314730 20150528025900 {"url": "https://www.automd.com/shops/pearl-auto-body-glass_314730/", "mime": "te
```


Figure 6.5: Cut of a sample CDX File.

- Create a Common crawl index sample.
- Build a WARC file.
- Parser Runner.

Additionally to creating a sample and building a WARC file services, a **parse WARC file** service was developed to run an specific parser in order to parse every document within a WARC file and copy the results to the file system by following the structure shown in Figure 4.3. Once the output files were generated for each parser, then the *comparator* can be run.

The arguments required are:

1. *Parse name*. Required to identify the parser output in the Comparator tool.
2. *Command to run the parser*. The parser runner use this command to run the parser as in command line.
3. *The path of the WARC filename*.

The **parse WARC file** reads every HTML document within the WARC file and executes the parser with the command provided. The output of the parser is then written in file system. The HTML documents extracted from the WARC file is temporarily written in file system. Therefore the command in arguments must be the one for parsing a file, e.g. *java -jar validatorNuAdapter.jar -f*.

A Common crawl record may have an empty HTML document, as a result of the response from an URL. For example the URL `http://www.utexas.edu/opa/blogs/research/2008/11/26/engineering-professor-gets-wired/feed/` returns an empty

document. Even if a empty document is a valid test case, it was decided to filter these occurrences in the WARC parsing process to avoid test duplication and test this scenario independently.

6.6 Summary

This chapter began by describing what is Common Crawl and how their data is stored in Amazon S3. Similarly, a description of Common Crawl Index was given. This index was useful for sampling Common Crawl data. Then, the Chapter went on to describe the procedures and methods used to retrieve the Common Crawl data and create a sample resulting in a WARC file. Such WARC file is used in the experiments shown in the following Chapter.

Chapter 7

Test framework execution

This chapter describes and discusses the experiments carried out in this investigation with the methods and test framework described in chapters 4 and 6.

The experiment consists in a description of how the test framework was executed, the results, the level of convergence and a discussion. In the results, a passed test means that the parser is part of the plurality, whereas a failed test means the opposite. Finally, the results, issues and limitations are discussed.

The experiments are presented in the order they were executed. Experiments from 1 to 4 were performed with data from the **Common Crawl Collection May 2015**. Then, experiment 5 used the **HTML5Lib test cases** as input data. Finally, though it was not planned, a last experiment with Common Crawl data from the **Collection July 2015** was performed to compare and validate the results of the first sample.

Following is shown the list of the parsers included in these experiments. Note that the parser with name *MScParser* is the parser developed in this project.

- **parse5** version 1.4.2.
- **html5lib** version 0.999999/1.0b7.
- **validatorNU** version 1.4.
- **anglesharp** version 0.9.0. Included after experiment 4.
- **jsoup** versions 1.8.2 included in experiments until experiment 3, then replaced to version 1.8.3.
- **MSc Parser**

Report details

NUMBER OF TESTS	1982
EQUALS	44
DIFFERENT	1938
DATE	02/08/2015 19:30:30

Test results

Parser name	Passed	Failed
parse5	1758	224
html5lib	1481	501
validatorNU	1594	388
MScParser	1033	949
Jsoup	90	1892

Figure 7.1: Number of passed and failed tests per parser in experiment 1

7.1 Common crawl sample from May 2015

This section describes the several experiments performed with data obtained from the Common crawl data set from May 2015. The online sample size calculator from Creative research Systems [1] was used to calculate the confidence interval (or margin of error) and confidence level for a given sample size.

7.1.1 Experiment 1

Setting

A initial sample size of 2000 index records was taken from the Common crawl index of May 2015 (CC-MAIN-2015-22). After removing empty HTML documents, the resulting size was **1982** documents.

Results

The Figure 7.1 shows the results of this first experiment. Only in 44 test cases the identical output was produced by all parsers. Furthermore, *Jsoup* shows a very low of passed tests, whereas *parse5* appear as the more successful.

The Figure 7.2 shows the compliance level of each parser in this experiment. This level is an indicator of how likely a parser output will converge with other parsers output. For example, the interpretation for the *parse5* parser has a probability of 88.7% that its output converge with the other parsers whereas *Jsoup* has a probability of 4.5%. This result is with reference to the Common Crawl data of May 2015 with a margin

[1] `http://www.surveysystem.com/sscalc.htm`

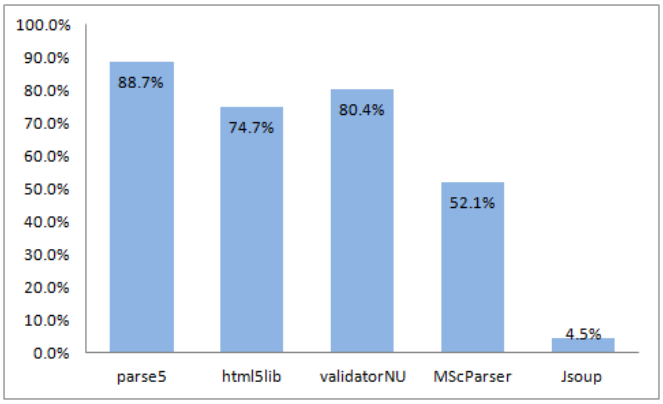


Figure 7.2: Convergence level in experiment 1

error of 2.2 and confidence level of 95%. Finally, the most surprising aspect of the data is the low level of *Jsoup* compared with the others, considering that it claims to be compliant with WHATWG.

Discussion

The unexpected high level of disagreement indicated a possible weakness in the test framework. By reviewing the differences in the outputs, it was found that the adapters had the following issues:

1. Some characters were not being displayed properly because the encoding UTF-8 was not correctly defined in the adapters.

2. The adapter of our parser was removing an extra line in the end of the file, causing a single difference in several files.

3. The *parse5* adapter was serialising only one percentage character when the DOM tree had two percentage characters (%%). The reason was the incorrect use of the function *console.log()* in javascript. The right solution is to specify the type of variable, e.g. *('%s',domTree)*.

7.1.2 Experiment 2

Setting

The same input of **1982** documents of the previous experiment was used. However, the issues in the adapters were fixed.

Report details

NUMBER OF TESTS	1982
EQUALS	70
DIFFERENT	1912
DATE	14/08/2015 23:39:05

Test results

Parser name	Passed	Failed
parse5	1962	20
html5lib	1625	357
validatorNU	1961	21
MScParser	1557	425
Jsoup	94	1888

Figure 7.3: Number of passed and failed tests per parser in experiment 2

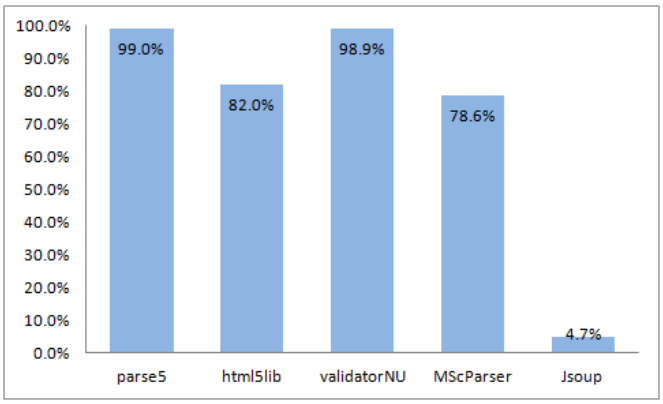


Figure 7.4: Convergence level in experiment 2

Results

From the data in Figure 7.3, it can be seen that the unanimous agreement increased compared with the previous experiment with a total 70 equal outputs. However, this number remains low because of the results of *Jsoup*. Furthermore, *parse5* and *html5lib* have almost the same number of passed tests, followed by the pair of *validatorNU* and *MScParser*. *Jsoup* remains with almost the same results as the previous experiment.

From the chart 7.4, it can be seen that both *parse5* and *validatorNU* levels practically converges with a probability of 99.0% and 98.9% respectively . *validatorNU* and *MScParser* shows an significant increment compared to the previous experiment whereas *Jsoup* rose slightly.

Discussion

Although a noticeable increase in convergence was obtained overall, the expected results were not obtained. After the manual revision of the differences, it was found that *Jsoup* was having the same issue with the extra line missing at the end of the output. Furthermore, according to the W3c specification a leading ”U+FEFF BYTE ORDER MARK” character must be ignored [6].*html5lib* and *MScParser* failed in removing the character in tests that had such character. This finding may indicate a missing test in the HTML5Lib test suite.

Once the code of *MScParser* was revised, it was found that it was missing the instruction to ignore the ”U+FEFF BYTE ORDER MARK” character. On the other hand *html5lib* probably has this compliance issue.

7.1.3 Experiment 3

Setting

In this experiment the *MScParser* issue related to leading ”U+FEFF BYTE ORDER MARK” character was fixed. Furthermore the new *Jsoup* version 1.8.3 was released and added to the test. It was decided to leave the old version 1.8.2 to compare results. Additionally the *Jsoup* adapter issue with extra lines was fixed.

Results

As it can be seen in Figure 7.5, both *Jsoup* versions had an identical result, maintaining a low agreement in relation to the other parsers. On the other hand, the data shows a decrease on agreements among the parsers compared with the previous experiment.

The chart 7.6 shows that *parse5*, *validatorNU* and *html5lib* levels decreased slightly whereas *MScParser* and both *Jsoup* versions presented the opposite behaviour.

Discussion

The most surprising aspect of the result is the identical output of both versions of *Jsoup*. It was expected that the new version had fixed compliance issues, however the same behaviour is shown as the old version. An error in the experiment could be the reason, though it was found nothing. However, it was found that both Jar files have the same size. Moreover, an online Maven repository reports the same size in both

Report details

NUMBER OF TESTS	1982
EQUALS	75
DIFFERENT	1907
DATE	27/08/2015 12:56:20

Test results

Parser name	Passed	Failed
parse5	1911	71
jsoup	97	1885
html5lib	1623	359
jsoup_old	97	1885
validatorNU	1911	71
MScParser	1567	415

Figure 7.5: Number of passed and failed tests per parser in experiment 3

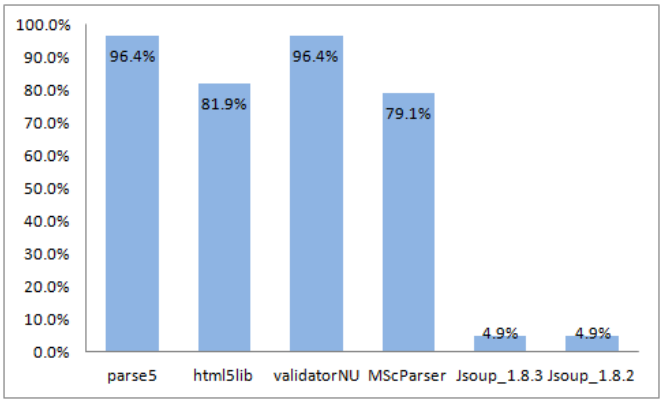


Figure 7.6: Convergence level in experiment 3

Report details

NUMBER OF TESTS	1982
EQUALS	71
DIFFERENT	1911
DATE	28/08/2015 10:31:13

Test results

Parser name	Passed	Failed
parse5	1978	4
jsoup	97	1885
html5lib	1625	357
AngleSharp	1546	436
validatorNU	1978	4
MScParser	1568	414

Figure 7.7: Number of passed and failed tests per parser in experiment 4

versions[2][3].

Additionally, as a result of having two parsers that completely converge, the other parsers results are affected. The reason is that it can decrease the possibility to have a plurality, and as a consequence a passed test. For example, when *parse5* and *validatorNU* agreed and the two *Jsoup* versions agreed, then no plurality is given.

7.1.4 Experiment 4

Setting

Jsoup 1.8.2 version was removed and *AngleSharp* parser was added to this experiment.

Results

From the data in Figure 7.7, it can be seen that *parse5* and *validatorNU* almost passed all tests, failing only 4 each. *html5lib* and *Jsoup* had the same results as the previous experiment.*MScParser* increased the number of passed tests by only one, practically maintaining the same results. On the other hand, *AngleSharp* showed a very similar result as *MScParser*, and consequently not to distant from *html5lib* numbers. This can be better appreciated in chart 7.8. Additionally, from this chart, it can be seen that *parse5* and *validatorNU* have almost a 100% probability of convergence.

[2]http://mvnrepository.com/artifact/org.jsoup/jsoup/1.8.2
[3]http://mvnrepository.com/artifact/org.jsoup/jsoup/1.8.3

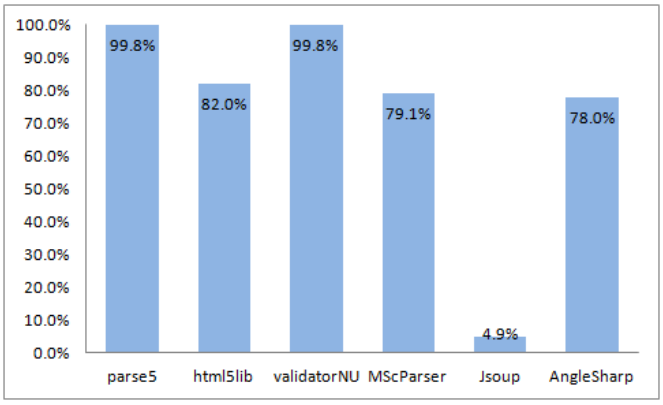


Figure 7.8: Convergence level in experiment 4

Discussion

The increment in *parse5* and *validatorNU* convergence level is the result of removing *Jsoup* 1.8.2 version. This allowed *parse5* and *validatorNU* to become plurality. On the other hand, the inclusion of *AngleSharp* did not change the parsers results in relation to the previous experiment, indicating that *html5lib*, *MScParser* and *AngleSharp* do not share the same test outputs. Otherwise, a plurality would have been found and their levels would have been increased.

7.2 HTML5Lib test suite

After processing a Common Crawl sample and reviewing the disagreements, it was found that most of them were because of specification differences while others were truly compliance bugs. However, the majority of these bugs should not occur if the parser passes the HTML5Lib test suite. This led to use the HTML5Lib input test cases in the test framework and identify those cases where the parsers are failing. Therefore, we will know which failing test cases shown in previous experiments are not included in the HTML5Lib test suite.

7.2.1 Experiment 5

Setting

In this experiment the current HTML5lib test suite was the input. The tests were extracted from the current versions of the HTML5suite tree constructor tests. Additionally, the cases for testing the scenarios with script flag set in false and document

Report details

NUMBER OF TESTS	1383
EQUALS	913
DIFFERENT	470
DATE	29/08/2015 13:34:06

Test results

Parser name	Passed	Failed
parse5	1367	16
jsoup	934	449
html5lib	1261	122
AngleSharp	1237	146
validatorNU	1369	14
MScParser	1369	14

(a) Excluding correct output

Report details

NUMBER OF TESTS	1383
EQUALS	913
DIFFERENT	470
DATE	29/08/2015 13:35:38

Test results

Parser name	Passed	Failed
parse5	1368	15
jsoup	934	449
html5lib	1261	122
AngleSharp	1247	136
validatorNU	1378	5
MScParser	1379	4
correct	1380	3

(b) Including correct output

Figure 7.9: Number of passed and failed tests per parser in experiment 5

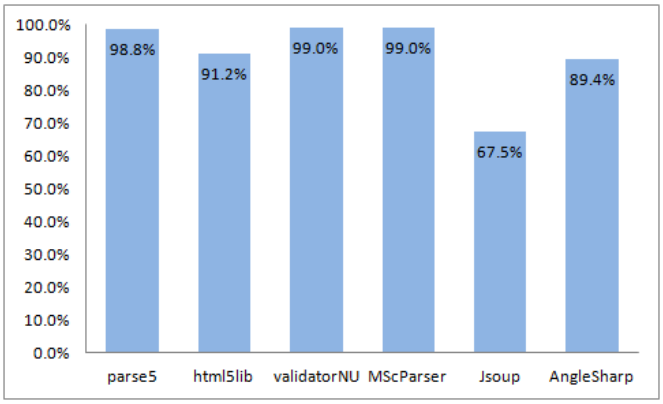
fragment were filtered leaving a total of **1383** tests. Moreover, a second experiment was reproduced including the correct output as the *correct* parser in the comparison.

The scripting flag set in false related tests were filtered because the adapters are configured to parse with scripting flag set in true as explained in subsection 4.1.2. On the other hand document fragment tests are beyond the scope of this dissertation.

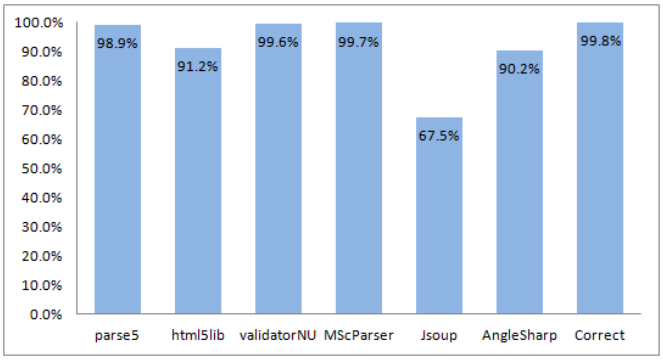
Results

The Figure 7.9 shows the results of the testing with the HTML5lib test cases, with and without the correct output. The inclusion of the correct outputs resulted in more tests passed for every parser but *html5lib*. Interestingly, the correct outputs shows three failed tests.

By comparing the charts in Figure 7.10, it can be seen that the levels of convergence barely differ. *MScParser* and *validatorNU* shows the most positive result followed



(a) Excluding correct output



(b) Including correct output

Figure 7.10: Convergence level in experiment 5

closely by *parse5*. On the other hand *Jsoup* highlights with a low level compared with the other parsers.

Discussion

Overall all parsers showed a high level of convergence and positive results, with the exception of *jsoup*. These results may prove that *jsoup* has a low compliance with the WHATWG specification. A possible explanation to this results is that *jsoup* does not use the HTML5Lib test suite in their testing (see Table 5.1). To support this statement, *jsoup* currently has 148 bugs open, in which several appear to be related to unexpected parsing outputs[42].

The inclusion of the correct outputs in the test led to improve the level of agreement among the parsers. The reason may be that the correct output helped as a tiebreaker in cases where there were not plurality.

On the other hand, it is interesting to note that the *correct* results does not have a 100% of convergence. After manually revising the test results, it was found that the

Figure 7.11: Number of passed and failed tests per parser in experiment 4

three failed test cases validate the *adjust SVG attributes* part of the specification, and that only *validatorNU* had the same result than the *correct* result (see Figure 7.11). This result may be explained by the fact that the *adjust SVG attributes* part differs between specifications, i.e. *AngleSharp* and *MScParser* follows W3C specification; and that some parsers may have not implemented this part yet, i.e. due to the ”living standard” property and constant evolution of WHATWG specification. Moreover, this example shows the case when correct or compliant results fail called *common-mode failure*, a limitation of this testing strategy.

7.3 Common crawl sample from July 2015

The following experiment was not planned, however, it was executed in order to compare and validate the results of the experiment 4 (7.1.4).

7.3.1 Experiment 6

Setting

Similar to experiment 4, a sample was taken from Common Crawl but from the Collection July 2015.

Report details

NUMBER OF TESTS	1070
EQUALS	32
DIFFERENT	1038
DATE	08/09/2015 20:57:44

Test results

Parser name	Passed	Failed
parse5	1069	1
jsoup	46	1024
html5lib	845	225
AngleSharp	846	224
validatorNU	1069	1
MScParser	830	240

Figure 7.12: Number of passed and failed tests per parser in experiment 6

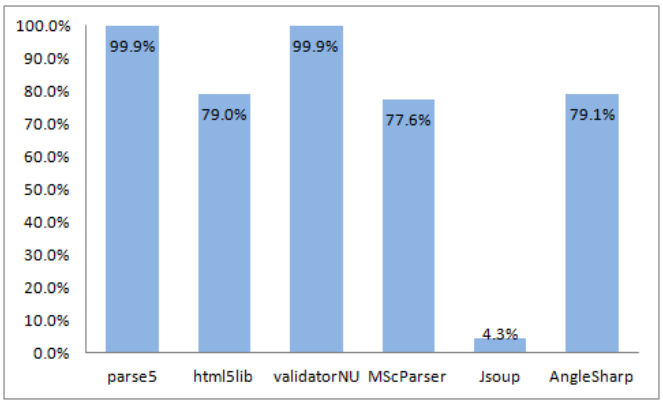


Figure 7.13: Convergence level in experiment 6

Results

From the Figures 7.12 and 7.13, it can be seen that compared to experiment 4, the results are almost equivalent to each other. *parse5*, *validatorNU* and *AngleSharp* passed tests slightly increased whereas *html5lib*, *MScParser* and *Jsoup* slightly decreased.

Discussion

The similar results between experiment 4 and 6, may suggest that the samples are uniformly distributed, and that they are a good representation of the whole Common Crawl data.

7.4 Summary

This chapter has described how the experiments with the test framework were performed. In addition, results and issues presented were discussed, however, the discussion of main findings found in these experiments are provided in more detail the next chapter.

Chapter 8

Results and discussion

8.1 MScParser evaluation

Additionally to the results obtained in the testing framework which shows a good compliance level with reference to W3C, The Figure 8.1 and Figure 8.2 shows the test results against the HTML5Lib test suite. HTML5Lib test suite is based on WHATWG specification whereas the parser is compliant with W3C specification, therefore several tests failed because of specification differences. These differences are shown in the following section.

The results show a fully W3C compliant parser according to the HTML5Lib test suite, built approximately in 6 weeks by three MSc. students. The Appendix A shows the estimated effort. The faster development is because of the requirement was only to be compliant with the specification, excluding other non functional requirements, e.g. performance or security.

The objective of building a functional HTML5 specification based parser was successfully met without major issues, thus it can be concluded that the specifications are reasonably understandable for any person with basic programming knowledge.

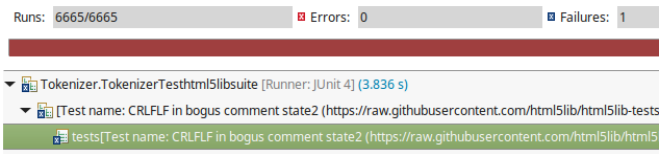


Figure 8.1: Tokenizer test results. The failed test is because of specification difference.

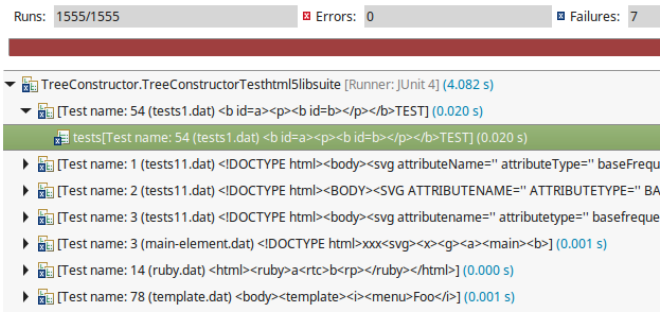


Figure 8.2: Tree constructor test results. The failed tests are because of specification differences.

8.2 Differences between W3C and WHATWG specifications

This section shows a compilation of differences found between W3C and WHATWG specifications as a result of the failed HTML5Lib tests of *MScParser*. These differences influenced directly the results of the experiments done with the test framework.

At the moment of writing this dissertation the current W3C specification was released the 28th October 2014 whereas the WHATWG last update was the 21th July 2015. The differences between the *Parsing HTML documents* sections of the specifications are:

1. In **Preprocessing the input stream** section, WHATWG have *Any LF character that immediately follows a CR character must be ignored, and all CR characters must then be converted to LF characters* whereas W3C have *All CR characters must be converted to LF characters, and any LF characters that immediately follow a CR character must be ignored*.

2. In **The stack of open elements** section, WHATWG categorizes the *menu* element as a special whereas W3C does not.

3. In **The *in body* insertion mode** section, WHATWG provides rules for processing the *menu* start tag token whereas W3C does not.

4. In **The *in body* insertion mode** section, WHATWG provides rules for processing the *menu* end tag token whereas W3C does not.

5. In **The rules for parsing tokens in foreign content** section, WHATWG provides rules for processing the *menu* start tag token whereas W3C does not.

6. In **The stack of open elements** section, WHATWG categorizes the *menuitem* element as a special whereas W3C does not.

7. In **The *in body* insertion mode** section, WHATWG provides rules for processing the *menuitem* start tag token whereas W3C does not.

8. In **The rules for parsing tokens in foreign content** section, W3C provides rules for processing the *main* start tag token whereas WHATWG does not.

9. In **Closing elements that have implied end tags** W3C provides rules for processing the *main* element whereas WHATWG does not.

10. In **The *in body* insertion mode** section, when processing a *body* or *html* end tag token, an *rtc* element in the stack of open elements could produce a parse error in W3C whereas in WHATWG is not mentioned.

11. In **The *in body* insertion mode** section, when processing a *rt* start tag token, W3C mentions the *rtc* element whereas WHATWG does not.

12. The **Adoption agency algorithm** differs between the specifications.

13. In **Creating and inserting nodes** section, the tables displayed for *adjust SVG attributes* instruction are different.

8.3 Level of agreement across parsers

According to the results obtained from the experiment 4 and 6, the Table 8.1 shows the probability of convergence with reference to common crawl data set of May 2015 with a margin error of 2.2 and a confidence level of 95%; and July 2015 with a margin error of 3 and a confidence level of 95%.

Similarly, the Table 8.2 shows the probability with reference to the HTML5Lib test suite.

In both tables, *validatorNU* has a higher probability to converge with another parsers whereas *Jsoup* has the lowest. Contrary to expectations, *MScParser* shows a notable contrast between Common crawl and HTML5lib test suite results. A possible explanation for this contrast is that HTML5lib test cases are small HTML documents designed to test an specific section of the specification whereas Common crawl documents are real web pages. The latter may represent a more complex test case that test

Parser	Probability Exp. 4(%)	Probability Exp. 6(%)
validatorNU	99.8	99.9
parser5	99.8	99.9
html5lib	82.0	79.0
MScParser	79.1	77.6
AngleSharp	78.0	79.1
Jsoup	4.9	4.3

Table 8.1: Probability of convergence with reference to Common crawl data set of may 2015. Ordered from highest to lowest.

Parser	Probability (%)
validatorNU	99.0
MScParser	99.0
parser5	98.8
html5lib	91.2
AngleSharp	89.4
Jsoup	67.5

Table 8.2: Probability of convergence with reference to HTML5Lib test suite. Ordered from highest to lowest.

several sections of the specification. *MScParser* was developed specifically to follow the W3C specification and pass the HTML5lib tests (those applicable to W3C specification), therefore it showed better results with isolated tests than with real web pages.

validatorNU and *parse5* were the two parses with more agreements. This may be the result of having five parsers that implemented the WHATWG specification, thus increasing the probability to converge.

On the other hand *MScParser* and *AngleSharp* shows similar results, because they are implementations of W3C specification. The lower level may be because, in case of a dispute result of specification differences, the plurality would be given to WHATWG implementations. This results in a failed test for W3C implementations. Further work is required to compare separately these two groups to ensure better results.

It is somewhat surprising that *html5lib* does not have a higher convergence, taking into account that the same developers maintain the HTML5Lib test suite. It is difficult to explain this result, but it might be related to a faulty adapter.

Finally *Jsoup* shows a poor level of convergence. As discussed in 7.2.1, *Jsoup* appears to have a low compliance with the WHATWG specification due to the lack of HTML5Lib test suite in their testing and high number of current bugs. However, a faulty adapter might be also a cause.

8.4 Disagreements analysis

Overall, it was found that the main reason of the disagreements among parsers evaluated was because of specification differences described in the previous section. Only the compliance bugs found of the parsers with more convergence will be discussed. These are *validatorNU* and *parse5*. This is because it takes a considerable amount of time to identify the failed tests that are not caused because of specification differences. Thus, developers are encouraged to test their parser against the HTML5Lib test suite and analyse the results.

As an observation, *parse5*, *html5lib* and *jsoup* do not allow parsing with a scripting flag set to false. Although this behaviour was not included in the testing, this is a compliance error.

validatorNU

As discussed in experiment 5 (Subsection 7.2.1) *validatorNU* showed three disagreements related to SVG attributes because of a *common-mode failure*. However, *validatorNU* is compliant with WHATWG specification whereas *AngleSharp* and *MScParser* are to W3C. Nevertheless, this is a compliance bug for *parse5*, *html5lib* and *jsoup*.

Two other cases of *common-mode failures* was presented in one of the test cases of Common Crawl. The first case consists in a line feed character right after a *textarea* start tag. The before inside a *table* scope. There were no plurality because of a tie between *validatorNU* and *parse5* result; and *html5lib* and *MScParser* result. She second case is pretty similar, but with a *pre* start tag instead of *textarea*. See Appendix section E.2.

Additionally, there was one test case that *validatorNU* failed because of its large size: 889.7kb. The *parse WARC file service* (see Subsection 6.5) returned a timeout error resulting in a failed test. This timeout error was designed to avoid halting the process, e.g. if the parser has an infinite loop.

The following compliance bugs were found in validatorNU:

1. Fail to return a U+FFFD REPLACEMENT CHARACTER from next input `FOO&#11111111111`. This bug is also presented in desktop browser Mozilla Firefox 40.0.3. See Appendix section F.1.

2. Fail handling *menuitem* according to WHATWG specification. However, this is compliant with W3C. Additionally, this bug is presented in desktop browsers

Mozilla Firefox 40.0.3 as well as in Google Chrome 45.0.5454.85 See Appendix section F.2.

3. It adds an extra character `&` after a malformed comment with a character reference inside. This bug could not be reproduced in Mozilla Firefox 40.0.3. See Appendix section F.3.

parse5

Experiment 5 7.2.1 showed that *parse5* fails to pass tests related to *ruby* element and children as well as parse correctly text in an ordinary[1] element.

On the other hand, in the experiment 4 (Subsection 7.1.4), *parse5* failed two tests because of the same reason as *validatorNU* with *common-mode failures* cases.

The following compliance bugs were found in parse5:

1. *parse5* stop working and shows the error `RangeError: Maximum call stack size exceeded` when parsing `<button><p><button>`. See Appendix section G.1.

2. A strange case found in the Common Crawl sample makes *parse5* moves Carriage Return and Line feed characters before a *table* element, instead of leaving them inside *table* scope. See Appendix section G.2.

8.5 HTML5Lib Missing tests

A missing test is that test case in which two parsers that passed the HTML5Lib test suite do not produce the same result. The following list shows possible missing tests for HTML5Lib test suite as a result of the analysis of results obtained in this study.

1. A leading *U+FEFF BYTE ORDER MARK* must be ignored if present, as described in the preprocessing [2] section of the specification.

2. A Line Feed character(LF) that is next to the *textarea* inside a *table* scope must be ignored if present, as described in *The "in body" insertion mode* [3] section.

3. A Line Feed character(LF) that is next to the *pre* inside a *table* scope must be ignored if present, as described in *The "in body" insertion mode* section.

[1] `http://www.w3.org/TR/html5/syntax.html#ordinary`
[2] `https://html.spec.whatwg.org/multipage/syntax.html#preprocessing-the-input-stream`
[3] `http://www.w3.org/TR/html5/syntax.html#parsing-main-inbody`

8.6 Specification bugs

A specification bug is the case in which two parsers that passed that are specification compliance, that is that both follows the specification rules, do not produce the same result. This suggests a hole or bug in the specification. However, this study did not found any case.

8.7 Summary

This chapter has reviewed the key findings of the experiments shown in previous chapter. It showed the level of agreement among parsers and thus an approximation of how compliant they are. Furthermore, the results of the parser implemented in this project proved to be compliant with the W3C specification. Moreover, it was useful for finding the differences between specifications. Such differences between specifications were the main reasons for disagreements among the parsers. Finally, the missing tests for the HTML5Lib test suite were listed, and no specification bug was found.

The conclusion of this dissertation is presented in the next chapter.

Chapter 9

Conclusions

The purpose of this study was to determine the level of compliance of HTML5 parsers by developing a testing framework based on N-version diversity. Moreover, to validate whether the HTML5lib test suite is enough to guarantee consistent behaviour i.e. to confirm if a parser that passes the HTML5 test suite is specification compliant. Prior to commencing these activities, a parser was developed in order to understand the specification and to know the implications involved.

It was expected that the test framework could find bugs and compliance errors in the parsers, and test cases that are not covered by the HTML5Lib test suite and possibly cases that are not covered by the HTML5 specification.

This study has shown that the HTML5 specification is reasonably understandable because the team managed to develop a full compliant parser without major problems within a reasonable amount of time. Furthermore, HTML5Lib test suite exposed those parts of the specification previously misinterpreted by the team.

The research has also shown that in general, that the parsers evaluated show a high convergence with the exception of *Jsoup*. The main reason for disagreements between parsers is the differences between specifications. This is interesting because the aim of the specifications is to ensure interoperability among software that use HTML technology. However, by having two different versions, it is difficult to achieve this objective. If the goal of an HTML5 parser developer is to converge as much as possible with other parsers, then, according to the results and parser survey obtained in this study, it is recommended to follow the WHATWG specification.

According to the experiments performed, it was confirmed that HTML5Lib test suite is a valuable source of test cases to ensure a high level of compliance with reference to the WHATWG specification. Although there were a few missing test cases

(Appendix E), the results show an extensive coverage of the test suite with reference to the WHATWG specification.

Regarding the parsers evaluated, the findings obtained from experiments suggest that in general *validatorNU* and *parse5* are the most compliant with the WHATWG specification. Whereas *MScParser* and *AngleSharp* to W3C. On the other hand, *html5lib* and *Jsoup* showed an acceptable and poor compliance respectively.

Additionally, the test framework showed that it is a useful tool for testing and revealing compliance violations in HTML5 parsers. The study proved that the test framework can be useful for finding errors not covered by HTML5Lib test suite (Section 8.4). Nevertheless, it is suggested to test against the HTML5Lib suite first. This may be a helpful tool for the test harness of HTML5 parser developers in order to ensure the best parser quality. On the other hand, no HTML5 specification issue or bug was found, which suggests that the specification is very robust.

9.1 Limitations

The major limitation of this testing approach is the *common-mode failures* with regards to N-Version Diverse Systems. This may produce incorrect results as shown with *adjust SVG attributes* in 7.2.1 where only *validatorNU* is compliant. However, the testing framework reports the opposite.

A significant weakness of this study was the lack of testing for *adapters*. As it can be seen throughout the development of the experiments, some difficulties with the adapters were encountered, generating wrong results. This can be avoided with properly conducted testing of adapters. This would be a fruitful area for further work.

The scope of this study was not fully complete in terms of the specification coverage. Parser behaviours like script execution or identification of encoding are not tested in the method presented. These form part of the specification, however not all parsers implement these features.

Another limitation faced is that the Common crawl data set may be biased. There is no conclusive information that explains the crawling process as explained in Chapter 6.

The final limitation encountered with the testing framework concerns the storing of output files and differences between output files. Once the comparator processes the output files, it generates the report and files. These files contain only the differences, deleting original output files. This limits the addition of new output files, e.g. in the

case of adding a new parser, it is necessary to restore original files from previously evaluated parsers in order to compare them.

The team also faced a significant challenge of teamwork. There were notable communication problems and team issues. Lack of trust was another reason why teamwork was difficult. Unfortunately, one of the members decided to work individually and ceased attending the weekly meetings. Otherwise the test framework would have had more parsers evaluated. Notwithstanding these challenges, the team successfully achieved the defined objectives of the project. Furthermore, the overall work carried out and results achieved may contribute to the HTML5 community.

9.2 Future work

Further work should be carried out mainly to overcome the limitations of the test framework developed in this project. Such further work may include enhancement of the comparison process. That is, to allow addition of a new parser without generating or restoring the original outputs of parsers previously tested.

It is recommended that further research be undertaken to test decoding and script execution behaviours. Parsers in evaluation would need to have these features implemented. It would be interesting to test the tree reconstruction caused by an script, as well of change in encoding while parsing.

Another possible area of future research could be to execute the experiments by grouping parsers per specification. This would result in greater convergence. Although it requires the installation of more parsers, as well as the development of their corresponding adapters.

This study should be repeated using different sources of test cases. It is suggested to obtain samples from other Common crawl collections or for instance from Alexa top websites[1]. In addition, filters could be implemented in the Common crawl sampling method to sample only those test cases with characteristics that are of interest to the research. The test cases could also be generated by developing a random input generator.

Finally, it is highly recommended to use this test framework to test browser based parsers in order to identify any disagreements among them. From research, it was found that there exists a high level of complexity to achieve complete installation and setup of browser engines. As a result of this complexity, such browser parsers were not

[1] http://www.alexa.com/topsites

utilized (installed) during the course of this project. Finding a disagreement using the test framework on such browser parsers will be a valuable contribution to the HTML and Web communities, which will help achieve the goal of interoperability.

Bibliography

[1] C. Anaya, *Analysing and testing HTML5 parsers*. M.s. thesis in submission, University of Manchester, 2015.

[2] Ian Hickson, "Error handling and Web language design." `http://ln.hixie.ch/?start=1074730185`, 2004. Accessed: 2015-04-24.

[3] J. Graham and G. Sneddon, "html5lib-tests." `https://github.com/html5lib/html5lib-tests`.

[4] Y. Minamide and S. Mori, "Reachability analysis of the html5 parser specification and its application to compatibility testing," in *FM 2012: Formal Methods* (D. Giannakopoulou and D. Mry, eds.), vol. 7436 of *Lecture Notes in Computer Science*, pp. 293–307, Springer Berlin Heidelberg, 2012.

[5] L. I. Manolache and D. G. Kourie, "Software testing using model programs," *Software - Practice and Experience*, vol. 31, no. February, pp. 1211–1236, 2001.

[6] *HTML5: A vocabulary and associated APIs for HTML and XHTML*, W3C Recommendation, 28 October 2014.

[7] W3C, "HTML, The Webs Core Language." `http://www.w3.org/html/`. Accessed: 2015-03-08.

[8] W3C, "Facts about W3C." `http://www.w3.org/Consortium/facts#history`. Accessed: 2015-04-23.

[9] W3C, "HTML, History." `http://www.w3.org/TR/html5/introduction.html#history-0`. Accessed: 2015-04-23.

[10] L. Stevens and R. Owen, *The Truth About HTML5*. Berkeley, CA: Apress, 2014.

[11] W3C, "DraconianErrorHandling." http://www.w3.org/html/wg/wiki/DraconianErrorHandling, 1998.

[12] WHATWGWiki, "FAQ HTML." https://wiki.whatwg.org/wiki/FAQ, 2015. Accessed: 2015-04-22.

[13] T. Bray, "DraconianErrorHandling." https://xml.web.cern.ch/XML/www.xml.com/axml/notes/Draconian.html, 2010.

[14] Ian Hickson, "HTML5 defines error handling." http://markmail.org/message/p5lkz5wrqh7loat3, 2006. Accessed: 2015-04-24.

[15] *Extensible Markup Language (XML)*, W3C Recommendation, 1.0 (Fifth Edition) 2008.

[16] W3C, "W3C Document Object Model." http://www.w3.org/DOM/#what. Accessed: 2015-08-06.

[17] *The Unicode Standard*, Version 7.0.0, 2014.

[18] D. author Grune, *Parsing Techniques A Practical Guide / by Dick Grune, Ceriel J. H. Jacobs.* New York, NY : Springer New York,, 2008.

[19] P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook, eds., *Testing Techniques in Software Engineering*, vol. 6153 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[20] W. standards project, "Acid Tests." http://www.webstandards.org/action/acid3/. Accessed: 2015-04-26.

[21] HTML5Test, "HTML5Test." https://html5test.com/index.html. Accessed: 2015-04-27.

[22] "cl-html5-parser — Quickdocs." http://quickdocs.org/cl-html5-parser/. Accessed: 2015-04-27.

[23] "Dev.Opera Ragnarök Viking Browser With HTML5 Parser!." https://dev.opera.com/blog/ragnarok-viking-browser-with-html5-parser/. Accessed: 2015-04-27.

[24] E. Seidel, "The HTML5 Parsing Algorithm." https://www.webkit.org/blog/1273/the-html5-parsing-algorithm/, 2010. Accessed: 2015-04-27.

[25] WHATWG Wiki, "Parser tests." https://wiki.whatwg.org/wiki/Parser_tests, 2013. Accessed: 2015-04-27.

[26] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," p. 38, Aug. 2012.

[27] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A Comparative Study on Automated Software Test Oracle Methods," in *2009 Fourth International Conference on Software Engineering Advances*, pp. 140–145, IEEE, Sept. 2009.

[28] M. Last and M. Freidman, "Black-box testing with info-fuzzy networks," *Artificial Intelligence Methods in Software Testing, World Scientific*, pp. 21–50, 2004.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Jan. 1995.

[30] B. Parhami, "Voting algorithms," *Reliability, IEEE Transactions on*, vol. 43, pp. 617–629, Dec 1994.

[31] D. Herman, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Servo Web Browser Engine using Rust.".

[32] "Gumbo-parser." https://github.com/google/gumbo-parser.

[33] "Parse5." https://www.npmjs.com/package/parse5.

[34] "The Validator.nu HTML Parser." https://about.validator.nu/htmlparser/. Accessed: 2015-05-04.

[35] "Newest 'jsoup' Questions - Stack Overflow." http://stackoverflow.com/questions/tagged/jsoup, note = Accessed:2015-08-25.

[36] E. Baykan, M. Henzinger, S. F. Keller, S. De Castelberg, and M. Kinzler, "A Comparison of Techniques for Sampling Web Pages," Feb. 2009.

[37] "Common Crawl." http://commoncrawl.org/. Accessed: 2015-08-17.

[38] S. Merity, "Crawling Strategy of newer Crawls." https://groups.google.com/d/msg/common-crawl/IGa7E680NUs/OQhYYzt9DAAJ.

[39] "Introduction to Amazon S3 - Amazon Simple Storage Service." http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html. Accessed:2015-08-21.

[40] "Announcing the Common Crawl Index! Common Crawl Blog." http://blog.commoncrawl.org/2015/04/announcing-the-common-crawl-index/. Accessed:2015-08-17.

[41] I. Kreymer, "CDX Index Format." https://github.com/ikreymer/pywb/wiki/CDX-Index-Format. Accessed:2015-08-18.

[42] "Jsoup Issues." https://github.com/jhy/jsoup/issues, note = Accessed:2015-08-29.

Appendix A

Team activities

The Table A.1 shows the distribution of work among team members. My work effort is that of the column with the name Jose. This table was done along with Carlos Anaya.

Please bear in mind that this is just an estimation calculated from a backlog and notes, and that it only contemplates **design and implementation** of the software, excluding time spent for exams. Additionally, this is the work realised from **March to July**. According to the plan, it was expected each member work half time in the period from March to May, that is **30 days**; and full time in thye period from June to July, that is **40 days**. In total each member should worked **70 days** in the period from **March to July**. As it can be seen, Xiao days is incomplete. The reason is that this person stopped coming to meetings around mid-June.

Furthermore, the Table A.2 shows a log maintained daily with detailed description of the tasks done by myself.

Component	**Activity**	**Estimated effort (days)**	**Team members**		
			Carlos	**Jose**	*Xiao*
Parser Implementation	Preprocessing	1	100	0	0
	Tokenizer	15	33	33	33
	Tree constructor	15	25	25	50
	Algorithms	10	50	50	0
	Test harness	4	25	75	0
	Testing	25	40	40	20
	DOM implementation	7	100	0	0
Adapters	Jsoup	2	100	0	0
	Anglesharp	2	100	0	0
	Html5lib	2	0	50	50
	parse5	2	0	100	0
	validator.nu	2	0	100	0
Comparator	Report generator	7	25	75	0
	Algorithm	5	0	100	0
	Output processing	5	100	0	0
Web Application	Comparator UI	5	100	0	0
	Tracer UI	5	100	0	0
	Input test	5	100	0	0
Common crawl	Design and implementation	30	0	100	0
Tracer	Design and implementation	15	100	0	0
	Total	164	72,45	71,95	18,45

Table A.1: Distribution of effort

Activity	Start Date	Finish date	Days	Details
Adding of parse5 Javascript HTML5 parser to the Comparator	5/jun/15	8/jun/15	4	parse5 is a javascript HTML5 parser implementation that claims to be WHATWG spec complaint. This activity consisted in setup the parser and develop the adapter to serialise into the html5lib format
Development of parser's output comparison	9/jun/15	11/jun/15	3	This Java component was made to compare the outputs of the different parsers, and then apply a plurality voting algorithm to decide which output were correct or not
Fix HTML5Lib adapter	12/jun/15	12/jun/15	1	It was not serialising correctly.
Common crawl	15/jun/15	24/jun/15	10	Read of a remote file Connection http. Read of a local file. Process of a gzip file and a stream. Retrieve of a single record with a particular offset. Change to call parsers from the same java process. Able to process a common crawl archive, some problems still though (argument size limit)
Common crawl and report	25/jun/15	26/jun/15	2	Able to process a particular number of records of a common crawl file and show the results in the web interface. Fixed a problem with argument size limit creating a tmp file. Parsers take too much time with big inputs, specifically jsoup (245kb). A timeout was implemented.
Fixing crawling process when Jsoup hangs	29/jun/15	29/jun/15	1	Jsoup was hanging the overall process and it was not being stopped correctly. Now, after the timeout all parsers executions are destroyed
Fixing bugs with crawling	30/jun/15	01/jul/15	2	Run a local file because a http error. However is still having an error when the report.xml is too big and exceed the memory limit
Refactor of Report and addition of partitioned report	2/jul/15	3/jul/15	2	Partitioned report. The report now generates several parts so a main report contains the totals of all parts. This allows to process beyond the limit of a java DOM size
Fixed plurality algorithm	6/jul/15	6/jul/15	1	Fix plurality algorithm because it was changed. Fix error when comparing, if a parser does not parse correctly or have a timeout thus produce an error instead of a tree
Change HTML crawler to run only one parser and write the output to FS	7/jul/15	8/jul/15	2	Major update to run the crawler with an particular parser and common crawl file and write the output of the parser to FS. Comparator is not called anymore, instead it will run after several parsers have been run
Install and run the validator.ns parser	9/jul/15	9/jul/15	1	This is a java parser but it needs a html5lib format serializer. Is particularly different than the others because it parse in a stream way with SAX capabilities.
Fixing comparator, initial version was not working	10/jul/15	10/jul/15	1	Initial run of the updated crawler process and comparator , though having problems with the file system (coded for windows and not working in linux)
Testing with more than 2500 html documents...	13/jul/15	13/jul/15	1	Run 3645 tests. We have performance and storage issues

Reading about web sampling	14/jul/15	14/jul/15	1	Principled Sampling for Anomaly Detection and A COMPARISON OF TECHNIQUES FOR SAMPLING WEB PAGES
Researching how to get random records form Common crawl	15/jul/15	17/jul/15	3	Two frameworks for processing ZipNum Sharded CDX Cluster, which is the format of the CC index. Using the java webarchive-commons.
Design and code a application to get random sampling from common crawl	20/jul/15	24/jul/15	5	An application was developed to get random records form a CC index from Amazon S3. This require a Amazon S3 account. This records have the information to retrieve the HTML from the common crawl corpus.
Create a sample	27/jul/15	27/jul/15	1	2000 indexes were retrieved from the last collection(CC-MAIN-2015-22) of the CC index randomly. Then with these indexes, the records documents were retrieved from the Common crawl corpus and finally a compressed WARC was created with the sample. All this was executed on a Amazon EC2 instance to minimize transferred data
Run the comparator tool with the sample	28/jul/15	28/jul/15	1	The sample was parsed by the parsers. 1982 HTML documents of 2000 were parsed due to 18 records were not valid HTML responses
Finish the validator.nu adapter	29/jul/15	29/jul/15	1	It was necessary to use the SAX instead of a DOM because the DOM implementation has problems with XML restrictions i.e. invalid characters in attributes names. I ran the html5lib tests and two tests fail.
Run the comparator tool with the sample and fix adapters bugs.	30/jul/15	31/jul/15	2	Several bugs were fixed in the adapters in order to have the correct output. i.e. Attributes order, extra lines, wrong encoding
Write dissertation and do experiments and final fixes	03/aug/15	11/sep/15	20	Mainly write dissertation. Run experiments in the test framework and do last changes and fixes.

Table A.2: Log with tasks done by myself. Period from June to September.

Appendix B

HTML5 Parser Architecture

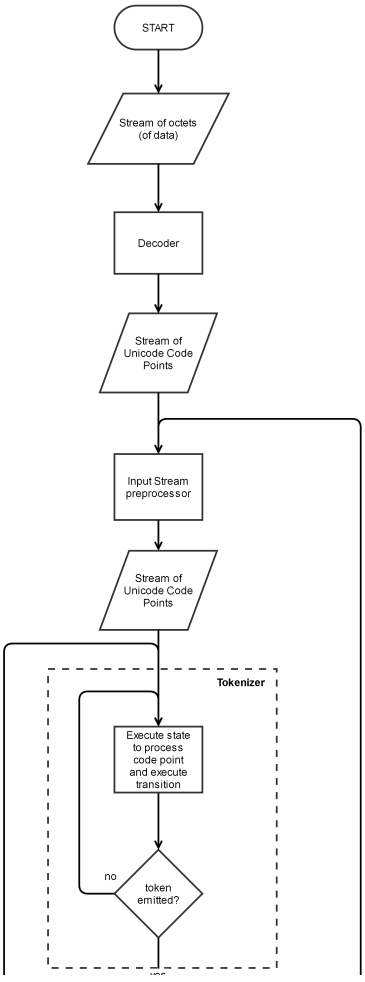


Figure B.1: Parser architecture part 1

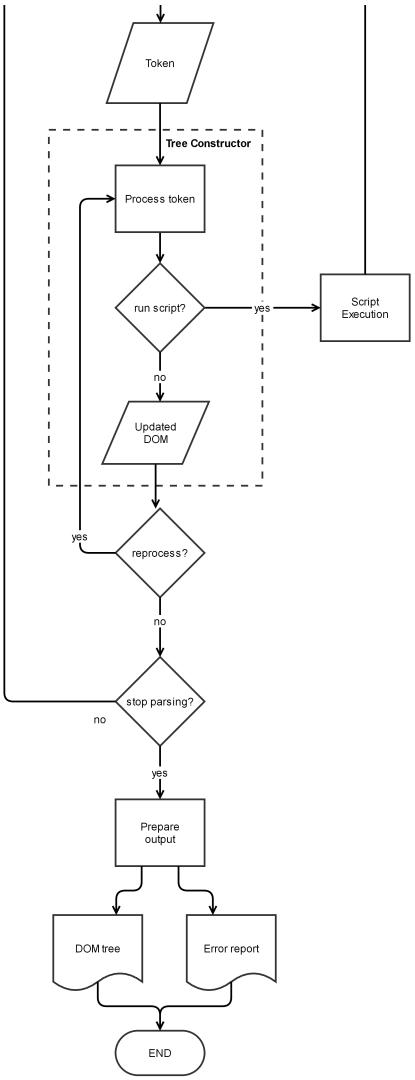


Figure B.2: Parser architecture part 2

Appendix C

HTML5 Parser Survey

The following pages shows a Table C.1 with the repository and references consulted in this parser survey. A second Table C.2 shows key data related to each HTML5 parser.

Parser name	Source Repository	URL
AngleSharp	https://github.com/FlorianRappl/AngleSharp	http://anglesharp.github.io/ http://www.codeproject.com/Articles/609053/AngleSharp https://github.com/FlorianRappl/AngleSharp/wiki
parse5	https://github.com/inikulin/parse5	https://www.npmjs.com/package/parse5
Html5lib	https://github.com/html5lib/html5lib-python	https://pypi.python.org/pypi/html5lib
jsoup	https://github.com/jhy/jsoup/	http://jsoup.org/
validator.nu	https://hg.mozilla.org/projects/htmlparser/	https://about.validator.nu/htmlparser/
cl-html5-parser	https://github.com/copyleft/cl-html5-parser	https://github.com/copyleft/cl-html5-parser
html5ever	https://github.com/servo/html5ever	https://github.com/servo/html5ever
tagsoup	https://github.com/ndmitchell/tagsoup.git	http://community.haskell.org/ ndm/tagsoup/
html	https://github.com/dart-lang/html	https://pub.dartlang.org/packages/html
gumbo	https://github.com/google/gumbo-parser	https://github.com/google/gumbo-parser
Hubbub	http://source.netsurf-browser.org/libhubbub.git/	http://www.netsurf-browser.org/projects/hubbub/
html5-php	https://github.com/Masterminds/html5-php	http://masterminds.github.io/html5-php/ https://packagist.org/packages/masterminds/html5
html	https://github.com/golang/net/tree/master/html	https://godoc.org/golang.org/x/net/html#pkg-subdirectories
HTML::HTML5::Parser	https://bitbucket.org/tobyink/p5-html-html5-parser	http://search.cpan.org/ tobyink/ HTML-HTML5-Parser-0.301/lib/HTML/HTML5/Parser.pm

Table C.1: HTML5 sources and references

Parser name	Language	Html5lib tests[1]	Test harness	Specification Compliance	Browser engine	Author	Licence	Current binary version	Last Update
AngleSharp	C#	Yes	Have their own test suite	W3C[9]	No	Florian Rappl	MIT License[3]	0.8.9	29/07/2015
parse5	javascript	Yes	Html5lib test suite	WHATWG	No	Ivan Nikulin	MIT License	1.5.0	24/06/2015
Html5lib	python	Yes	Html5lib test suite	WHATWG	No	James Graham, Geoffrey Sneddon, Lukasz Langa	MIT License	1.0b7	07/07/2015
jsoup	Java	No	Have their own test suite	WHATWG	No	Jonathan Hedley, Mozilla	MIT License	1.8.3	02/08/2015
validator.nu	Java	Yes[2]	Html5lib test suite	WHATWG	C++ port in Gecko	Henri Sivonen	MIT License	1.4	29/05/2015[4]
cl-html5-parser[5]	Common Lisp	Yes	Html5lib test suite	WHATWG	No	Thomas Bakketun, Copyleft	GNU Lesser General Public License v3.0	Unknown	17/07/2014
html5ever[6]	Rust	Yes[7]	Html5lib test suite	WHATWG	Servo	Adam Roben, Akos Kiss,Wojciech Zarazek-Winiewski	Apache License,Version 2.0	Not released	31/08/2015[8]
tagsoup	Haskell	No	Have their own test suite	No specified	No	Neil Mitchell	BSD3	0.13.3	01/10/2014
html [5]	Dart	Yes	Html5lib test suite	WHATWG	No	Dart team	MIT License	0.12.1+2	06/07/2015
gumbo	C99	Yes	Html5lib test suite	WHATWG	No	Google	Apache License, Version 2.0	0.10.1	30/04/2015
Hubbub	C	Yes	Html5lib test suite	WHATWG	Probably in future, NetSurf engine	NetSurf	MIT License	0.3.1	08/03/2015
html5-php	Php	No	Have their own test suite	W3C	No	Masterminds	MIT License	2.1.2	07/06/2015
html	Go	Yes	Html5lib test suite	WHATWG	No	Go Authors	BSD	Unknown	28/07/2015
HTML::HTML5::Parser	perl	Yes	Html5lib test suite	No specified	No	Toby Inkster	GNU General Public License	0.301	08/07/2013

[1] The value ”No” means that there is no information provided nor tests included in the source code that indicates the use of HTML5lib test suite
[2] Assumed from the acknowledgements section of their page
[3] Seems to be MIT with some modifications
[4] This is the date of the last update of the source code, since the binay file have not updated. The binary file last update in maven is 05/06/2012.
[5] Html5lib port
[6] Currently under development
[7] Currently passes all tokenizer tests and most of tree builder.
[8] This is the date of the last update of the source code, since this parser is under development
[9] W3C compliant with some WHATWG extensions https://github.com/AngleSharp/AngleSharp

Table C.2: HTML5 parser survey

Appendix D

Files used from the HTML5Lib test suite

tests1.dat	tests14.dat	tests26.dat	main-element.dat
tests2.dat	tests15.dat	adoption01.dat	pending-spec-changes-plain-text-unsafe.dat
tests3.dat	tests16.dat	adoption02.dat	pending-spec-changes.dat
tests4.dat	tests17.dat	comments01.dat	plain-text-unsafe.dat
tests5.dat	tests18.dat	doctype01.dat	ruby.dat
tests6.dat	tests19.dat	domjs-unsafe.dat	scriptdata01.dat
tests7.dat	tests20.dat	entities01.dat	tables01.dat
tests8.dat	tests21.dat	entities02.dat	template.dat
tests9.dat	tests22.dat	foreign-fragment.dat	tests_innerHTML_1.dat
tests10.dat	tests23.dat	html5test-com.dat	tricky01.dat
tests11.dat	tests24.dat	inbody01.dat	webkit01.dat
tests12.dat	tests25.dat	isindex.dat	webkit02.dat

Appendix E

Possible HTML5Lib test suite missing tests

Totally, the number of possible missing tests found were 5. Additionally to the three shown here, the bugs F.3 and G.2 are also considered as missing tests.

E.1 *U+FEFF BYTE ORDER MARK* character

This test case tests if the leading *U+FEFF BYTE ORDER MARK* character is removed.

Input

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
  TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Listing E.1: Input test case

Expected output

```
| <!DOCTYPE html "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-transitional.dtd">
| <html>
|   <head>
|   <body>
```

Listing E.2: Correct output

Wrong Output

```
| <html>
|   <head>
|   <body>
|     "    "
```

Listing E.3: Wrong Output

E.2 Line feed next to textarea

This test case involves Foster parenting when a *textarea* is after a *table*. However, *HTML5Lib MScParser* failed to ignore a Line Feed character that is next to the *textarea*. Is interesting that they do not fail if *table* is not present. In the following coding snippets, the character cannot be seen. However is the first character in the input.

Input

```
<table><textarea>
test
```

Listing E.4: Input test case

Expected output

```
| <html>
|   <head>
|   <body>
|     <textarea>
|       "test"
|     <table>
```

Listing E.5: Correct output

Wrong Output

```
| <html>
|   <head>
|   <body>
|     <textarea>
|       "
test"
|     <table>
```

```
```

Listing E.6: Wrong Output

E.3 Line feed next to pre

This case is exactly the same as the Line feed next to *textarea* start tag, but with the *pre* start tag.

Input

```
<table><pre>
test
```

Listing E.7: Input test case

Expected output

```
| <html>
|   <head>
|   <body>
|     <pre>
|       "test"
|     <table>
```

Listing E.8: Correct output

Wrong Output

```
| <html>
|   <head>
|   <body>
|     <pre>
|       "
test"
|     <table>
```

Listing E.9: Wrong Output

Appendix F

ValidatorNU bugs

F.1 Character reference bug

Fail to return a U+FFFD REPLACEMENT CHARACTER (0xEF 0xBF 0xBD (efbfbd) in UTF-8). Bug in section Tokenizing character references.

Input

```
FOO&#11111111111
```

Expected output

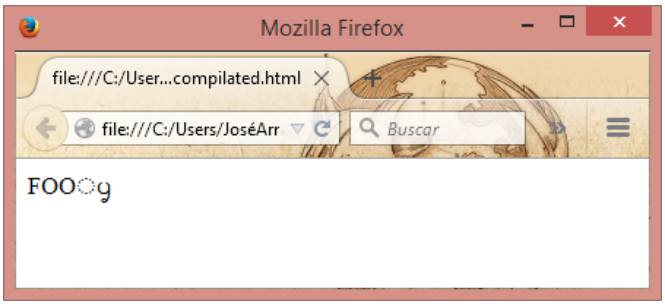
```
00000000  7c 20 3c 68 74 6d 6c 3e  0a 7c 20 20 20 3c 68 65  |· <html>·|   <he|
00000010  61 64 3e 0a 7c 20 20 20  3c 62 6f 64 79 3e 0a 7c  |ad>·|   <body>·|
00000020  20 20 20 20 20 22 46 4f  4f ef bf bd 22 0a        |     "FOO...".|
0000002e
```

Listing F.1: Plurality and correct output. Correct hexadecimal values

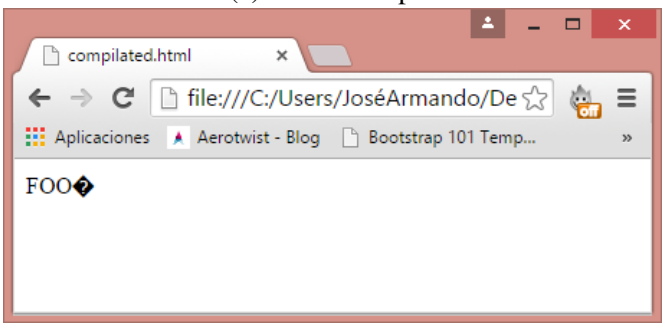
ValidatorNU output

```
00000000  7c 20 3c 68 74 6d 6c 3e  0a 7c 20 20 20 3c 68 65  |· <html>·|   <he|
00000010  61 64 3e 0a 7c 20 20 20  3c 62 6f 64 79 3e 0a 7c  |ad>·|   <body>·|
00000020  20 20 20 20 20 22 46 4f  4f e1 a7 87 22           |     "FOO..."|
0000002d
```

Listing F.2: ValidatorNU Output. Incorrect hexadecimal values



(a) Firefox output



(b) Chrome output

Figure F.1: Character reference bug

F.2 Menuitem bug

Fail to handle *menuitem* element according to WHATWG specification.

Input

```
<!DOCTYPE html><body><menuitem>A
```

Expected output

```
| <!DOCTYPE html>
| <html>
|   <head>
|   <body>
|     <menuitem>
|       "A"
```

Listing F.3: Correct output according to WHATWG specification

ValidatorNU, Firefox and Chrome output according to W3C specification

```
| <!DOCTYPE html>
| <html>
|   <head>
|   <body>
|     <menuitem>
|       "A"
```

Listing F.4: ValidatorNU Output.

F.3 Extra character after malformed comment bug

It adds an extra character `&` after a malformed comment containing a character reference. This bug was found in the Common crawl sample.

Input

```
<!&#8212; >
```

Expected output

```
| <!-- &#8212; -->
| <html>
|   <head>
|   <body>
```

Listing F.5: Correct output according to majority.

ValidatorNU

```
| <!-- &#8212; -->
| <html>
|   <head>
|   <body>
|     "&"
```

Listing F.6: ValidatorNU Output.

Appendix G

Parse 5 bugs

G.1 Button tag bug

This test is covered by HTML5Lib test suite, however, *parse5* stopped working with this scenario.

```
/home/jose/node_modules/parse5/lib/tree_adapters/default.js:0
(function (exports, require, module, __filename, __dirname) { 'use strict';

RangeError: Maximum call stack size exceeded
    at Object.exports.getTagName (/home/jose/node_modules/parse5/lib/tree_adapters/
        default.js)
    at Parser._isSpecialElement (/home/jose/node_modules/parse5/lib/tree_construction
        /parser.js:871:31)
    at genericEndTagInBody (/home/jose/node_modules/parse5/lib/tree_construction/
        parser.js:1899:15)
    at Object.endTagInBody [as END_TAG_TOKEN] (/home/jose/node_modules/parse5/lib/
        tree_construction/parser.js:2002:17)
    at Parser._processToken (/home/jose/node_modules/parse5/lib/tree_construction/
        parser.js:619:38)
    at Parser._processFakeEndTag (/home/jose/node_modules/parse5/lib/
        tree_construction/parser.js:663:10)
    at buttonStartTagInBody (/home/jose/node_modules/parse5/lib/tree_construction/
        parser.js:1351:11)
    at buttonStartTagInBody (/home/jose/node_modules/parse5/lib/tree_construction/
        parser.js:1352:9)
    at buttonStartTagInBody (/home/jose/node_modules/parse5/lib/tree_construction/
        parser.js:1352:9)
    at buttonStartTagInBody (/home/jose/node_modules/parse5/lib/tree_construction/
        parser.js:1352:9)
```

Listing G.1: Parse 5 error when processing <button><p><button>

G.2 Table and Carriage Return(CR) characters references

This is a strange case found in the Common Crawl sample which parse5 moves Carriage Return and Line feed characters before a *table* element, instead of leaving them there. Note that the input contains character references for the Carriage Return(CR). Actually, Carriage Return(CR) characters are sent to the Tree Constructor, even if normal Carriage Return(CR) characters were removed in the pre processing stage. Thus the tree constructor receives a Carriage Return(CR) character followed by a Line feed(LF).

Input

```
<table><td></td>&#13;
&#13;
<tr><td></td>&#13;
```

Listing G.2: Input test case

Expected output

```
| <html>
|   <head>
|   <body>
|     <table>
|       <tbody>
|         <tr>
|           <td>
|             "


"
|         <tr>
|           <td>
|             "

"
```

Listing G.3: Correct output according to majority.

ValidatorNU

```
| <html>
|   <head>
|   <body>
|     "



"
|     <table>
|       <tbody>
|         <tr>
|           <td>
|         <tr>
|           <td>
```

Listing G.4: Parse5 Output.