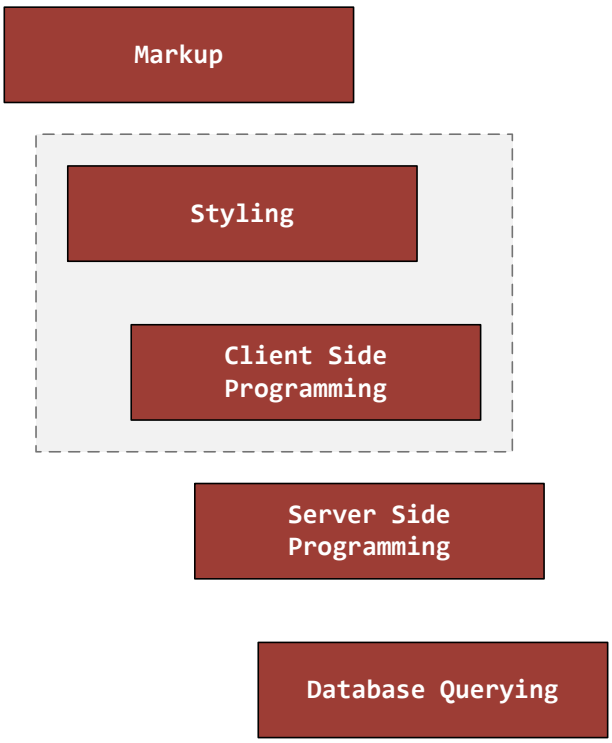
Dynamic Styling in Web Development

Creating PDSS, a Powerful Dynamic Styling Solution



Master's thesis by:
Daniel Solsø Korsgård
Michael Stampe Knudsen

Aalborg University, June 12th, 2009

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Phone 96 35 80 80, Fax 98 15 97 57
http://www.cs.aau.dk

Abstract:

In an earlier project, nine existing solutions for unified web development of RIAs were researched. It became apparent that none of these solutions have integrated styling mechanisms that aid in RIA development. In this project, some existing styling solutions are examined to determine their styling capabilities. Nine use cases are presented and used to test the capabilities of the styling solutions. The test shows that the most promising styling solutions are the least widespread.
The goal of the project is to integrate dynamic styling mechanisms into a general purpose programming language. The integrated styling mechanism must employ more expressive power than that of CSS. This includes: More powerful selectors, ability to constrain the style of elements on one another, and advanced expressions. The solution, an extension for JavaScript, is designed. The design is described with complete syntax and semantics, and a prototype of the design is implemented.
The prototype is then tested by implementing the nine use cases.
The test shows that the developed solution is capable of implementing all the use cases. In most cases, the implementations in the developed solution are found to be more concise than the implementations in the existing solutions.
The conclusion is that a powerful and dynamic styling solution has been developed.

Title:
Dynamic Styling in Web Development

Theme:
Programming Technology

Project period:
DAT6, spring 2009

Project group:
d625a

Authors:
Daniel Solsø Korsgård
Michael Stampe Knudsen

Supervisor:
Kurt Nørmark

Printcount: 4

No. of pages: 111

Appendixcount: 2

Completed:
12-06-2009 at Aalborg University

The content of this report is accessible without boundary, publication, however, is only allowed through an agreement with the authors.

Contents

Preface

This report has been written during the 2009 Dat6-project period by computer science group d625a at Aalborg University. The report is addressed to other students, supervisors and anyone else who might be interested in the subject. To read and understand the report, it is necessary to have knowledge of the most basic computer related terms and web development.

Abbreviations and acronyms will at first appearance be written in parenthesis to avoid breaking the reading stream. Specification of gender in the report is not to be understood as suppression or any form of political/religious position. The gender is only specified to simplify the process of writing for the authors.

References to sources are marked by [#], where # refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found in the last chapter of the report. All source code created during the project is located on a compact disc, which is attached to the last page of the report. On the compact disc is also located an electronic version of this report in PDF format.

Chapter 1

Introduction

In recent years, the concept of Rich Internet Applications (RIA) has emerged. These RIAs are highly dynamic and visually advanced web applications. Many different languages are used in the development of a RIA, and each of these languages is concerned with its own area of the application. The languages are: a markup language, a styling language, a client side and server side programming language, and a database query language.

In a previous project[1], we have looked at unification of the languages used in web application development. Nine different solutions to unified web development were examined in the report. These were Echo Web Framework[2], GOA WinForms[3], Google Web Toolkit[4], Helma[5], HOP[6], Links[7], Script#[8], Silverlight[9], and Swift[10]. After the solutions were examined, a development experiment was conducted, to test how uniform the solutions are, and to get hands-on experience of how the uniformity affects a development process. In the experiment, the same web application was implemented in Links, Google Web Toolkit, GOA WinForms and traditional web development (HTML, CSS, JavaScript, PHP and SQL). The conclusion of the experiment was that language uniformity in web development does yield a better development process.

However, a problem common to these solutions is the lack of visual styling functionality. Links does not offer any kind of styling itself, but relies completely on CSS. Google Web Toolkit and GOA WinForms both support styling through properties on their own graphical elements (which are abstractions over HTML elements). Google Web Toolkit however also supports user defined CSS. With styling solely through the use of properties, you navigate through a data structure manually, and then write or read values to and from properties on that data structure. This is, for instance, the traditional way to control the behavior of graphical elements in desktop applications. When using rules, as known from CSS, the assignment of values to certain properties in the data structure is based on rules. That is, a rule is specified to match certain elements in the data structure (often a tree) based on certain relations such as ancestor/descendant relation. All those matched elements are styled according to the specification of the rule.

Styling should be flexible and expressive, and none of the examined solutions offer that. The approach to styling with properties is very flexible, because the graphical elements can be styled exactly as one can imagine, only limited by the visual capabilities of the graphical elements themselves. The CSS approach is more expressive, in terms of the ability to style a large number of elements based on rules. CSS is however not as powerful and flexible. It

is for instance not possible to style an arbitrary element based on the properties of another arbitrary element.

When taking the idea of unified web development into account, we see another problem. The problem is that CSS is another layer in itself, and does its own thing. Ideally, the concept of styling should be tightly integrated into the web development platform. JavaScript already has the ability to navigate the DOM and change the CSS properties for all graphical elements. But applying new rules or changing existing rules through JS is cumbersome.

A solution to this would be a more expressive styling mechanism integrated into the underlying programming language (UPL). The underlying programming language is the fundamental language of the used web development platform, for instance PHP or C#. Integrating styling into the UPL means that some styling abstractions are added on top of the UPL.

This report will be concerned with creating a styling solution that is integrated into an UPL and supports the trend of RIAs. By creating such an expressive styling solution, it would allow web developers to create simple solutions to advanced styling problems.

In the following section, some of the features which can be used to create an expressive styling solution are outlined.

1.1 Wanted Features

The initial idea of this project is to create a more powerful styling solution than CSS for creating RIAs. What we would like from a powerful styling solution is described in the following list, by describing the features that are missing in CSS:

Advanced selection of elements: For instance, selecting elements with a particular element as child is not possible in CSS, nor is selecting the second last child of an element. It is also not possible to select an element based on the properties of that element. An example of this is selecting all elements that contain the text “Styling is fun” or have a specific color. The idea is to give more expressive power in selectors.

Relative styling: In CSS, it is not possible to set the color of an element based on the color of another arbitrary element.

Better integration with the UPL: Integration into the UPL will allow styles to be controlled dynamically. For instance, applying a style based on user interaction, and dynamic altering and creation of styles.

Arithmetic expressions: Setting the size of an element based on the combined size of two other elements is not possible in CSS, which makes CSS very weak. This feature would make styling radically more expressive. Allowing the use of variables and functions from the UPL in these styling expressions will yield an even higher degree of integration into the UPL.

If, and to which extent these functionalities are available in the existing styling solutions will be described later in the analysis.

1.2 What is Styling

In order to speak about styling, we need to define what styling really means, and what technicalities are concerned with styling.

Styling is about changing the visual appearance of something. There are mainly two ways to change the visual appearance of elements in a web page:

Element structure: Removing/inserting graphical elements. For instance, inserting a button or a text string.

Element appearance: Change the visual appearance of an existing element. For instance, changing the color or font size for an element.

A styling language is a language that either makes it possible to restructure graphical elements or specify properties of the visual appearance of elements.

Chapter 2

Analysis

In the first part of the analysis some existing styling solutions are examined to find which features they have. As CSS is the most used of these styling solutions, a more comprehensive examination of CSS will take place. Then, some use cases are presented and implemented in some of the existing styling solutions. Some wanted features for styling solutions were presented in section 1.1. The use case implementations are used to find whether or not these wanted features exist in the existing styling solutions and to which extent. At the end of the analysis, it is examined how a web browser renders a web page, and how the styling can be changed after the web page has been rendered.

The results of the analysis will be used later in the report, when our alternative approach to styling will be designed and implemented.

2.1 Styling Solutions

The availability of web development frameworks for creating RIAs (as of 2008) was examined in a previous report about Unified Multi-tier Web Development [1]. As this report also focus on RIAs, these web development frameworks will be examined further regarding what kind of styling abstractions they have. Likewise, a number of languages specifically intended for styling the web are examined to determine how they express styling. The styling abstractions of the web development frameworks together with the styling languages will collectively be called *styling solutions*. The web development frameworks are:

- Echo Web Framework [2]
- GOA WinForms [3]
- Google Web Toolkit [4]
- HOP [6]
- Links [11]
- Silverlight [9]
- Swift [10]
- Haxe [12]
- Curl [13]

The styling languages are:

- CSS[14][15]
- CCSS[16]
- DSSSL[17]
- XSLT[18]
- XSL-FO[18]
- PSL[19]
- Relational Grammar[20]
- SASS[21]
- JSSS[22]

We have examined each of the web development frameworks above, with the purpose of determining whether or not any of them make use of interesting styling features. Interesting styling features are features we are not already aware of. What we found was that only one of the web development frameworks have something out of the ordinary, namely Silverlight with its XAML language.

Goa WinForms, Google Web Toolkit, Swift, Haxe, Echo Web Framework, and Curl all make use of traditional and well known OOP-like ways of managing the visual user interface. By OOP-like we mean the mechanism of instantiating GUI elements, adding and removing them from other elements, as well as accessing and modifying properties by using dot notation. **Dot notation** refers to the act of accessing an object member of an object member in a recursive manner, usually by means of a "." (dot) in mainstream OOP languages.

HOP uses HSS (Hop CSS), which is basically CSS with some preprocessing capabilities, which, for instance, make it possible to define constants. Links does not has any styling capabilities itself, and rely entirely on CSS.

XAML uses a template approach, where a template describes the style of an element. Conceptually, the approach in XAML is much like the OOP way, where changing the prototype/template of an element is an equivalent way to style a large amount of elements. The major difference is that XAML is based on XML. XAML also interacts directly with C#, which could explain the similarities.

We have chosen not to look further into any of the web development frameworks, as none of them seem promising with regard to finding an alternative and more expressive styling solution for RIA development.

When it comes to the styling languages, we have chosen to take almost all of them into further examination. Only two of them will be omitted. These two are XSL-FO and Relational Grammar. XSL-FO is mostly concerned with typesetting similar to that of TeX, which is outside the scope of this project. Relational Grammar does however represent an unique idea, but the idea is not suitable for RIA development.

In the following sections, the chosen solutions will be introduced and described in terms of their origin, general features and market penetration. A code example will also be included to give a better feel for each solution.

2.1.1 CSS

There are currently three levels of Cascading Style Sheets. Level 1 [23], Level 2 Revision 1 [24], and Level 3 [25]. CSS 1 was released by the CSS Working Group in December 1996. The CSS Working Group is affiliated with the World Wide Web Consortium (W3C). W3C is an international consortium that maintains a large number of web related technologies (specifications, guidelines, software, and tools) with the purpose of advancing the web to reach its full potential[26].

CSS 2.1 is the successor of CSS 1 and currently the most used styling language for web development. The first working draft was made in November 1997 and released as a W3C Candidate Recommendation in July 2007 [24].

The initial idea behind CSS was to separate the styling and structuring of HTML documents. The fundamental idea in CSS is the *cascade*. The *cascade semantics* defines what styles to be applied to the document. The author as well as the user can both specify the styling of a document. At the same time the browser has a default style. The cascade allows these three styles to be combined to a final style, which will then be applied.

As noted by Philip M. Marden Jr. et al.[27], "*CSS has a very simple syntax, but limited expressive power. CSS has been explicitly designed to be written by non-programmers and often provides nice, intuitive ways to express style ideas.*". We fully agree with this observation.

CSS 3 will have more expressive power than the previous versions of CSS, and a great number of additional properties. CSS 3 consists of a number of modules, each concerned with specific areas of the styling. This modularization allows for more flexibility in updating the specification [25]. Although the work on CSS 3 begun in May 2001 [25], Bert Bos, chair of the CSS Working Group at W3C, stated in October 2007 that he believe the most CSS 3 modules will be implemented in four to five years (by 2011/2012) [28]. Many browsers have however already begun implementing some of the CSS 3 modules.

Since CSS is the most widely implemented solution and therefore used for styling of websites, a deeper look into CSS will be taken in section 2.2.

A CSS file consists of a number of rules, and each rule consists of a selector and a number of declarations. An example of CSS is shown in code example 2.1.

```
1  @import url("fineprint.css") print;
2
3  .someClass,
4  #someId,
5  *[href]
6  {
7    font-size: 12pt;
8    color: red;
9    padding: 10px 2em 5% auto;
10   margin 2em;
11 }
12
13 form input,
14 div > img,
15 p + p,
16 {
17   position: relative;
18   top: 20px;
19   width: 60px;
20 }
```

Code example 2.1: *CSS 2.1*

In the code example, line 1 includes another style sheet for use in situations where the content is to be styled in a page based manner, for example in print.

Line 3 to 11 define a rule, and so do line 13 to 20. Line 3 to 5 is called the selector, and consists of three patterns separated by commas. The commas mean that the rule matches on all of the three patterns. This selector matches all tags with the class attribute set to "someClass", the tag with the id "someId", and all tags that have the href property set. The second rule matches all "input" tags that are descendants of "form" tags, all "img" tags that are children of a "div" tag, and all "p" tags immediately following another "p" tag.

2.1.2 CCSS

This description of CCSS is based on an article of Greg J. Badros et al.[16]. Constraint Cascading Style Sheets was developed in 1999 and is an extension to CSS that introduces the concept of constraints. The problems with CSS that CCSS solves are:

- It is not possible for the designer to control the appearance of the document in various environments. For instance, styling based on window sizes.
- It is not possible to style an element relative to other elements than its parent.
- The CSS specification is complex, and it is difficult to understand how some features interact. For instance table layout features.
- Limited browser support for CSS 2. This is probably because of complexity and that a unifying implementation mechanism not has been suggested in the specification.

The main idea behind CCSS is to describe the visual appearance by using *linear arithmetic constraints*[29]. *"The extension allows the designer to add arbitrary linear arithmetic constraints to the style sheet to control features such as object placement, and finite-domain constraints to control features such as font properties."*

It is clear that the presence of these constraints make CCSS more expressive than CSS. These constraints can be both equalities and inequalities. CCSS also adds the ability to evaluate arithmetic, which makes it even more expressive and intuitive to work with than CSS.

A subset of CCSS has been implemented in the Amaya[30] v1.4a browser to demonstrate the expressiveness [16]. It has not been possible to obtain the prototype implementation.

An example of how to express some of the constraint features of CCSS can be seen in code example 2.2

```
1  /* <div class="thumbs">
2       <div><img /><p>some img</p></div>
3       <div><img /><p>some other img</p></div>
4       ...
5     </div> */
6
7  @precondition Browser[frame-width] < 800px;
8
9  @variable thumbs-width;
10 @constraint thumbs-width <= Browser[frame-width] - 70;
11
12 .thumbs { constraint: width = thumbs-width; }
13
14 .thumbs > div { float: left; }
15
16 .thumbs > div > img
17 {
18   constraint: width  <= (parent[parent])[width] / 3;
19   constraint: height <= (parent[parent])[width] / 3;
20 }
```

Code example 2.2: *CCSS*

The comment on line 1-5 contains the HTML structure to be styled by the CCSS. The `@precondition` rule on line 7 specifies the conditions under which the styles in this file will be used. According to the precondition, the style in the example will only be used if the canvas is less than 800 pixels wide. The variable `thumbs-width`, declared on line 9, can be used among constraints to link several properties together in arbitrary ways. It is possible to declare constraints among variables in order to be more expressive (by using `@constraint`), as done on line 10. This constraint causes the variable `thumbs-width` to always be less than

the canvas width subtracted by 70. On line 12, the variable `thumbs-width` is bound to the width of the `div` element with the class name `thumbs`. In the last rule, line 16-20, the code `(parent[parent])[width]` will take the width of the grand parent of the element, which will be the `.thumbs` element. The style sheet is meant to style a thumbnail page in a way that allows for three columns of thumbnails and a side panel of 70 pixels.

2.1.3 DSSSL

This section is primarily based on a draft of the specification of DSSSL [17].

The Document Style Semantics and Specification Language (DSSSL) is a language for formatting and transforming SGML (Standard Generalized Markup Language) documents, such as HTML and XML. In the 1960s the idea of separating the information of a document from the formatting of a document emerged. SGML was suitable for the role of representing the information. DSSSL was then designed to take on the role of formatting SGML, as SGML itself has no visual representation.

Fujitsu implemented a XML/SGML browser named HyBrick, which had support for DSSSL. This browser was demonstrated at the SGML conference in 1997[31]. HyBrick does not seem to be available anymore. An open source DSSSL processor called OpenJade is still obtainable, but according to the SVN repository, no significant changes have been committed since December 2005[32].

The most prominent use of DSSSL seems to be in DocBook[33], a document type definition for technical information. It is uncertain whether DSSSL is still used with DocBook. In general, there is little evidence that DSSSL is in widespread use today.

What makes DSSSL extremely expressive comes from the fact that it is derived from a general purpose programming language. DSSSL, unlike the other styling solutions, is Turing complete, which means that DSSSL actually can perform arbitrary computation. It is however also a very complex language. One of the things that make DSSSL extremely expressive is the possibility of declaring and calling functions.

DSSSL consists of several languages:

Expression language: The foundation for the three following languages. This language is a cut down version of the Scheme programming language. For instance, all the parts that introduce side effects have been removed, which makes it purely functional.

Standard Document Query Language: This language is for navigating through and selecting elements in the document that needs processing. This language adds two data types; `node-list` and `named-node-list` and some additional syntax for performing operations on the data types. Operations can not only be performed on the structural level but also on the content level, that is, parsing, generating, and transforming text. This language is part of the two following two languages.

Transformation language: This language is intended for transforming one SGML document into another SGML document. This is done by means of query expressions, transform expressions and priority expressions.

Styling language: The language for formatting a SGML document. The formatting process is defined by means of construction rules, which are described in this styling language. Construction rules map the elements of the document tree to formatting objects with visual representation. This language includes syntax for specifying ty-

pographic properties for elements. The styling allows for styling of both paged and continuous media.

The DSSSL language is very large and complex, and an explanation of how the different parts of the language interact is therefore beyond the scope of this report.

An example of the formatting capabilities in DSSSL can be seen in code example 2.3. This code example is based on examples from an introduction to DSSSL[34].

```
 1 (define heading-font "Times New Roman")
 2 (define heading-weight 'bold)
 3 (define document-font-size 10pt)
 4
 5 (element H1
 6   (make paragraph
 7     font-family-name: heading-font
 8     font-weight: heading-weight
 9     font-size: (* 2 document-font-size)
10     quadding: 'left ))
11
12 (element (p important) (if (equal? (attribute-string "visible") "true"))
13   (make paragraph
14     font-size: 12pt
15     (make sequence
16       font-weight: 'bold
17       (literal "!!!")
18       (process-children)
19       (literal "!!!"))))
20
21 (define (indent?)
22   (if (first-sibling?)
23     6pt
24     0pt))
25
26 (element p
27   (make paragraph
28     first-line-start-indent: indent?)
```

Code example 2.3: *Formatting capabilities of DSSSL.*

The first three lines define constants that can be used later. All lines starting with `(element` are beginnings of construction rule declarations. A DSSSL processor automatically applies these construction rules to all the elements they match.

On line 5, a construction rule is defined that matches all `H1` elements and maps paragraph formatting objects to them. Further, a number of properties are set on the paragraph. The properties `font-family-name` and `font-weight` are defined by the constants defined on line 1-2. The property `font-size` is defined by an expression that results in the value `20pt`. The property `quadding` is known as justification in word processors, and is set to left justification. The DSSSL processor automatically processes the sub elements of the `H1` element and inserts the formatting objects appropriately.

The construction rule on line 12 will only match `important` elements that are children of `p` elements, and that have an attribute named `visible` with the value `true`. Unlike the construction rule on line 5, the one on line 12 takes control of the processing of the children of the matched element. This is because it has been specified that the children must be inserted in a special location, by calling the procedure `(process-children)` on line 18. What otherwise happens is that a paragraph with a font size of 12 points will be created, which contains a sequence of bold elements. The first and last of these elements will constitute the literal string `!!!`. Between these two, the formatting objects of the processed children of the `important` element will be placed. It is possible to programmatically select what children to be processed as well.

A function named `indent?` is defined on line 21. This function tests if the current element is the first sibling in the document tree. If so, the function evaluates to the value

`6pt` otherwise `0pt`. The construction rule on line 26 calls this function for every `p` element, and sets the indentation of the first line of this text according to the value returned from the function.

Due to lack of suitable documentation, we have left out an example of the transformation capabilities of DSSSL.

2.1.4 XSLT

XSLT is an acronym for XSL Transformations, where XSL is an acronym for Extensible Stylesheet Language. So XSLT is a part of XSL and provides, as the name indicates, transformations to the language. XSL is a XML-based language that is developed and maintained by the World Wide Web Consortium (W3C). XSL is derived from DSSSL described in section 2.1.3, and has some of the same features as DSSSL [27] (XSL is also Turing complete as DSSSL). XSL is for XML what DSSSL is for SGML. Whereas CSS is specifically intended to style HTML documents, XSL is a more general styling language for all documents in XML format. Besides XSLT, XSL consists of the languages XPath (XML Path) for navigating in XML documents and XSL-FO (XSL Formatting Objects) to format XML documents. Two versions of XSLT exists (1.0 [35] and 2.0 [36]), which were published as recommendations by W3C in November 1999 and January 2007, respectively. The purpose of XSLT is to transform XML documents into other formats, often other XML document formats like XHTML documents. The transformation that should be applied is specified through *XML translation style sheets*.

A XML translation style sheet is declarative and built up of a set of template rules. These template rules are matched against and applied to a copy of the *source tree* (document that is being styled) to generate the *result tree* (the final document). Each template has a `match` attribute, in which a XPath expression defines which elements in the source tree the template match. A template can match a single or multiple elements. In each template, it is possible to use, among others, the following functionalities:

Conditional Processing: It is possible to use some of the most used control structures like `if` and `choose`. The `choose` control structure is a multiple choice with a final "default" part named `otherwise`. The conditions are specified as XPath expressions.

Repetition: If the template matches multiple elements, then the `for-each` construction can be used to iterate over these.

Sorting: If a template matches multiple elements, then the elements can be sorted before they are processed. It is possible to control the sorting order by specifying an XPath expression that is evaluated for each element, and is the base for the sorting.

Variables: Every result of an XPath expression or a tree of elements can be stored in a variable and used later.

Structural changes: When creating the result tree, elements and their attributes can be copied from the source tree, or new elements with new attributes can be created.

Functions: All the functions from XPath can be used in expressions. Examples of these functions are: `ceiling`, `count`, `string-length`, `substring`, `sum` and many other.

XSLT allows styling in form of changing the element structure, and does not provide any way to directly affect the element appearance. This can however be achieved through setting CSS styling attributes on elements or inserting CSS style blocks.

All major web browsers such as Internet Explorer, Firefox, Safari and Opera support XSLT. Implementations also exist in JavaScript, PHP, C, C++, .Net Framework, Java, and many others. Taking this into consideration, and the fact that the standard was revised in 2007, it is reasonable to conclude that XSLT is a language in wide use.

```
1 <?xml version="1.0" ?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns="
      http://www.w3.org/1999/xhtml">
3   <xsl:output method="xml" indent="yes" encoding="UTF-8"/>
4
5   <xsl:variable name="headerRepeatInterval" select="5"/>
6   <xsl:variable name="header">
7     <tr style="background-color: silver; font-size: 1.4em;">
8       <td>Full name</td>
9       <td>Affiliation</td>
10      <td>Mail</td>
11    </tr>
12  </xsl:variable>
13
14  <xsl:template match="/people">
15    <html>
16      <head></head>
17      <body style="font-size: 11px; font-family: verdana;">
18        <table style="border: 1px solid black; width: 400px;">
19          <xsl:copy-of select="$header" />
20          <xsl:for-each select="person">
21            <xsl:sort select="affiliation"/>
22            <xsl:sort select="fullname"/>
23            <xsl:sort data-type="number" select="string-length(mail)"/>
24
25            <xsl:if test="position() mod $headerRepeatInterval = 0">
26              <xsl:copy-of select="$header" />
27            </xsl:if>
28
29            <tr>
30              <td><xsl:value-of select="fullname" /></td>
31              <td><xsl:value-of select="affiliation" /></td>
32              <td><xsl:value-of select="mail" /></td>
33            </tr>
34          </xsl:for-each>
35        </table>
36      </body>
37    </html>
38  </xsl:template>
39 </xsl:stylesheet>
```

Code example 2.4: *XSLT*

An example of using XSLT is shown in code example 2.4. The XSLT code, source document tree, result document tree, and a screenshot of the final result is included in appendix A.1.1. In this example, a list of names of people in XML format is presented to the user in a HTML table. Line 1-3 define that the document contains XSLT data that is used to transform XML data to XHTML data. Two variables are declared on line 5-12. The first is `headerRepeatInterval`, which defines how often (in number of rows) the header of the table should be repeated in the table (useful for long tables). The second is `header`, which contains the head data in form of a `TR` element. Note that CSS is used to style this element. Line 14-38 define a template that match a list of people, which is the case if the XML document contains an element with the tag name "people" (`<people></people>`). The elements inside a template are inserted into the result tree, except the elements that belong to the `xsl` namespace. The tags on line 19-27 belong to the `xsl` namespace, while the tags on line 15-18 do not. The first element inserted in the table is the table header on line 19. On line 20-34, a for-each construction iterates through all the "person" elements inside the "people" element. Instead of the for-each construction, another template could have been created with a `match` value of "/person". The "person" elements are **sorted** on line 21-23 based on: 1) affiliation, 2) full name, and 3) length of mail address. Line 25-27 use a conditional statement and the XPath function **position** to test if it is time to insert

the table headline again. The data about the "person" elements are inserted into the table on line 29-33 by using the `xsl:value-of` construction.

2.1.5 PSL

The Presentation Specification Language (PSL) is a declarative styling language. The language was developed in 1995 as a research language by Mundo at the university of Wisconsin-Milwaukee[19]. PSL is not specifically intended for styling of web content but as a general styling language for structured documents as XML documents, for instance.

When PSL is used in connection with styling of web content, it can, according to Philip M. Marden Jr. et al. [27], be characterized as a "midpoint between CSS and XSL in complexity and power", which seems to be a reasonable characterization of the language. Some of the ideas behind the language are: simple syntax, easily described semantics, and considerable expressive power [27]. When styling a web page with PSL, it is possible to both change the structure and appearance of elements. The features of the language can be described in terms of three different aspects, which the PSL authors call presentation services:

Property propagation: It is possible to set properties on elements to control the appearance of them. Just like setting e.g. a color or border property in CSS.

Tree elaboration: New elements can be added to the presentation, and conditional statements can be used to control this.

Box layout: Gives the possibility to position elements on the screen in a different order than in the document. It also allows an element to use rendered positions or sizes of other elements to calculate its own. The Rendered position and size of an element is the position and size of the element when it has been rendered.

The knowledge available about the PSL language is rather limited, and only two articles that describe the languages are known. With this in mind, together with the fact that the articles date back from 1995, it is reasonable to postulate that the language has become extinct, and that it is not used in production today.

To give a short impression of the language, a code example of PSL is presented. The purpose of the code is, given a document with only a series of links (`a` elements), to place all links next to each other in a diagonal direction from top-left to bottom-right. At the same time, an image of a tick will be placed on the right side of each link, if the link has a non-empty `href` attribute. The example is shown in code example 2.5.

The code example begins with some headers that declare that, the media for the presentation is Mosaic (A web browser that supports PSL), the name of the presentation is "links" and the name of the document structure that is being presented is html (line 1-2). The most interesting sections of the code example are the `ELABORATIONS` and the `RULES` section. The `ELABORATIONS` section contains definitions of code for later use (line 4-8), while the `RULES` section contains styling rules that specify the visual appearance of elements (line 10-19). The only rule in this example matches all `A` tags (line 11). First, it makes the matched elements visible (line 12). Then, the horizontal position of the left edge of the matching element is set to be the horizontal position of the right edge of the left sibling (e.g. they are placed between each other) (line 13). The keyword `Actual` means that it should be the position of the element when it is rendered on the screen. The calculated position is also increased by 10. The vertical position is computed in a similar fashion (line 14). Lastly, it is checked if the `href` attribute of the matched element is empty (line 15), and if not, the

image of a tick is inserted to the left of the matched element (line 16). The tick is declared in the `ELABORATIONS` section (line 5-7).

```
1  MEDIUM mosaic;
2  PRESENTATION links FOR html;
3
4  ELABORATIONS {
5    tick : Markup ("<IMG src=tick.gif>") {
6      visible = Yes;
7    }
8  }
9
10 RULES {
11   A {
12     visible = Yes;
13     HorizPos: Left = LeftSib.Actual Right + 10;
14     VertPos: Top = LeftSib.Actual Bottom + 10;
15     if (getAttribute(self, "href") != "") then
16       createLeft (tick);
17     endif
18   }
19 }
```

Code example 2.5: *PSL*

2.1.6 SASS

SASS (Syntactically Awesome StyleSheets)[21] is an abstraction over CSS for use with Haml (XHTML Abstraction Markup Language)[37], which is an abstraction over HTML. HAML and SASS are intended for making it easy and pleasant to create XHTML templates and style sheets.

As SASS is compiled to CSS, it cannot do more advanced styling than CSS, so within this aspect the two languages are equally powerful. The syntax and semantic of SASS however make the language more elegant, and the functionalities of the preprocessor make the style sheet easy to manage.

Some of the most notable functionalities of SASS are:

Nested rules: Are used in the case where e.g. three `div` elements with different ids should be styled, and these are inside another `div` element. In CSS, you would normally write three rules, where the selectors would look something like: "`div#container div#someID`". In SASS, you can avoid to write `div#container` in each of the three rules, by nesting the three rules inside a selector of the outer `div` element.

Syntax: The syntax of SASS is simpler than CSS, in that you do not need to use a semicolon to end a statement, only a line break is necessary. And when setting a styling attribute you place the colon before the statement and not between the attribute and the value. This makes it easier to distinguish selectors and styling rules.

Constants: It is possible to use constants in the language. This is very helpful if for instance some colors from a design should be used multiply times through the style sheet.

Arithmetic: The constants can be used to perform basic arithmetic (+, −, ∗, /, %), also if the constants are colors or strings.

Attribute Namespaces: When setting styling attributes like `font-family`, `font-size` and `font-weight` in CSS you write "font" many times. In SASS, it is possible to declare that a group of styling attributes all belong to the font-namespace, so that you only have to write `family`, `size`, and `weight`.

Mixins: A group of CSS attributes and values can be defined and used throughout the style sheet. This saves time for the developer, and makes it easier to maintain the style sheet.

The development of the language is still in progress and new versions appear from time to time, the latest from mid-February 2009. A community of almost 1000 users exists, and about 160 posts are submitted on the forum each month (average for 2008)[38]. Based on this, Haml and SASS do not seem to be widely used in production today, even though a small community exists.

```
1  !color = #999999
2  !hover_effect = #202020
3
4  =highlighted-text
5    :font-size 2em
6    :color red
7
8  #menu
9    :background-color orange
10   :font-family verdana
11
12   .tabs
13     :border
14       :width 1px
15       :style solid
16       :color= !color
17       :bottom-style none
18     &:hover
19       +highlighted-text
20       :border-color= !color - !hover_effect
21
22     #searchfield
23       +highlighted-text
```

Code example 2.6: *SASS*

```
1  #menu {
2    background-color: orange;
3    font-family: verdana; }
4    #menu .tabs {
5      border-width: 1px;
6      border-style: solid;
7      border-color: #999999;
8      border-bottom-style: none; }
9      #menu .tabs:hover {
10       font-size: 2em;
11       color: red;
12       border-color: #797979; }
13     #menu .tabs #searchfield {
14       font-size: 2em;
15       color: red; }
```

Code example 2.7: *CSS code generated from SASS code*

An example of SASS code is shown in code example 2.6. The code example is about styling a menu that consists of tabs. First, two **constants** define some colors (line 1-2). On line 4-6, a **mixin** is defined that is used to style text. The rest of the code declares the styling of the element with the id "menu" and elements inside this element. On line 12 is an example of using **nested rules**. Here, all elements that have the class "tabs", and are inside the menu element, are selected. To style these elements, an **attribute namespace** is used (line 13). This namespace is automatically prepended to all the style attributes on line 14-17. Note that `!color` refers to the previously defined constant, and that the bottom border is removed to give the element the feeling of being a tab. On line 18-20, it is defined how the tabs should look when the cursor is over them. Again, **nested rules** simplify the selector. Here the styles in the **mixin** are included, and a new color is calculated by using

arithmetic on the two colors in the constants (a light gray color is changed to a darker one). The rest of the code (line 22-23) shows how the mixin is used to style another element again. The CSS code that is generated from code example 2.6 is shown in code example 2.7.

2.1.7 JSSS

JavaScript-Based Style Sheets (JSSS) is a styling language for web content that was in development around the middle of 1996 by the Netscape Communications Corporation. The language is also known as JavaScript Style Sheets, JavaScript Accessible Style Sheets (JASS), JavaScript Style Sheets (JSS) and Style Sheets with JavaScript Syntax [39]. In October 1996, Netscape proposed the language as a standard to the World Wide Web Consortium (W3C), but it was never accepted as a standard [22]. Later that year, the alternative CSS 1 (used in Internet Explorer) was instead accepted as a recommended standard. No official documentation says why CSS was picked in favor of JSSS, but the submitted standard of CSS seemed to be more complete than that of JSSS [40]. After CSS was recommended by the W3C, Netscape stopped the development of JSSS, and started to support CSS in their browsers. This means that support for JSSS is only available in Netscape's browser Netscape Communicator version 4.x.

The JSSS styling language is a good example of what possibilities can be achieved by including the client side programming language into the styling solution. In many aspects, JSSS is similar to CSS:

Selectors: It is possible to select one or multiple elements in a selector. In a selector, tag names, class attributes, and id attributes can be used to select elements. Pseudo classes like `firstLetter` or `firstLine` can also be used. Selectors can express that a specific element should only be selected if contained inside another specific element.

Style rules: The appearance of elements that match a selector is set by means of setting styling properties. If the appearance of an element is defined twice, priority rules are used to decide which appearance to use. Styles can be specified in an external file, as a block of styling rules, or as a style attribute directly on an element.

JSSS however has the following extra functionalities:

Arithmetic, conditions and function calls: As JSSS is implemented in JavaScript, it is possible to use all the functionalities of JavaScript. So arithmetic and function calls can be used to calculate values for the styling expressions, and conditional statements can be used to further control e.g. which style should be applied.

Delegates: A JavaScript function can be assigned to a selection (using the `apply` property). Later, when an element that matches the selector is loaded, the delegate function is executed in the context of that element. This function can then be used to style the element of concern.

Environment information: It is possible to get information about the environment in which the document is shown. This includes information like the size of the canvas of the browser, or the number of colors available on the user's display. Information about the browser type and version can be used together with conditional statements to create browser specific styling.

Based on the fact that JSSS only works in Netscape 4.x, and that Netscape 4.x only has a market share of under 0.02%, it is reasonable to conclude that JSSS not is used in production today [41].

An example of using JSSS is shown in code example 2.8. In this example, the function `table_stripes` is applied to all rows in tables that have the class attribute set to `stripes` (line 16). This function is executed each time one of these rows appears in the document. The function `table_stripes` uses some environment information to decide whether the browser (or user's display) supports more or less than 16 colors (line 2). Based on this, and the current color applied to table rows (line 3 & 9), the function change the color to use for future rows (line 4-6 & 10-12). The final result will be that each table will have stripes, by means of rows in different colors.

```
 1 table_stripes() {
 2   if (visual.colorDepth > 16) {
 3     if (color == "MediumBlue")
 4       color = "MediumBlue";
 5     else
 6       color = "Gold";
 7   }
 8   else {
 9     if (color == "Blue")
10       color = "Blue";
11     else
12       color = "Yellow";
13   }
14 }
15
16 contextual(classes.stripes.TABLE, tags.TR).apply = table_stripes;
```

Code example 2.8: *JSSS*

2.2 Cascading Style Sheets

This section gives an overview of CSS 2.1 and CSS 3.

CSS is the fundamental styling mechanism in browsers today. Previously, styling was done by means of designated tags. Tags such as `<TT>`, `<I>`, `<B>`, `<BIG>`, `<SMALL>`, `<STRIKE>`, `<S>`, `<U>`, `<FONT>`, and `<BASEFONT>` are all for styling of text. Some aspects of the styling were done through attributes on tags. For instance the `align` attribute, which was used to align text and float objects [42]. CSS was made with the purpose of separating the structure of documents from the styling, in order to prevent the invention of new incompatible styling tags.

There are three ways to apply CSS styles to a document.

Inline: CSS makes it possible to set CSS properties of an element in a style attribute of the element. For example:
`<p style="text-align:  justify;">some text</p>`
This will cause the text to be justified.

Embedded: CSS can also be specified in a style tag inside the header of a HTML document. For example:
`<style type="text/css">p { text-align:  justify; }</style>`

External: The last option is to link the document to an external style sheet by inserting a link tag into the header of the HTML document like this:

```
<link rel="stylesheet" type="text/css" href="fancy.css" title="Fancy" />
```

The advantage with external style sheets is that the same style sheet can easily be applied to several documents.

None of the three ways of styling excludes the others. So it is possible to use inline, embedded, and external style sheets at the same time. If there are some conflicting styles, the inline styles will have highest precedence, while the external styles will have lowest precedence.

Syntax

The basic building blocks of CSS code are called rules. There are two kinds of rules: *at-rules* and *rulesets*. At-rules are rules that start with an at-sign ("@"), and rulesets are rules that declare values of properties of selected tags.

There is only one notable at-rule, and that is the `@import` rule. This rule is used for importing rules of an external style sheet, which makes it possible to combine several styles. Imported rules will have lower priority than the rules in the document they were imported in. In that way it is possible to override imported rules.

A ruleset consists of a selector and declarations of properties. The selector results in a number of elements. The declaration of the properties applies to each of the selected elements.

An example of a simple CSS ruleset can be seen in code example 2.9.

```
1 p
2 {
3   font-size: 14pt;
4 }
```

Code example 2.9: *Example of a ruleset in CSS.*

In the code example, the `p` is the selector, which is followed by a declaration block. In the declaration block, the property `font-size` is declared to have the value `14pt`. This structure is the same for all CSS rulesets. It is however possible to specify more complex selectors and declare more properties than in the simple example.

In the following sections, the different kinds of selectors and declarations are presented.

Selectors

A selector is a kind of expression used for selecting elements in a document.

In CSS 2.1 the following selection operators are available:

$*$ Matches all elements. This universal selector can be omitted.

E Matches all *E* elements.

E F Matches all *F* elements that are descendants of an element matched by *E*.

E>F Matches all *F* elements that are children of an element matched by *E*.

E+F Matches all *F* elements immediately preceded by a sibling element matched by *E*.

E,F Matches all elements matched by *F* and all elements matched by *E*.

E.identifier Matches all elements matched by *E* that are members of the class *identifier*. This is just a shorthand notation for *E[class∼="identifier"]*.

E#identifier Matches all elements matched by *E* that have the id attribute set to *identifier*. This is just a shorthand notation for *E[id="identifier"]*.

E[attr] Matches all elements matched by *E* that have the attribute *attr* set.

E[attr="value"] Matches all elements matched by *E* that have the attribute *attr* set to *value*.

E[attr~="value"] Matches all elements matched by *E* and where the space separated attribute *attr* contains the token *value*. A space separated attribute is for instance the class attribute when it contains multiple class names, e.g. `class="text important headline"`.

E[attr|="value"] Same as above, but hyphen separated instead of space separated.

E:pseudo Matches all elements matched by *E* that are members of the pseudo class/element *pseudo*.

There are a number of pseudo classes and pseudo elements that represent different concepts. Some pseudo classes such as `:hover`, `:active`, and `:focus` rely on user interaction, whereas other pseudo classes such as `:first-child` and `:lang` do not.

The pseudo classes make it possible to style links differently based on whether or not they have been visited, as well as if they are currently being visited. The `:hover` pseudo class makes it possible to e.g. create menus that expand when hovered, as well as makes it easy to highlight elements based on interaction. When filling forms and otherwise interacting with input elements, the :active pseudo class allows these elements to be styled differently when active. These pseudo classes are said to be dynamic because they reflect user interaction.

Pseudo elements allow for styling of the first letter or first line of a paragraph, as seen in some papers and books. There are also pseudo elements such as `:before` and `:after`, which can be used for inserting content to be rendered before and after an element in the document.

For instance, the author of a web page might want to alter the style of all links inside a `div` element with the id `content`. The selector needed to select all these links is simply `div#content a`. If the author wants to style the currently active element, regardless of the type of that element, the selector would look like `*:active`.

CSS 3

In addition to the selection features of CSS 2.1, CSS 3 introduces some new and more expressive selection features [43].

Three of these are just attribute selectors for making it possible to select based on a substring of the value of an attribute (begins with, contains, ends with). The `E` $\sim$ `F` structural selector is more generally useful, and matches all *F* elements preceded by a sibling element matched by *E*. So `img` $\sim$ `p` will select all `p` elements that have an `img` sibling, where the `img` sibling is lexically before the `p` element.

A number of new pseudo classes will likewise be available, which makes CSS 3 substantially more expressive regarding selection of elements. Some of the more interesting pseudo classes that make new kind of selections possible are described in the following list.

:root Matches the root element of the document.

:nth-child(n) Matches the n-th child. The argument can be expressed as $an + b$. For instance `div:nth-child(4n+1)`. What will happen is that all the children of the `div` element will be split into groups of 4 elements, and then every second element will be selected. Two keywords can be used as argument as well: odd and even. `:nth-child(odd)` is the same as `:nth-child(2n+1)`, and `:nth-child(even)` is the same as `:nth-child(2n)`. What this really selects are all the children that have an+b-1 siblings after it, for a given positive (including zero) value of n. What this means is that this pseudo class can be used for selecting the first 3 elements by `:nth-child(-n+3)`.

:nth-last-child(n) Matches the n-th last child. This works the same way as `:nth-child(n)`, except `:nth-last-child(n)` is selecting the children that have an+b-1 siblings after it, for a given positive (including zero) value of n.

:nth-of-type(n) Matches the n-th sibling with the same element name. Matches in a similar way as `:nth-child(n)`, but completely ignores the elements of types other than the reference element.

:empty Matches empty elements. An element is empty if it contains no content. That is neither sub elements nor text.

:last-child Matches elements that are the last child, similar to but opposite of `:first-child`.

:not(S) This pseudo class negates another pseudo class. For instance `h1:not(:first-child)` will select all `h1` elements, except the ones that are first child.

The `:not(S)` pseudo class in itself is a rather big advancement with regard to expressiveness. The new structural pseudo classes, of which only the most basic have been listed above, are also a significant advancement in expressiveness. It would however have been substantially more expressive if it was possible to specify arbitrary conditions.

Declarations

In CSS 2.1 there are a number properties. The major groups of properties are:

Typography: Text type, size, alignment, and style. As well as table and list formatting.

Screen reader: Voice pitch, volume, and speed.

Box layout: Positions, sizes, paddings, and margins.

Graphics: Backgrounds, borders, and cursors.

Setting these properties is very simple. Code example 2.9 sets the property `font-size` on line 3, which is the only way to set a property. Some properties are shorthand notations for several properties. It is, for instance, possible to set `border-width: 1px;`, `border-color: red;`, and `border-style: dashed;` just by setting `border: 1px red dashed;`.

Some of the properties, in particular those that refer to typography, are special. What make them special is that they are inherited by the descendants of the element where the property actually is specified. For instance, if element A has a child B and B has a child C. Then all of these three elements will have their font type changed if A's font type is

changed. It is however possible to explicitly set the font type of B, which will then affect B and C but not A.

In CSS there are also relative values, such as percentages, em, and ex. For instance, consider the declaration `width: 50%;`. What this declaration means is that the width of the element will be set to 50% of the width of its parent. For other properties, percentages may be relative to other properties on the same element, rather than the same property on the parent element. In the case of em and ex, the rules are simpler. One em is always the same as the text size of the property. For instance, if the `font-size` was set to `12px`, then `2em` will correspond to `24px`. The ex unit is similar to the em unit, but is relative to the `x-height` property rather than the `font-size` property.

There are a number of other units, all fixed size. One of these is px, which refers to pixels. Other units refer to other measures than length, which are irrelevant for the purpose of this report.

In CSS 3, a lot of new properties will be available. But as all of them are concerned with the presentation of elements and not new ways to specify the styling in general, they will not be covered here. For instance, CSS 3 has new properties for specifying custom borders, but these properties are set in the same manner as all other properties.

But CSS 3 does supply a new way of specifying properties, namely with the help of the `calc` function [44]. This function makes it possible to express lengths by means of arithmetic operators. An example of the use of the `calc` function looks like this: `height: calc(5% * 4em + 200px);` . It is however not possible to declare variables nor constants in CSS 3, so only constant numeric values can be specified as arguments to the `calc` function. While CSS 3 has many other kinds of units like angle, time, and frequency, none of these units are valid inputs for the `calc` function.

Cascading Rules

One of the fundamental parts of CSS is the cascade, which will now be described. Styles can be specified by the author of the web site, the user and the web browser. As mentioned earlier, the author can specify the styles in three ways: As style attributes on the elements, as a block of style declarations in a style tag, or as a link to an external style sheet. Furthermore, it is possible for the user and author to declare styles as important. Important styles declared by the user or author will override the styles of the other parts if the priority of the selectors are the same.

To compute the style for an element, all the styling declarations that match the element and the properties of the element are found. The priorities of the found styling declarations are then given by the following list, where the value of the number designates the priority (1 is the lowest and 5 is the highest):

1. Browser defined

2. User defined

3. Author defined

4. Author defined (important)

5. User defined (important)

If two styling declarations for the same element and CSS property have the same priority, then they are prioritized by the specificity of the selector. The specificity is calculated by taking the following characteristic into consideration:

- Is the style defined in a style attribute.
- How many times are the following used in the selector:
  - IDs
  - Attributes and pseudo-classes
  - Element names and pseudo-elements

If the priorities of two styling declarations still are the same, the last declared styling declaration will be used.

2.3 Use Cases

In this section, we put forward a number of use cases we have created that reflects real life styling challenges. These use cases will later be implemented in some of the styling solutions described in the previous sections. In the chapter 7, these use cases will be used again to evaluate our solution.

When the use cases have been implemented, they will be compared with regards to some styling aspects, to estimate the relative expressiveness of the styling solutions.

While most of the use cases are very specific and concerned with layout and boxes, the underlying concepts can be generalized to other use cases that are concerned with other styling aspects. These use cases are however less apparent and harder to illustrate, so they are not part of the use cases in this report. For instance, these use cases could be concerned with setting fonts, colors, borders, margins or other CSS properties.

The use cases will be explained in further details in the following sections. Illustrations of each of the use cases are shown in figure 2.1. Each use case has a name, and the letter in parentheses at the end of the name indicates which illustration in figure 2.1 that illustrates the use case.

Multi Level Menu (A)

When making multi leveled menus, the menu items containing sub menus often have to be styled differently than the menu items not containing sub menus. This is to give visual feedback that some menu items contain sub menus.

The items should be styled according to whether or not they have a specific child that contains sub items. An illustration of this can be seen in figure 2.1 (A). The blue box is an ordinary menu item among the white boxes, the only difference is that the blue item contains sub items (red boxes). The blue box is the one to be styled specially to make it apparent that it contains sub items.

Submenu Positioning (B)

One of the common use cases in RIA development is drop-down menus. Some of these menus contain submenus. This is usually styled in a specific manner as illustrated in figure 2.1 (B).

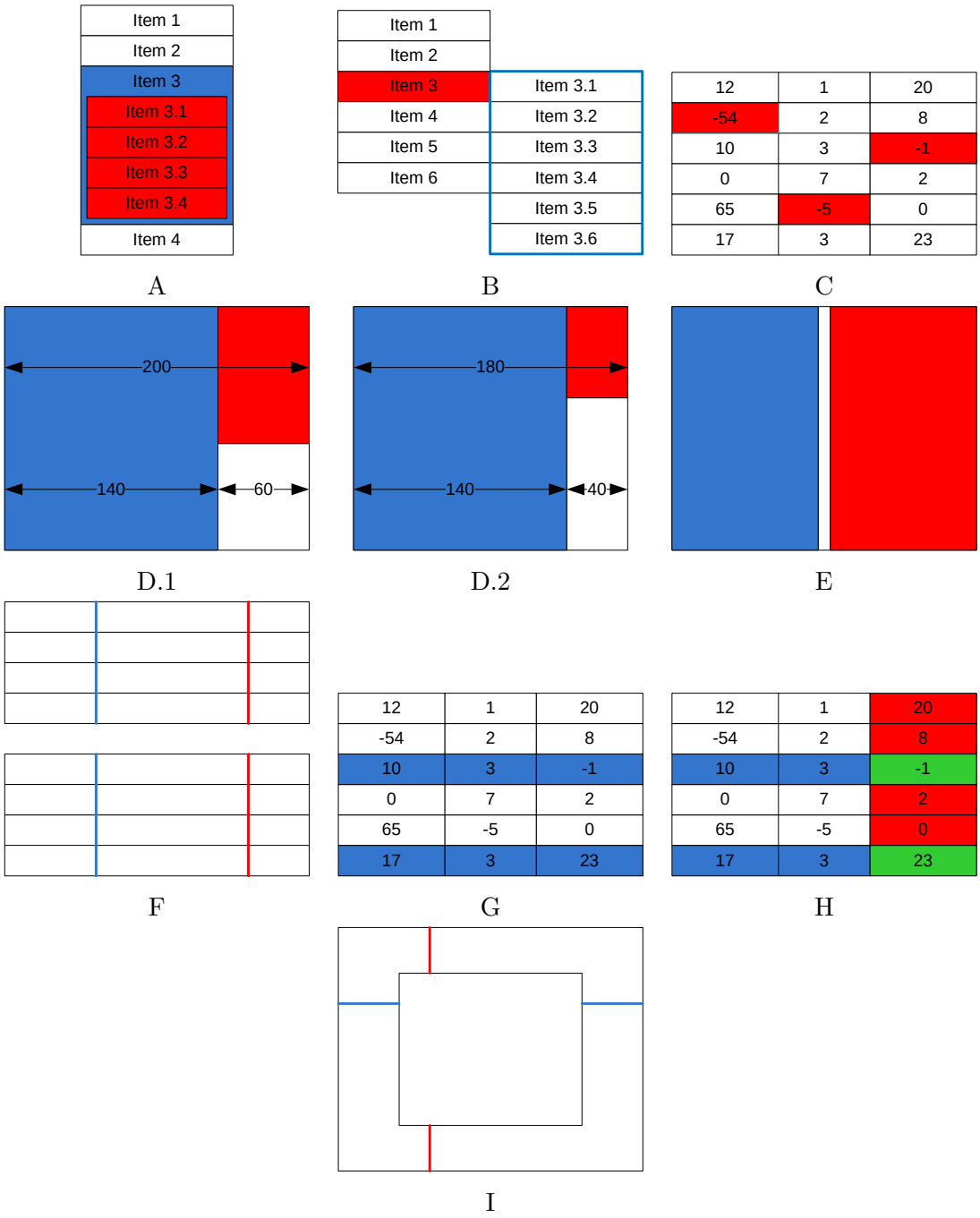


Figure 2.1: *Illustrations of use cases*

The menu containing the red rectangle in figure 2.1 (B) is the parent menu, and the one containing the blue rectangle is the submenu. When opening a submenu, this is usually aligned with the item it originated from. So the red and the blue box are horizontally aligned.

Cell Content (C)

When working with numbers in tables, it is sometimes desirable to style the cells of the tables according to the actual value of the numbers in the cells. An example of this is illustrated in figure 2.1 (C), where all negative numbers are highlighted. Or generally, when having a lot of data, it is sometimes desirable to style that data according to what the data actually mean.

The idea is that elements should be styleable according to the actual content. For instance, all cells in a data grid should be styled in a way that highlights the important data.

Aspect Maintenance (D)

When making dynamic layouts, some elements are less important than others, and these elements are the ones that shrink when the user decides to reduce the size of the browser window. The element that should shrink could be an image.

The idea behind this is illustrated in figure 2.1 (D.1) and (D.2). The blue rectangle is a fixed width container, and the red rectangle is the element to be scaled. The size of the black box can be dynamically changed by the user. When resizing an image, it is usually desirable to maintain the aspect ratio of the image. So when the width of the black box is reduced, the image should shrink while retaining its aspect as seen in figure 2.1 (D.2).

Column Layout (E)

A common use case is to have multiple columns, of which one is for the main content, and other columns are designated to secondary information and site navigation. When web browsers renders content, it automatically extends the height of containers. This causes two containers with different content to have different heights. It is not uncommon that designers want these containers to be of equal height.

This is visualized in figure 2.1 (E), where the red and blue rectangles are two different containers with different amount of content. Further, the footer should be placed immediately under both columns.

Table Alignment (F)

When having many tables that contain largely the same kind of information, it is sometimes desirable to align the columns of these tables for convenience.

Figure 2.1 (F) illustrates this with two horizontally aligned tables, with aligned columns. The blue lines are aligned as well as the red lines. These lines are borders of the columns of the two tables.

Ordinal Selection (G)

When having tables of data with many columns, it is common to style every other row in a different manner, in order to make it easy to determine what data is related. Having an offset for what row to start with, as well as a row to end with, would be ideal for flexibility. This is visualized in figure 2.1 (G), where the blue boxes represent every 3th row in a table.

Selection Intersection (H)

It is not unlikely that one wants to apply styles to specific columns as well as rows at the same time. In addition to that, one might want to style the cross selected styles in an even more specialized manner, to highlight them even further. A representation of this can be seen in figure 2.1 (H). The blue cells are every third row, and the red cells are every third column. The green cells are the cross selected cells. These cells are the intersection of the selection of the red and the blue cells.

Centering (I)

Centering of elements is a common way to draw attention to something, but it is also used for general layout purposes. As seen in figure 2.1 (I), the blue box is completely centered within the black box. Each of the red lines has the same length.

2.4 Use Case Implementations

In section 2.3, a number of use cases were presented. In this section, some of these use cases will be implemented in some of the styling solutions described in section 2.1. The implementations will then be compared to each other on basis of groups of features. For each group, only the relevant implementations will be included. Implementing the use cases in existing styling solutions gives insight into the current situation within styling. This knowledge is used to conclude whether or not there are any needs for further research within the area. This includes especially if, or how well, the wanted features mentioned in section 1.1 are supported.

Table 2.1 describes the capabilities of the styling solutions in terms of our use cases. The table also describes which styling solutions will be used to implement which use cases. There is one designation for each possible pair of use case and styling solution. The legend for these designations can be found right under the table. The designations indicate the possibility of implementing the use case in the styling solution.

Whether or not something is cumbersome is based on our research into the solutions and is our subjective opinion. Cumbersome also covers the cases where the styling solutions are capable of implementing the use case, but in a way we find unrelated to the point of the use case.

In the cases where something has been marked as unknown, we have been unable to find research material, or deduce with certainty, whether or not the solution has the necessary capabilities.

We have chosen which implementations to make based on which implementation would yield the most diverse use of features, compared to the other styling solutions in the table.

The use cases generally represent three groups of features. Each of these groups is given a name that will be used for later references. A classification of what use cases that covers which groups is listed here:

Element selection: Cell Content, Ordinal Selection, Selection Intersection, Multi Level Menu.

Interelement constraints: Submenu positioning, Column Layout, Table Alignment, Centering.

Expressions: Aspect Maintenance.

Use cases	CSS 2	CSS 3	CCSS	DSSSL	XSLT	PSL	SASS	JSSS
Cell Content	-	-	-	+	**+**	?	-	(+)
Ordinal Selection	-	**+**	-	+	+	+	-	+
Selection Intersection	-	-	-	+	(**+**)	-	-	-
Multi Level Menu	-	-	-	+	+	**+**	-	(+)
Submenu Positioning	+	+	+	?	+	+	**+**	(+)
Column Layout	-	-	+	?	-	+	-	(+)
Table Alignment	-	-	**+**	?	-	-	-	(+)
Centering	(+)	(+)	+	+	(+)	**+**	(+)	(+)
Aspect Maintenance	-	-	**+**	-	-	+	-	(+)

+	Possible
(+)	Possible but cumbersome
+	Implemented
(**+**)	Implemented but cumbersome
-	Not possible
?	Unknown

Table 2.1: *Relations between use cases and styling solutions.*

When evaluating the styling solutions, it will be done with respect to the three groups of features to divide the evaluation into smaller and more manageable parts. For each group, it will first be described in more details what the group covers in terms of features. The rest of the section will then largely be based on the implementation of the use cases that cover those groups.

2.4.1 Element Selection

All of the styling solutions use some functionality to select the elements that should be styled. The powerfulness of these functionalities are however very different. In the following sections, element selection will be split into some features and described within these. The features are: Content based selection, context based selection, predicates in selectors and set operations on selectors.

Content based selection

All the solutions have the possibility to select elements from the source document based on some attributes like id, class name, tag name or other attributes. There are some solutions that take the selection abilities a step further and allow selecting elements based on the content of the element. This is for instance the case in XSLT, where it is possible to test the content of an element using a `choose` control structure, and then execute different code based on the test. An example of such is shown in code example 2.10, which is a part of the cell content use case in XSLT. This part of the code takes care of creating rows in a table where some people are listed by name, together with their income and expenses. The idea is that the income and expenses should be summed up, and the color of this number should be red if the number is negative and black if positive. The most interesting line in the code is line 26, in which the content from the income and expense fields are used in a test expression. The complete code for the use case including source document tree, result document tree and a screenshot of the final result is included in appendix A.1.2.

```
18          <xsl:for-each select="person">
19            <xsl:sort select="fullname"/>
20            <tr>
21              <td><xsl:value-of select="fullname" /></td>
22              <td><xsl:value-of select="income" /></td>
23              <td><xsl:value-of select="expense" /></td>
24
25              <xsl:choose>
26                <xsl:when test="income - expense &lt; 0">
27                  <td style="color: red;">
28                    <xsl:value-of select="income - expense" />
29                  </td>
30                </xsl:when>
31                <xsl:otherwise>
32                  <td>
33                    <xsl:value-of select="income - expense" />
34                  </td>
35                </xsl:otherwise>
36              </xsl:choose>
37            </tr>
38          </xsl:for-each>
```

Code example 2.10: *XSLT implementation of Cell Content use case.*

The ability to select elements based on content is, for instance, good to have in the use case with income and expenses. Traditionally, such styling cannot be performed without the influence of an "advanced" server side or client side scripting language. But the most natural place to define this functionality is in the style sheet, since the functionality does not have any other purpose but to style the document.

DSSSL has the possibility to check content constraints in its selectors by using predicates. In JavaScript, DOM functionalities exist to select elements and their content, so it is possible in JSSS to test the content of an element and base the style on this. As described in section 2.1.7, it is possible to create a delegate in JSSS, and then assigning the delegate to a selector. As the delegate is executed in the context of the element that it matches, the delegate has a reference to the matching element, and can use this and the DOM functionalities to access the content of the element.

```
1  MEDIUM mosaic;
2  PRESENTATION MultiLevelMenu FOR html;
3
4  ELABORATIONS {
5    expand : Markup ("<IMG src=expandIcon.gif>") {
6    }
7  }
8
9  RULES {
10   LI {
11     if (NumChildren(Self) > 0) then
12       createLeft (expand);
13     endif
14   }
15 }
```

Code example 2.11: *PSL implementation of Multi Level Menu.*

Common for DSSSL and JSSS is that checking the content of an element is not an integrated part of the selector, which would be more natural. Checking the content should look something like the way CSS select elements based on attributes, which is integrated directly into the selector (`E[attr="value"]`). Instead of specifying an attribute in this example, it should be possible to use some keyword (`content`), to specify that the constraint is about the content of the element. It is not possible to perform content based selection in CSS, CCSS and SASS. It is uncertain to which degree it is possible in PSL to perform content based selection, as no documentation could be found that states this. But at least it is possible to test some aspects of the content, for instance how many elements that are

present in the content. An example of this is shown in code example 2.11 on line 10-11. Here, the styling of `LI` elements (menu items) is based upon if the `LI` element contains other elements (sub menus).

Context based selection

One feature that is close to that of content based selection is context based selection. Whereas content based selection is about what is inside a specific element, context based selection is, for instance, about what the specific element is inside or besides. One example of context based selection is the use case ordinal selection. This use case has been implemented in CSS 3, and is shown in code example 2.12. The code styles all table rows that are every 3th row in the table to which they belong. The text in these table rows will be colored red.

```
1 tr:nth-child(3n)
2 {
3   color: red;
4 }
```

Code example 2.12: *CSS 3 implementation of Ordinal Selection use case.*

As seen in the example, it is very easy to do this because the language provides a construct for exactly this purpose.

In CSS, and the other solutions that are based on CSS (SASS and CCSS), it is possible to perform some contextual based selections. This includes selecting elements only if they are descendants or children of some other elements, or if they have some elements as a left sibling. The contextual selectors are created by using special selector operators between selectors. These selector operators were described in more details in section 2.2.

In JSSS, it is possible to select elements by specifying the descendant relation between two or more selectors. This is achieved by calling the native JS function `contextual` with two or more selectors as parameters. This function will then return the elements that match the last selector and are descendants of the elements that match previous selectors. An example of this is shown in the description of JSSS in code example 2.8 (line 16).

DSSSL and XSLT can select elements based on ancestor, descendant, child, parent, sibling and other relationships. For instance, in XSLT (using XPath) the selector `//img[parent::div]` selects all `img` elements that have a `div` element as parent. The selector `//img[following-sibling::div]` selects all `img` elements that are siblings to a `div` element, and are placed before the `div` element in the code. But DSSSL is more powerful in contrast to the other styling solutions, as it is possible for a developer to define his own relationships. These defined relationships can then be used in the selectors in the same way as the built-in relationship operators. PSL is the only of the styling solutions that does not support contextual selectors.

Predicates in selectors

A styling solution would be very powerful if it was possible for a developer to define his own arbitrary predicates for use in the selectors. In that way, the developer would not be limited to the predefined predicates, which are described in the two previously sections about content and context based selection.

It is however not many of the styling solutions that support user-defined predicates directly, only DSSSL and XSLT (through XPath) do. An example of a user-defined predicate

in DSSSL is shown in the description of DSSSL in code example 2.3 (line 12). In XSLT, the predicate is simply written in square braces after the selector that selects the elements that should be checked.

The styling languages PSL and JSSS do not support user-defined predicates directly, but it is possible to achieve the same effect in these solutions. This is done by creating a selector that matches the elements that should be filtered by the predicate. In the body to this selector (the place where the styling declarations are), the code that would have been used in the predicate is written.

CSS, CCSS and SASS do not support user-defined predicates.

Set operators in selectors

In some of the styling solutions, it is possibly to create a selection by performing set operations on selectors. The only solutions that do not have this feature are PSL and JSSS. In CSS, CCSS and SASS, it is only possibly to select elements by using the union operator (`,`) between two other selectors. DSSSL seems to support many set operations like union, intersection and complement, but we have not been able to completely verify this. In XSLT, it is not possibly to perform e.g. intersection operations, but it is possibly to simulate it. An example of this is shown in code example 2.13, which is an example of how to style all table cells, which are in an even numbered row and column. The complete code for the use case including source document tree, result document tree and a screenshot of the final result is included in appendix A.1.3.

```
11          <xsl:for-each select="tr">
12            <tr>
13              <xsl:choose>
14                <xsl:when test="position() mod 2  = 0">
15                  <xsl:for-each select="td">
16                    <xsl:choose>
17                      <xsl:when test="position() mod 2 = 0">
18                        <td style="background-color: silver;">
19                          <xsl:value-of select="."/>
20                        </td>
21                      </xsl:when>
22                      <xsl:otherwise>
23                        <td>
24                          <xsl:value-of select="."/>
25                        </td>
26                      </xsl:otherwise>
27                    </xsl:choose>
28                  </xsl:for-each>
29                </xsl:when>
30                <xsl:otherwise>
31                  <xsl:for-each select="td">
32                    <td>
33                      <xsl:value-of select="."/>
34                    </td>
35                  </xsl:for-each>
36                </xsl:otherwise>
37              </xsl:choose>
38            </tr>
39          </xsl:for-each>
```

Code example 2.13: *XSLT implementation of Selection Intersection use case.*

First, all `tr` elements are selected and iterated through (line 11), as we need to check a constraint on them. The constraint is if the row is an even (line 14) or odd (line 30) numbered row in the table. If the row is an odd numbered row, we do nothing else but inserting the row into the result tree (line 31-36). If the row is an even numbered row, then the individual cells of that row are iterated through (line 15) and tested for being in an even numbered column (line 17). If this is the case, then a style attribute is applied to the cell (line 18).

2.4.2 Interelement Constraints and Expressions

Styling a given element in relation to another element is one of the core concepts in many of the styling solutions. Simple inheritance for instance, is present in all the styling solutions in the case of font size and font type. It is interesting to determine if and how the solutions handle more advanced relations. These advanced relations cover the concept of getting access to properties of other elements, as well as using those values to set properties. Being able to use those values in arithmetic expressions, functions and conditionals are closely related, and will likewise be described in the following sections. As noted earlier in the beginning of section 2.3, even though the use cases are highly concerned with positioning and sizing of boxes, we are concerned with the general idea behind the use cases.

We have identified a number of features related to this category, these features are: variables, constraints, expressions, arithmetic, conditional statements and functions, relative selection, reading property values, and accessing attributes. In the following, these features will be explained and pointed out in code examples.

Variables and Constraints

Both the SASS and the CCSS implementations have variables. Actually SASS only has constants, but these make it very easy to make style sheets that can be easily changed. Code example 2.14 shows that the variables declared on line 1 and 2 are used on the lines: 10, 12, 22 and 23. If CSS was used, all of these lines would have used literal values rather than constants, and would have had to be redefined if the width of the menu was to be changed. The names of the constants also help in understanding why certain properties need to have a certain value.

```
1  !menu_width = 100px
2  !border_width = 1px
3
4  ul
5    :padding 0
6
7  .menu li
8    :border
9      :color black
10     :width= !border_width
11     :style solid
12   :width= !menu_width
13   :list-style none
14   :margin 0
15   :position relative
16
17   > ul
18     :display none
19   &:hover > ul
20     :display block
21     :position absolute
22     :top= -!border_width
23     :left= !menu_width + !border_width
```

Code example 2.14: *SASS implementation of Submenu Positioning use case.*

In CCSS, the variables are not necessarily bound to a constant value. As can be seen in code example 2.15, there are three variables declared on line 1-3. Each of these variables will be constrained to the width of different `td` elements, as declared on line 7, 12, and 17. On line 5 in the code example, a number of `td` elements with their class set to "first" are selected. All the widths of these `td` elements will be constrained to the same variable, as seen on line 7, and therefore they will all be set to the same width. The evaluation of all properties and variables will be done dynamically, since they are actually constraints which will be enforced at any given time.

What goes for the other solutions, it can be mentioned that DSSSL and XSLT have constants, while JSSS has mutable variables. In JSSS, the variables need to be assigned values explicitly, unlike CCSS.

```
1 @variable first_col;
2 @variable second_col;
3 @variable third_col;
4
5 table tr td.first
6 {
7   constraint: width = first_col;
8 }
9
10 table tr td.second
11 {
12   constraint: width = second_col;
13 }
14
15 table tr td.third
16 {
17   constraint: width = third_col;
18 }
```

Code example 2.15: *CCSS implementation of Table Alignment use case.*

Expressions and Arithmetic

While it is possible to do arithmetic in SASS, the expressions are evaluated by means of simple constant folding. There is no way that a mutable variable can be used in these expressions.

CCSS and PSL have the ability to do simple arithmetic too. In the Aspect Maintenance use case for CCSS, which can be seen in code example 2.17, it can be seen how arithmetic is possible on lines: 6, 23, and 24. Line 7 and 8 in code example 2.16 shows an example of arithmetic expressions in PSL.

CSS 3, DSSSL, XSLT and JSSS can also represent arithmetic expressions. The capabilities of CSS 3 in this regard are very similar to that of SASS, but in CSS 3 it is possible to mix relative units with constant units. In addition, DSSSL, XSLT, and JSSS each have a number of predefined functions that make expressions quite powerful, for instance by means of the string manipulation functions, and conversion functions.

```
1 MEDIUM mosaic;
2 PRESENTATION centering FOR html;
3
4 RULES {
5   DIV {
6     if (getAttribute(self, "id") == "innerDiv") then
7       VertPos: Top = ((Parent.Actual Height-Self.Actual Height)/2) + Parent.Actual Top
            ;
8       HorizPos: Left = ((Parent.Actual Width-Self.Actual Width)/2) + Parent.Actual
            Left;
9     endif
10  }
11 }
```

Code example 2.16: *PSL implementation of Centering use case.*

Conditional Statements and Functions

As can be seen in code example 2.16 on line 6, PSL has conditional statements, which is a considerable improvement in relation to CSS for instance. CCSS also has conditional rules, but these can only be used as preconditions for an entire style sheet, which is very coarse grained compared to PSL.

DSSSL, XSLT and JSSS also have the ability to do conditionals in computations, to determine whether or not to style something. In addition to that, DSSSL and JSSS have the abilities to define arbitrary functions, which can be used in expressions, as a way to reuse code. XSLT and PSL also have a way to reuse code, and that is by means of templates and elaborations, respectively. In XSLT, the templates are the primary way to express computation, and these can be applied multiple times. The elaborations in PSL can likewise be reused several times, but it is uncertain whether or not they can be nested recursively. CSS and SASS have neither conditional statements nor the ability to declare functions.

```
1  @variable body-width;
2  @variable content-width;
3  @variable image-aspect;
4
5  @constraint content-width = 400px;
6  @constraint image-aspect = 3/4;
7
8  body
9  {
10   constraint: width = body-width;
11 }
12
13 div
14 {
15   constraint: width = content-width;
16 }
17
18 img
19 {
20   position: absolute;
21   top: 0;
22   right: 0;
23   constraint: width = body-width - content-width;
24   constraint: height = (body-width - content-width) / image-aspect:
25 }
```

Code example 2.17: *CCSS implementation of Aspect Maintenance use case.*

Accessing element properties

The properties of an element refer to the data about the visual appearance of the element, in contrast to element attributes, which live solely in the markup.

In SASS, it is not possible to read a property from an element in order to get its height for instance. It is however possible in CCSS to get the value of properties, or at least bind variables to them.

This makes it possible to make the style of other elements derive from the properties of a given element, exactly as seen in code example 2.15. Here the width of all the elements that are selected by the rule on line 5 will be based on the width of the others. This is also possible in PSL to a limited extent. In PSL it is only possible to read the values of properties of the parent of the element of which a property is to be set. This prevents elements closer to the root to have properties derived from elements closer to the leafs of the document tree.

DSSSL and XSLT both allow accessing arbitrary properties of arbitrary elements as well. In fact, both of these can make use of their selection mechanisms for this. XSLT however has a disadvantage, in that it has done its job before the document get to be displayed, after that the generated CSS is in play. So it can only access the element properties it has specified through CSS style attributes, which is quite cumbersome. JSSS has access to element properties through JS, which also means that it can retrieve the rendered/computed properties of an element.

In XSLT, it is possible to use XPath as an embedded expression, to further select elements from the tree in order to take other elements into account when computing a value.

CSS has no means what so ever to access properties of other elements.

2.4.3 Evaluation

What we have found out is that there are many interesting styling solutions, and that many of them have some interesting ways to specify styling.

Unfortunately, the more powerful solutions are also the ones that seem to be least widespread today, with the exception of XSLT. The most widespread solution is CSS, which we took a closer look at.

Some use cases were thought up and explained. All of the existing styling solutions were then researched to a degree that allowed us to determine what use cases they could implement. Some of the possible solution and use case mappings were then selected to be implemented, and were implemented. For many of the solutions, it has not been possible to get hands on code examples that would allow us to verify if the solution can implement specific use cases. In those cases, they were implemented to the best of our knowledge.

The research shows that there exist more expressive solutions than CSS, and it even shows that CSS 3 is minimally more expressive than the currently widely available CSS 2.1. The least expressive of these alternative styling solutions is SASS, but it is still marginally more expressive than CSS, in that it allows constants to be defined and computed. CCSS is the second least expressive, yet it is an extremely interesting approach. With some more powerful selectors than those of CSS (which it relies on), it could be in the top, because that is the area where it lacks.

With regards to the number of use cases the solutions could implement, XSLT is on a shared third place with PSL. XSLT is however mostly about element structure, and even though the computational powerfulness of XSLT is better than that of PSL, PSL is still more expressive when it comes to styling of the visual appearance of elements. The reason for this is that XSLT relies on CSS, and has done its job before the information is presented to the user. PSL does however run while the information is shown to the user, and therefore has the advantage of taking computed values into account as well as user interaction.

With respect to styling, DSSSL has a lot of good ideas, especially the idea of combining styling with a general purpose programming language. One of the downsides is however that DSSSL is very complex, which has also prevented many of the use cases to be implemented, and even prevented us from determining if some use cases even could be implemented. Despite of this, DSSSL gets a second place, because it inherits all the traits of a general purpose language, while still being tailored to styling.

While JSSS is, by far, the one that can implement the most use cases, this is mostly due to the JS part. This is also why most of the use cases have been deemed to be “possible but cumbersome”, because the solution would rely more on JS than JSSS. The only thing JSSS adds to JS seems to be the idea of using a callback for styling an element. The styling properties are however more accessible than with JS alone. Despite its very few additions to JS, it is still the winner when it comes to implementing the use cases. But JS without the JSSS extensions would still be the winner.

In section 1.1 some wanted features were listed. No single existing styling solution offers all of the wanted features. All of the solutions, with the exception of CSS 2, are capable of doing arithmetic expressions. CSS 3 and SASS can however only perform arithmetic on constants.

With regards to advanced selection of elements, only DSSSL and XSLT are convincing. PSL does however allow a certain degree of expressiveness in this compartment, by making

it possible to use conditions. Again JSSS rely mostly on JS, and can therefore not be said to have particularly advanced selection features.

When it comes to relative styling, CCSS excels with its constraints. JSSS again has the advantage of JS, and therefore have access to everything. PSL can also perform styling based on elements, but is limited to only reading properties from parent elements. Only DSSSL and JSSS can be said to be integrated into an UPL.

2.5 Web Browsers

The only way a user can get access to a web site is through a web browser, so web browsers naturally play an important role in this report. In this section, web browsers will be taken into deeper consideration, especially with regards to the process of transforming a document to something that can be rendered on the user's screen. As described in section 2.1.1, CSS is the styling language that is most used for styling web sites today, so this section will be concerned with how CSS is handled by browsers.

First it will be described how a web browser renders a static web page using the CSS styles available from the web site, the user, and the default styles of the web browser itself. Then it is described how dynamic changes to the CSS styles can be performed using JavaScript, and how these are handled by the web browser. This gives an overview of the interplay between the web browser, the source code, CSS, and JavaScript. This overview will later be used as a foundation, when it is determined how the new styling solution technically should be implemented.

2.5.1 Layout Engines

A web browser consists of multiple components that each provides some specific functionality to the browser. These components are, among others, responsible for handling JavaScript execution, bookmarks, downloads, and browser history. But the most interesting, when taking the styling aspects into consideration, is the component called the *layout engine*. This engine performs the work of rendering the source code of a web site onto the canvas of the user's browser. Specifically, it parses HTML and CSS, and creates an internal representation of this, which is then rendered onto the canvas. Some of the most used layout engines and the browsers that use them are:

Trident: Internet Explorer

Gecko: Mozilla Firefox

WebKit: Safari and Google Chrome

Presto: Opera

KHTML: Konqueror

In the following sections, it will be described in more details how a layout engine generally works. The most used layout engine today is indisputable Trident, because of the market share of Internet Explorer of above 65%[41]. It is however very difficult to find information about Trident, because it is proprietary software. Instead, the Gecko [45] layout engine, which is the second (above 21%) most used, will be used as basis for the description in this section[41]. Gecko is an open source project, and information about it is more easily available than for Trident.

2.5.2 Rendering Static Content

When a layout engine renders a web site, it goes through a series of processes as illustrated in figure 2.2. In the figure, the light-blue rectangles are the processes, and the green boxes are data structures. In the following sections, each of the processes, and the data that they generate, will be described in more details. The Parser and Content Sink processes are considered as one process in the following description The description is based on documentation from Mozilla[46][47].

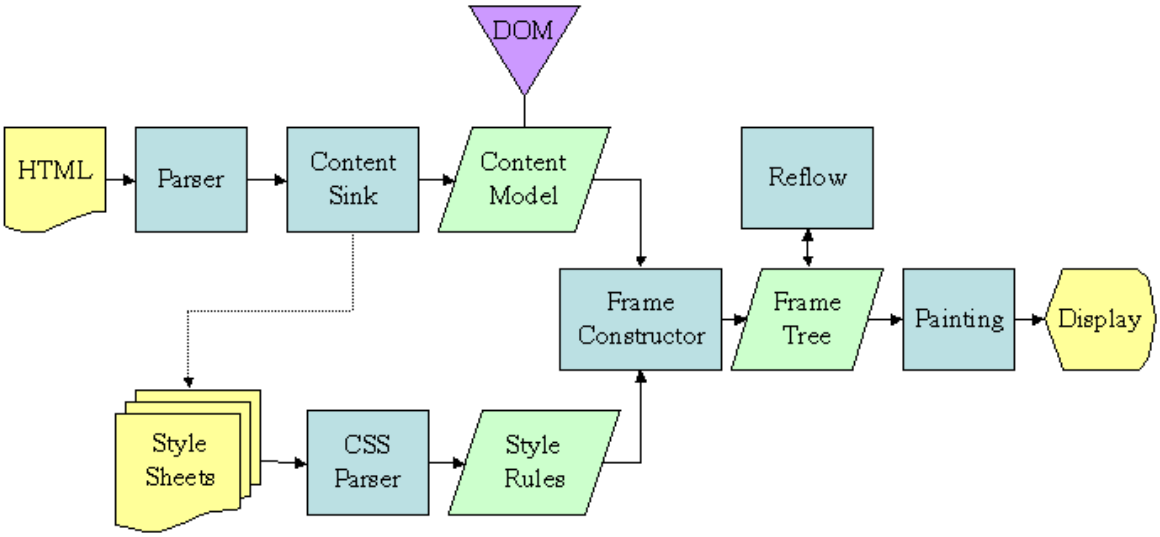


Figure 2.2: *Rendering process in Mozilla Firefox. The figure is created by L. David Baron from the Mozilla Corporation for a presentation of Mozilla's Layout Engine[46].*

Parser

The layout engine takes the *source document* as input, and parses it into a *content model*, which is implemented as a tree structure. In the content model, each element from the source document is represented as a node. In the implementation of the layout engine, there is one data type for each possible DOM element. And each node in the content model has the data type that corresponds to the DOM element it represents. It is the content model which can later be accessed through the DOM interface methods, which will be described in more details later in section 2.5.3.

CSS parser

During the parsing of the source document, all CSS style attributes on elements and embedded style rules are passed to the CSS parser, as illustrated by the dashed line in the figure. The CSS parser then parses this styling information, together with any external style sheets attached to the document. The parsing of all the CSS styling information results in a set of *style rules*.

Frame constructor

The frame constructor uses the content model and the style rules to create a *frame tree*, which is a tree of rectangular *formatting primitives*. Each rectangular formatting primitive represents a content node and thereby one of the elements from the source document. Later, the reflow process of the layout engine uses the frame tree and the formatting primitives in it, to decide how each of the content nodes should be positioned in relation to each other.

Before the frame tree is created, a *style context* is created for each node in the content model. A style context represents *style data* for a formatting primitive. The style data consists of a list of CSS properties and their values.

To create a style context for a node in the content model, the style context of the parent node in the content model is needed, together with style data defined by style rules for the current content node. In the following list, it is described why the data is needed and how it is accessed.

Parent style context: The style context of the parent node in the content model is needed, as some CSS properties, if not defined for the current node, should inherit the value of the parent node. Examples of this are the CSS properties "font" and "color". A *style context tree*, which has the same structure as the content model, contains all the style context data that have been created for each node. This style context tree is used to get the style context of the parent node. As each node uses the style context of the parent node to create its own style context, the style context data are created for the elements in the content model in a top-down manner.

Style data: Besides the styling from the parent content node, a content node can of course be styled by style data in style rules, so this data is needed. To get the style data defined specific by style rules for the content node, the following is performed: All the parsed style rules are iterated through with the most specific rule first (as defined by the CSS cascade). For each of the style rules, it is then tested whether the selector of the style rule matches the content node. All the style rules where this is the case, are then used to compute the style data. To compute the style data, the style data from all the style rules are added to a style struct, with the style data from the most specific rule first. It is not possible for a style rule to add style data for a property that has already been set by a more specific style rule.

Now that the parent style context and the style data defined by style rules are known, it is possible to create the style context for the current content node. This is done by combining the style data in the context style of the parent content node and the style data defined by the style rules. If there are any conflicting properties, then the ones defined by the style data in the style rules are used.

The process of creating the style context for each content node is, as seen in the above description, a time-consuming process. But it has been optimized in various ways:

Hash tables: To decrease the time used to find style rules that match a content node, hash tables have been used. The indices used in the hash tables are tag name, ID and class name. So it is quick to find style rules that match a content node, if the selectors in the style rules use tag name, ID or class name.

Rule tree: To avoid iterating through all the style rules for each content node, the result (style rules that match the content node) of each iteration is cached in a *rule tree*. So

when style rules should be found that match another content node, and that content node is similar to a content node that already has been looked up, the result is read directly from the rule tree. This is for instance good when having a table with 100 `tr` content nodes, because it is only necessary to find matching rules for one of these.

Now that a style context has been created for each content node, the frame tree is created. One node is created in the frame tree for each node in the content model, with one exception. If the style data in the style context of a content node has its CSS property `display` set to "none", then the content node will not be inserted in the frame tree.

In the frame tree, each node has a data type that corresponds to the value of the CSS `display` property in the style data of the style context of the content node. Some of the possible values are "text", "inline" and "block". All the nodes from the frame tree can be seen as rectangles (or *frames*) that have to be drawn in the user's browser window. The frames do however not any size or position yet.

Reflow

The frame tree is used through the process reflow, where it is determined where each of the frames should be positioned and what size they should have. This is a complicated process, which is out of the scope of this report.

Painting

When the frames have been positioned, it is finally possible for the layout engine to draw the web page on the canvas in the user's browser.

2.5.3 JavaScript and Styling

In this section, it is described how the styling of elements in a document can be changed dynamically, and how these changes are performed by the layout engine, by relating to the previous section about layout engines. The description is based on the official documentation from Mozilla for Firefox[48] and a dedicated JavaScript web site[49].

Updating and creating new styles

As described in the previous section, a layout engine takes a static document and renders it. But with the concept of RIAs, it is often desirable to perform incremental updates to the document. Incremental updates means that small parts of a document are updated continually, and each update is reflected in the document that was created during the previously update. If this is to be done, a whole new document needs to be created that reflects the changes. This document should then be parsed and taken through the whole rendering process described in the previous section. But this is inefficient, so we need some method that allows us to update the styling of the document, without creating a whole new document. JavaScript, which is a general-purpose programming language, provides this ability.

Every modern web browser supports JavaScript as the client side programming language. By using JS, it is possible to change the styling of the elements in a document in two different ways: By changing the external and embedded style sheets or by changing the style attributes on the elements. These two methods are described in more details in the following list:

Style sheets: It is possible to create, modify and delete style rules in external and embedded style sheets. The implementation of this however differs in the various web browsers. An example of how it works in Mozilla Firefox is shown in code example 2.18. In the code example, one embedded style sheet is declared on line 1-12. On line 15, the first external style sheet is selected by using JS code, it is this style sheet that later is modified. First, the second style rule in the style sheet is selected and modified on line 16-17. The color that the style rule defines is changed from blue to red. Then, on line 19, a new style rule is inserted as the first style rule in the style sheet (index 0). This style rule sets the padding of the `div` element with the id "body" to "10px".

Style attributes: Instead of changing a style sheet, it is possible to change the style attribute on each element in the DOM. The implementation of this is identical in all modern web browsers. Examples of how to change the style attributes of elements are shown in code example 2.19. To get the elements that should be styled, some DOM methods are used to navigate the DOM tree. In the code example on line 2-3, the color CSS property is set to "yellow" for the element with the id "container". This is quite simple and requires only two statements, but if all the elements with the tag name "div" should be styled, it also requires another language construct and thereby a bit more code. An example of this is shown on line 5-9, where all the elements are selected by a simple selector on line 5. But to set the styling property (color), it is necessary to create a construct that iterates through all the elements (line 6). Further, another construct is needed when e.g. all `div` elements that have a `div` element as parent should be selected. For this, we need to create a conditional construct that tests the parent of each element (line 14).

```
 1 <style type="text/css">
 2   div#container {
 3     border: 1px solid black;
 4     width: 250px;
 5   }
 6
 7   div.header {
 8     color: blue;
 9     font-size: 2em;
10     padding: 10px;
11   }
12 </style>
13
14 <script type="text/javascript">
15   var firstStyleSheet = document.styleSheets[0];
16   var secondStyleRule = firstStyleSheet.cssRules[1];
17   secondStyleRule.style.color="red"
18
19   firstStyleSheet.insertRule("div.body{padding: 10px;}", 0);
20 </script>
```

Code example 2.18: *Modifying an embedded style sheet through JavaScript.*

When the style of some elements should be changed, the developer must choose between changing the style sheets or the style attributes. Which method to use depends on the number of elements to be styled, and whether the elements are easy to select by a CSS selector. Changing the style sheets is preferred to use when a lot of elements should be styled identical, and a CSS selector can describe all the elements that should be styled. Changing the style attributes is preferred to use when only a single or few element should be styled, and the elements are easy to access through the DOM traversal methods.

```
<script type="text/javascript">
  var element = document.getElementById("container");
  element.style.color = "yellow";

  var elements = document.getElementsByTagName("div");
  for(var i=0; i<elements.length; i++)
  {
    elements[i].style.color = "yellow";
  }

  var elements = document.getElementsByTagName("div");
  for(var i=0; i<elements.length; i++)
  {
    if (elements[i].parentNode.tagName == "DIV")
    {
      elements[i].style.border = "1px solid black";
    }
  }
</script>
```

Code example 2.19: *Modifying style attributes through JavaScript.*

Implementation of style changes

What JS actually does when it changes some styling, is updating the internal representations of the CSS properties in the layout engine, and then tells the layout engine to re-render the web page. Dependent on which CSS properties are changed, the layout engine performs some or all of the steps in the rendering process shown in figure 2.2. If the `display` CSS property has been changed, the whole rendering process must start from the frame constructor process, as the change influences which frames there are in the frame tree. In case the size of an element has been changed, the frame tree is not affected, and the rendering process can start from the reflow process. The reflowing is needed, as the change of the size of one element might affect the size and position of many other elements. Finally, if the color of an element is changed, the only process that needs to be executed in the rendering process is the painting process.

Getting style information

In JavaScript, it is possible to get the values of the same CSS properties that are possible to set, which was described earlier in this section. These included CSS properties in external, embedded and attribute styles. To get the value of a property from one of these style sheets, the properties are accessed in the same way that is used to set them, as shown in the code examples. But besides the regular CSS properties, it is also possible to get the value of some other very interesting properties. These properties describe the rendered position and size of each DOM element (except elements like `head` and `title`, which are not rendered).

```
<script type="text/javascript">
  var element = document.getElementById("container");
  var left = element.offsetLeft;
  var top = element.offsetTop;
  var width = element.offsetWidth;
  var height = element.offsetHeight;
</script>
```

Code example 2.20: *Accessing the rendered properties of an element.*

The properties can for instance be used to position elements based on other elements positions. For instance, if one element should be placed beside another element, and the position of the other element is based on a non-fixed value (e.g. `width: 25%;` or `margin: 10px;`), then it is simple to get and use the rendered position of the other element to place

the first element. The values of these properties are calculated in the reflow process in the rendering process of the layout engine, so they are read-only. To set the position of an element or its size, one must use the CSS properties. An example of accessing the rendered properties is shown in code example 2.20. In the example, the four rendered properties of the element with the id “container” are assigned to variables.

Chapter 3

Problem Statement

The analysis of existing styling solutions in section 2.1, which included market penetration, showed that styling in web development today is limited to only a few solutions, primarily CSS but also XSLT. To test these solutions and all the other existing styling solutions, some use cases of styling challenges were presented in section 2.3.

For each of the use cases, it was considered which of the styling solutions could implement the use case, and if it would be cumbersome to do. One important experience from this evaluation was that only a few of the use cases could be implemented in CSS (both version 2 and 3). The expressive power of CSS is inadequate to implement several of the use cases, as described in the section 2.4. To sum up, this is the case because the selectors are not advanced enough, it is not possible to express interelement constraint, and expressions used to calculate the values for styling properties can only be constants. It is not possible to implement all the use cases in any of the existing styling solutions, but CCSS, DSSSL, XSLT, PSL, and JSSS can all implement more than CSS. This is because each of them is more powerful than CSS.

But these styling solutions, with XSLT as an exception, have not become widespread, and the knowledge available about them is rather limited. This is most likely because the solutions are very old, and because each of the solutions have only been implemented in browsers that are not available anymore. Even though XSLT is more powerful than CSS, and the solution is also widely used, XSLT is primarily used for changing the structure of elements in a document, and has almost nothing to do with the appearance of elements. Therefore XSLT is not considered as a viable alternative for styling documents.

So there is a need for a new styling solution that can implement all the use cases, and this requires a solution more powerful than CSS.

New styling solution

To create a new styling solution that is more powerful than CSS, it is first necessary to define what "more powerful" covers. In the following list, it is described how the different aspects of a styling solution could be created more powerful. The list is based on experience gained from implementing the use cases in section 2.4.

Element selectors: It should be possible with some predefined operators to select elements based on their content and the context in which they exist. To give even more

expressive power to the selectors, it should also be possible for a developer to define his own arbitrary constrains in form of predicates.

Interelement constraints: When styling an element, the solution should have the possibility to access properties of other elements. These properties should be the ones that are set by other styling rules, but also properties that represent the actual position and size of elements in the browser.

Expressions: To increase the expressiveness of expressions, it should, besides accessing properties of other elements, be possible to use arithmetic, variables and functions.

The centralized problem that the rest of this report will be concerned about is as follows:

Create a styling solution that can implement all of the use cases, and is more powerful than CSS. In addition, the styling solution must be able to run natively in the browser. The problem includes designing the language of the solution and implementing a prototype.

Chapter 4

Ideas

In this chapter, some of the ideas and thoughts for the design of the new styling solution are presented. The styling solution is called *Powerful Dynamic Styling Solution* (PDSS), and will be referred to as PDSS in the rest of the report. PDSS is highly based on creating rules, and in the first part of this chapter, the ideas for the structure of a PDSS style rule are presented. Then, the ideas for the features of PDSS are presented and prioritized. Some of the challenges that the features may cause are then discussed. Finally, there will be a discussion about how PDSS should be realized.

4.1 Style Rule Structure

In this section, the ideas for the structure of a PDSS style rule are presented. Each part of the style rule is given a name, and these names will later be used to refer the individual parts. The structure of the style rules in PDSS is highly inspired by the structure of rule sets in CSS. The structure of a single style rule in PDSS is shown in figure 4.1, a PDSS code will most often consist of multiple style rules.

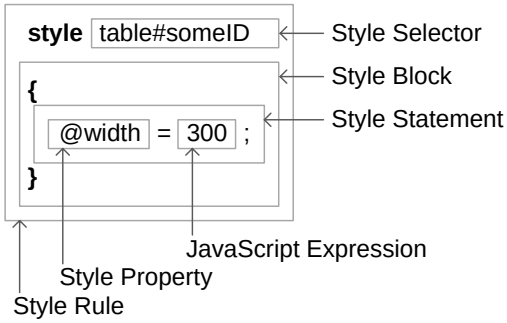


Figure 4.1: *Structure of a PDSS style rule construction.*

4.2 Ideas for Features

At the very beginning of the report, in section 1.1, a number of wanted features were put forth. In this section, the ideas for features that have emerged through the analysis of the existing styling solutions will also be listed.

Integration into the UPL

As mentioned in the introduction of the report, PDSS is supposed to be integrated into a programming language. This programming language will be called the underlying programming language (UPL).

This integration means that style statements are written in the UPL, and at the same time it allows expressions of the UPL to be integrated into the styling constructs. It will for instance be possible to use arithmetic expressions, variables, constants and functions from the UPL to set a color property inside a statement, as well as in predicates in style selectors.

The integration into the UPL can be made even tighter, and allows statements from the UPL to be placed inside styling blocks alongside styling statements. This will allow very fine grained control of the styling, as well as complete expressive power.

Predicates

In a selector, it should be possible to use predicates to further filter a selection of elements. A predicate is an expression about a particular element that has to evaluate to true, in order for that element to be matched. This should be possible by means of user-defined functions in the UPL and built-in predicates as known from CSS (e.g. `hover` and `focus`). When using a function as a predicate, it is referred to in the selector through its function name. This function is evaluated for each of the elements that match the previously part of the selector. If the function evaluates to true, then the element is included in the result of the selector. If the function evaluates to false, then the element is omitted.

Environment information

In expressions, it should be possible to use information about the browser environment to adapt the appearance to the current browser. Environment information is e.g. the size of the browser window, the coordinates of the view port, and the browser version. Some of the styling solutions already have access to the dimension of the canvas of the browser, but this could be extended to having access to the position of the canvas too. That can help the developer to glue things onto the visible area, such as a menu.

Element properties in expressions

When having an expression in a style selector or a style statement, it should be possible to get and use the values of properties of other elements. This include properties that have been explicitly set by the developer earlier, properties that have been inherited, and properties that represent the rendered size and position of the element.

First class values

One of the ideas is to have the style selector and style blocks as first class values in the UPL. A complete style rule will then be assembled from these two distinct values, at the point where the style is needed.

Having the style selector as a first class value allows the developer to create functions that can do computation based on the result of style selectors. Having this possibility allows the developer to create new operators for use in style electors.

Similar things are possible with first class style blocks. First class style blocks can be combined to form a new style blocks, and altered in order to change styles dynamically.

Dynamic rule modification

While it is possible to apply a lot of styles, it is also interesting to be able to remove styles again, as well as changing the currently applied styles. While this, in many cases, can be done by applying a new style that reflects the changes, it may be desirable to have the possibility of editing rules. This could both be possible before the rule is applied to the document tree, but also after.

Nested rules

One feature that has no effect on the expressive power of the solution, but nevertheless is very useful in a user-friendly and language construction perspective, is nested rules. Nested rules mean that it is possible to insert a new style rule into the scope of the style block of an existing rule. In the new rule, it is again possible to insert another new rule and so on into an arbitrary depth. A rule that is inserted into another rule is called the *inner rule*, and the rule, into which the inner rule is inserted, is called the *outer rule*. When a rule is inserted into another rule, the selector of the inner rule is only matched against the elements that match the selector of the outer rule. By having nested rules, the developer avoids to write the selector from the outer rule each time he wants to style some elements that are inside the matched elements from the outer rule. The feature about nested rules is inspired by SASS, which was described in section 2.1.6. An example of nested rules in SASS is showed in code example (2.6).

4.2.1 Prioritization

This section contains a prioritization of the features and argumentation for the prioritization. The prioritization will be used to guide the development of PDSS, as the highest prioritized features will have most focus and be implemented first. The features are prioritized as follows:

1. **Integration into the UPL:** The main idea about PDSS is to create a powerful styling solution. So the most important feature naturally is that PDSS operates on top of a powerful web programming language that can provide the expressive power.

2. **Element properties in expressions:** The ability to base some styling on properties of other elements, especially sizes and positions, are very important. It allows creating advanced layouts, where the positions of some elements are base on the positions of other elements. This is particular useful when the content of the other elements, and

thereby their sizes, are unknown. This is for instance the case then creating RIAs, where the content of an element might change.

3. **Predicates:** Having predicates is an important feature because predicates give the developer great expressive power in the selectors. With the predicates, it is possible for the developer to specify any possible conditions, such that he is not only limited only to a few.

The features prioritized until now are the most important. Important features in this case mean features that add expressive power to the styling solution in contrast to traditional styling solutions (CSS). This means that they make it possible for the developer to do styling that is not possible in traditional styling. The rest of the features do not provide more expressive power, but are abstractions that make it easier and more logical to do styling.

4. **First class values:** When the feature was described, it was stated that one of the advantages of first class style selectors is that it would be possible for a developer to define his own style selector operators. This will of course give the solution great expressive power, but the built-in operators and the very expressive predicates already give enough expressive power. First class style blocks could be used to allow combining style blocks, but the result of this can also be achieved just by creating a new style block that contains the same style statements as in the two existing style blocks. So first class values do not add expressiveness, and that is the reason that the feature is not prioritized high.

5. **Dynamic rule modification:** The effect that can be achieved by modifying or even deleting a style can be achieved otherwise by creating a new style that overrides the first (by setting the same style properties). Because of this, the feature is not very important. It is however troublesome to delete a style by creating a new style, because this requires that the developer knows all the default values of the properties that are set in the existing style.

6. **Environment information:** By using JavaScript today, it is not difficult to get environment information like the size of the browser. This information can however not be used directly when creating CSS styles. So a developer who wants to adapt some elements to the environment must do this in another way. This is typically done by changing the relevant CSS properties of the elements through JavaScript, such that the elements fit the environment (see section 2.5.3). So this feature is possible today, but as it not is an integrated part of the styling solution, it is cumbersome to use. Therefore the feature is prioritized low.

7. **Nested rules:** This feature is good to have in a styling solution as it eases the development and makes complicated styling code easier to understand. It does however not increase the powerfulness of the styling solution, and is therefore not considered so important.

4.3 Challenges

In the following sections, some of the problems regarding styling based on rules are presented, along with some possible solutions.

4.3.1 Conflict Resolution and Elimination

As with CSS, several rules can be in conflict with each other. For instance, setting all `div` elements to be red in one rule, and then setting all `div` elements to be blue in another. In CSS, these conflicts are solved by means of specificity and cascading rules (see section 2.2). We feel that some of the semantics of this way to resolve the conflicts can cause elements to be styled in unexpected ways.

To solve this, the idea is to have a simpler conflict resolution scheme, which will be based on the order in which rules are introduced. This also makes more sense when styling is meant to be done in a dynamic fashion rather than statically.

```
1 div
2 {
3   color: red;
4   font-size: 14pt;
5 }
6
7 div
8 {
9   color: blue;
10 }
```

Code example 4.1: *Two rules, one overriding the `color` property of the other.*

When having a rule that defines the values of several properties; X, Y and Z, and then having a second rule with the same selector, but only defines the properties; X and Y, the previously defined value of the Z property should stick, while X and Y are redefined. Code example 4.1 gives an example of an obsoleted property declaration. CSS notation is used here because the syntax and the semantics of our solution will first be defined in section 5.2. In order to perform elimination of obsoleted declarations, we need to split rules into sets of rules that define only a single property each. The point in eliminating obsolete declarations is to avoid computing values that will be overwritten by subsequent evaluations.

If the declarations in a rule are simple assignments, it will be trivial to eliminate obsolete declarations of properties. Code example 4.1 is suitably simple. But if conditional statements and loops are allowed inside style blocks, it will become harder to determine if the rule can be eliminated. In fact, there can be cases where this will be undecidable, because it is undecidable to determine if two arbitrary program pieces will perform equivalent computations. The same holds true if the developer decides to use homemade predicates in selectors, since these predicates are simply conditional statements that call a function which can contain arbitrary code.

For this reason, we need to explicitly support a range of sensible predicates, because that will allow us to compare the semantics of the two predicates and eliminate the other. For instance, assume that the developer creates a rule using a predicate that says every sixth child must be selected. Later, the developer creates an equivalent rule, where the predicate selects every third child. Clearly every sixth child is covered by every third child, so we can remove the first rule altogether if we know the exact semantics of the predicate. If the developer implemented the predicate, there would be no guarantee that we could eliminate any of the rules. Now assume that the precedence of the two rules are opposite. The rule that selects every third child will now be overruled by the other rule, which means that some of the elements will have their property written twice. Again, this can be prevented if the semantics of the predicates are known.

When it comes to selectors without predicates, there should not be any computational problems determining what elements two different arbitrary selectors have in common, in order to eliminate excessive writing of properties.

4.3.2 Property Dependencies

The idea that the styling of an element can depend on the style of another element can yield some problems. For instance, given two elements, A and B, where B is a descendant of A, and the developer creates a style that says A must be twice as wide as B. Usually, the width of B will automatically be determined by the width of A, so what is supposed to happen when a style suddenly says that A must be defined by the width of B? Having such a rule does not make sense, because the width of both A and B will really depend on how many times the rule is evaluated.

Suppose that two other elements, C and D, where D is a sibling to C and C comes before D in the document tree. Usually, C will be evaluated first, and then D. Now suppose we make a rule that says C must be twice as wide as D. There should not be anything wrong with this, as the width of an element should not affect the width of its siblings. But we still need a way to make sure that the width of C is not evaluated until D has been given a width.

In order to detect cases like this, a graph of dependencies needs to be generated. If a rule generates a cycle in that graph, the rule cannot be enforced, and an error will be emitted. If there are no cycles, then the rules can be ordered in a way that will allow all dependencies of a rule to be evaluated first.

This must however be combined with the concept of splitting rules as described in the section about conflict resolution. This is because two rules can each set two properties, where one property from one rule depends on one property of the other rule, while the other property of the other rule depends on the other property of the first rule. See Code example 4.2 for an example.

```
1  A
2  {
3    color: red;
4    font-type: B.font-type; // set the font type to that of element B
5  }
6
7  B
8  {
9    font-type: Verdana;
10   color: A.color; // set the color to that of element A
11 }
```

Code example 4.2: *Double dependency.*

If we look at the two rules in their entirety, they form a circular dependency. But these two rules can easily be split into four rules that do not form a circular dependency.

4.4 Underlying Programming Language

Since PDSS is meant to expand the capabilities of styling within RIAs, it would be natural to integrate PDSS into one of the existing web frameworks.

PDSS should be implemented in a way that allows it to be executed natively in the browser. Some of the choices that have been considered are; Google Web Toolkit, Echo Web Framework, and Links. All of these web frameworks can run natively in the browser, because they are compiled to JS.

But in the end we decided to simply use JS as the UPL, since this will allow us to concentrate more on the actual styling features rather than the integration with a framework. The choice of JS also ensures that PDSS will work in most browsers and thereby have a long lifespan.

The way PDSS will be integrated into JS is by making a compiler that can parse JS code. The styling constructs from PDSS will be written inside and among the JS constructs. When the compiler is given a source file, it will skip over all the JS constructs. But as soon it meets some PDSS constructs, it will generate some JS constructs that take care of the styling. The generated source will be assisted by a JS library that contains the fundamental implementation of the styling mechanisms from PDSS.

Chapter 5

Design

Based on the experience from the analysis in chapter 2 and the ideas in chapter 4, we have created a design for PDSS. In this chapter, the design of PDSS is presented.

First, the basic concepts of PDSS are presented to give an overview of the design. Then the actual syntax and semantics of PDSS is presented. At last, it is described which helper functions the PDSS library provides.

5.1 Overview

PDSS is an extension of JS, which makes it possible to do styling from JS in a CSS-like manner, but with several advantages over CSS. The structure of style rules in PDSS is chosen to be the same as the structure idea presented in section 4.1. Consider code example 5.1 which is a simple demonstration of a few of the features of PDSS. All of the features of JS are kept, and a few new styling features are added on top.

```
1  var text_color = "green";
2
3  // Change the font and color of all paragraphs and list-items
4  style p + li
5  {
6    @fontFamily = "Verdana, Arial, sans-serif";
7
8    // test if the color is acceptable
9    if (text_color != "pink")
10   {
11     @color = text_color; // use the color in the JS variable.
12   }
13 }
```

Code example 5.1: *A simple code example of our PDSS*

Line 1 creates and initializes an ordinary JS variable. Line 4 begins a style rule, which is a PDSS construct. This PDSS style rule is conceptually similar to a CSS rule set, from which it was inspired. As a rule set in CSS, a style rule in PDSS starts with an expression that defines what elements will be altered. After that expression, a block describes what properties and values will be set for those elements.

Unlike CSS, PDSS is dynamic. The rule will first take effect when the flow of execution passes such a rule. Likewise, it is possible to have computation carried out as part of the styling, as can be seen on line 9.

The properties and the values for the properties in PDSS are entirely based on the properties and values known from CSS. This is apparent on line 6, where the `fontFamily`

property is directly corresponding to the `font-family` property of CSS, and the value `"Verdana, Arial, sans-serif"` is taken verbatim from CSS. The styling caused by setting these properties with corresponding values, will also be exactly that of CSS.

It does however cause undefined behavior if inline, embedded or external CSS are used alongside PDSS.

The previous example was just a simple PDSS example. Code example 5.2 shows a slightly more advanced example of what is possible in PDSS.

```
1  style *#fancy
2  {
3    @backgroundColor = "blue";
4  }
5
6  var selector = ${ div / *[hover(e)] };
7
8  var block = @{
9    @border = "thin solid red";
10   @backgroundColor = ${ *#fancy }@backgroundColor;
11 };
12
13 createRule(selector, block); // apply the style
14
15 sleep(15); // wait 15 seconds
16
17 destroyRule(selector, block); // remove the style
```

Code example 5.2: *A simple PDSS code example*

Line 1-4 makes the element with the id "fancy" have a blue background. On line 6, a variable `selector` is declared and initialized with a first class style selector instance. The first part of the style selector (`div / *`) will select all elements that have a div element as ancestor. The second part of the style selector (`[hover(e)]`) filters these elements, such that only hovered elements are selected. To do the filtering, the second part is executed once for each of the elements that match the first part of the selector. The symbol `e` is a reference to the element that matched the first part of the selector, and the symbol `hover` is just an ordinary JS function. A style block is created on line 8, and assigned to the variable `block`. The style block specifies two properties to be set, and their values. The value of the background color property is defined to be the same as the background color of the element styled on line 1. On line 13, the selector and block is turned into a rule, and the styling will then take effect. If the background color of the element styled on line 1 is ever styled with a new color, this will affect the style of the elements that match the selector. These elements will get the same background color as the new background color of the element styled on line 1 Line 15 halts the execution for 15 seconds, after which the rule just created will be destroyed. That causes the selector and block to no longer have any effect on the styling of the document.

Basically the code causes all hovered elements with a div element as ancestor to be styled with a red border and blue background, but only for the first 15 seconds of the execution.

When having some PDSS source code (see code example 5.1 and 5.2), it needs to be compiled to pure JS and included into a HTML document before it will take effect.

In addition to that, an external library that implements all features of the styling constructs in PDSS needs to be included. When the PDSS compiler gets a JS file, it just skips over all the actual JS. When a PDSS styling construct is met by the compiler, this construct will be transformed into JS that makes use of the external library. Implementation details are described later in chapter 6.

5.2 Syntax and Semantics

As described in section 5.1 PDSS is an extension of JS. In this section the extension is described in more details.

The new language constructs that the extension provides are the following:

- Style selectors
- Style properties
- Value selectors
- Style statements
- Style blocks
- Style rules

The following sections contain descriptions of each of the six new language constructs. For each of the constructs the syntax and semantics are described, and some code examples are presented which demonstrates the constructs.

The syntax of each language construct is presented by means of production rules from the PDSS grammar. If a language construct corresponds to a single production rule in the grammar, then this rule is listed. If a language construct corresponds to an alternative in a production rule, then this rule is listed, but with the other alternatives left out. If one of the production rules contains symbols for other production rules, then these production rules are also listed, if they are considered as important to understand the syntax. As the parser generator ANTLR[50] has been used to implement the parser for PDSS, the grammar is in ANTLR format. Note that ANTLR uses EBNF to specify grammars. In the ANTLR grammar, fully capitalized symbols are tokens and symbols starting with a lowercase character are production rules. To increase the readability of the syntax, the tokens in the production rules have been replaced with the corresponding terminals.

The whole grammar for PDSS is available in the appendix of the report (section A.2).

5.2.1 Style Selector

A style selector is a pattern that matches nodes in a tree structure. Style selectors can be handled as first class values that can be executed similarly to a delegate. When a style selector is executed it will return a list of elements from the DOM. The elements in the returned list of elements are said to be matched by the selector. Style selectors can be used in expressions as any other value in JS.

Style selectors can be created by means of a style selector literal. In order to identify a style selector literal it is enclosed between '${' and '}'. Style selectors are also present as part of a style rule. In that case, the style selector does not need to be enclosed with special characters.

The rule in the grammar that corresponds to a style selector is `selector` and this rule is shown in syntax example 5.3 together with other relevant rules.

A selector consists of a sequence of element names glued together with operators to express a pattern. An element name can either be an asterisk (`*`) or an alphanumeric unquoted string. For example `*`, `div`, and `abc012xyz789` are all valid names, `abc012x*yz789` is invalid. The asterisk is a wildcard for all possible element names. An element name matches all DOM elements with that name.

```
 1 selector
 2   : selector_Term_First (('^' | '+' | '-') selector_Term_First)*
 3   ;
 4
 5 selector_Term_First
 6   : (NAME | '*') selector_Predicate* selector_Term?
 7   ;
 8
 9 selector_Term
10   : selector_Operator (NAME | '*') selector_Predicate* selector_Term?
11   ;
12
13 selector_Predicate
14   : '[' js_expression ']'
15   | ',' NAME
16   | '#' NAME
17   ;
18
19 selector_Operator
20   : '/' ('(' js_expression ')')?
21   | '.' ('(' js_expression ')')?
22   | '<' ('(' js_expression ')')?
23   | '>' ('(' js_expression ')')?
24   ;
25
26 NAME
27   : AplhaNum+
28   ;
```

Syntax example 5.3: *Selectors*

Right after an element name it is possible to specify zero or more predicates, as can be seen in syntax example 5.3 on line 6 and 10. A predicate is enclosed in square brackets; '[' and ']'. A name and its following predicates will be referred to as an *atom* in the rest of this chapter. When one or more predicates are present for a given element name, all of the predicates must be true in order for a DOM element to match the atom.

Any JS expression can be used as predicate, which means that all accessible symbols can be used in the predicate as well. Whether or not the predicate yields true or false follows the same rules as using the predicate in a JS `if` control structure. So what normally yields true in JS will also yield true as predicates in selectors.

Since style selectors themselves can be used in expressions, it is possible to use a style selector inside the predicate of another style selector. Inside a predicate, the symbol `e` will be bound to the DOM element on which the predicate is to be tested for. That means, that the symbol `e` is bound to an element in a similar fashion as the keyword `this` is bound to an object instance in various OOP languages (including JS). If predicates contains nested style selectors with predicates, the symbol `e` will refer to the nearest surrounding selector.

In addition to the general predicates, there are two shorthand notations for specifying that an element must have a specific id or class. This is reflected in the production rule named `selector_predicate` in syntax example 5.3. A comma followed by an alphanumeric unquoted string, for instance `,menu` specifies a predicate that will only be true for DOM elements that are of class "menu". If the comma is swapped with a hash (`#`), then a predicate would test the id instead of the class. The definition of class and id corresponds to those defined in the CSS specification[24].

The predicates for a single atom will always be evaluated together, left to right. But the overall order of evaluation of predicates is implementation specific. The number of times the predicates for an atom is evaluated is undefined. It is possible to cause side effects from predicates, but relying on such side effects is an error.

There are two kinds of operators that can be used to glue atoms together, *match operators* and *set operators*.

There are four match operators, and they all have higher precedence than the set operators. A sequence of match operators specifies a chain of relations among elements in the DOM tree, and as such there is no particular order of evaluation. The last atom in a chain of relations denotes the element to be matched/selected.

The four match operators are '`/`', '`.`', '`<`', and '`>`', which respectively corresponds to the relations; descendant, ancestor, left sibling, and right sibling. For instance, `a / b` specifies that `b` is a descendant of `a`, so all `b` elements which have an `a` element as an ancestor, will be selected. The other three match operators work in similar ways. Consider the following examples.

Descendant: `a / b` matches all `b` elements that have an `a` element as ancestor. That is, all `b` elements contained in a subtree with an `a` element as root.

Ancestor: `a . b` matches all `b` elements that have an `a` element as descendant. That is, all subtrees with a `b` element as root, that contains an `a` element.

Left sibling: `a < b` matches all `b` elements that have an `a` element as right sibling. That is, all `b` elements for which there exist an `a` element with higher index. The index of each element is one greater than the index of its left sibling.

Right sibling: `a > b` matches all `b` elements that have an `a` element as left sibling. That is, all `b` elements for which there exist an `a` element with lower index.

For each of the four match selectors it is possible to specify a number of *links*. For instance, there is one link between a parent and a child, while there are two links between a grandparent and its grandchild. The number of links between two siblings is given by the absolute difference between their index values.

The number of links is expressed by a JS expression between '(' and ')'. For instance, `div /(2) p` will select all `p` elements that are grandchildren of a `div` element. Any JS symbol can be used inside the expression.

If the expression for the link count does not evaluate to a positive integer (0 not included), the behavior is undefined. Just as with predicates, the order of evaluation, as well as the number of times the expression is evaluated is undefined, and it is an error to rely on side effects caused by the evaluation of such an expression.

There are three set operators denoted '+', '-', and '^', which means union, complement, and intersection, respectively. The set operators have the least precedence inside selectors, and are evaluated left to right.

The set operators perform the same computation as in formal set theory.

Union: `a + b` matches all `a` elements and `b` elements.

Complement: `b - a / b` matches all `b` elements except from the `b` elements that have an `a` element as ancestor.

Intersection: `a / b ^ c / b` matches all `b` elements that have both an `a` element and a `c` element as ancestor.

Example

Some examples of style selectors are shown in code example 5.4.

```
1 //Selects all td elements that contain the text "content".
2 td[e.innerHTML == "content"]
3
4 //Selects all elements, except the div element with the id "speciel".
5 * - div#speciel
6
7 //Selects all tr elements that have the class "even" and has a td element as child
      that contains the text "someCell".
8 td[e.innerHTML == "someCell"] .(1) tr,even
9
10 //Selects all table elements that is nested inside two div elements while also having
      a tr element as decendant.
11 div / div / table ^ tr . table
```

Code example 5.4: *Four individual style selectors.*

5.2.2 Style Property

A style property is a property that can be accessed on a DOM element and represents an aspect of the visual appearance of DOM element. The style property is an abstraction in PDSS over existing CSS properties and some additional properties that the browser provides for each DOM element. Normally these two types of properties are accessed differently, but the style property unifies this through one language construct.

The production rule that corresponds to a style property is named `property` in the grammar. This production rule is shown in syntax example 5.5. A style property consists of a name, which is prepended with a `@` character. The name consists of at least one character, and all characters in the name are alphanumeric.

```
1 property
2   : '@' NAME
3   ;
4
5 NAME
6   : AplhaNum+
7   ;
```

Syntax example 5.5: *Style properties*

It is possible to access two types of styling properties: *CSS properties* and *rendered properties*. The set of rendered properties is defined by PDSS, and consists of four properties. PDSS does not define which properties exist in the set of CSS properties. To access a property from the rendered properties the name of one of the four rendered properties is used in the style property. Using anything else but one of these four names is considered as accessing a CSS property. The two types of properties are described in more details in the following:

Rendered properties: As the name indicates rendered properties reflect how a DOM element is rendered in the browser. The name is used in this report as no other name is known for the concept. The values of the rendered properties are calculated in the layout process of the browser (described in section 2.5). The rendered properties available are the rendered height, width, vertical position and horizontal position. These rendered properties are named `offsetHeight`, `offsetWidth`, `offsetTop` and `offsetLeft` respectively. An example of accessing a rendered property is shown in code example 5.6 line 2.

CSS properties: The CSS properties that are accessed are the ones in the style attribute of a DOM element. An example of accessing a CSS property is shown in code example 5.6 line 5.

To access CSS properties that contain the dash character (-), the name of the CSS property must be converted as follows: All dashes are removed and the characters that followed the dashes are capitalized. An example of accessing a CSS property with a dash character (-) is shown in code example 5.6 line 8.

The name of a style property is not checked for being a valid CSS property. This means that it is the developer's responsibility to ensure that the style property matches a valid CSS property. This allows a style property to access CSS properties from all versions of the standard, and also from future versions.

Style properties are only used in value selectors and style statements which are described in section 5.2.3 and 5.2.4 respectively.

Example

Some examples of style properties are shown in code example 5.6.

```
1 //A style property that access a rendered width.
2 @offsetWidth
3
4 //A style property that access the CSS property "color".
5 @color
6
7 //A style property that access the CSS property "background-color".
8 @backgroundColor
```

Code example 5.6: *Three individual style properties.*

5.2.3 Value Selector

A value selector is an expression that gets the value of a style property of one or more DOM elements. To specify the DOM elements a value selector uses a style selector.

There is no single production rule in the grammar that corresponds to a value selector. So the following part of the grammar (syntax example 5.7), shows the rule `expression` where the value selector corresponds to one of the alternatives in the rule. The other alternatives in the rule are left out. The alternative however only corresponds to a value selector when `property` is specified. A value selector consists of a style selector immediately followed by a style property. The syntax of these are described in section 5.2.1 and 5.2.2 respectively.

```
1 expression
2   : '#' '{' selector '}' property?
3   ;
```

Syntax example 5.7: *Value selector*

If the style selector of the value selector matches multiple elements, then a list with the values of the style properties is returned (see code example 5.8 line 2). If the style selector only matches one element, then the value of the style property for that element is returned (see code example 5.8 line 6).

A value selector can be used at the same locations where a JS expression can be used (see code example 5.8 line 9-12).

Example

Some examples of value selectors are shown in code example 5.8.

```
1 //Evaluates to the width of the div element with id "content".
2 ${div#content}@width
3
4 //Evaluates to a list of widths, one for each div element.
5 //(Only if multiple div elements exist in the DOM).
6 ${div}@width
7
8 //Using a value selector in JS.
9 if(${div#content}@color == "red")
10 {
11   //Do something
12 }
```

Code example 5.8: *Three individual value selectors.*

5.2.4 Style Statement

A style statement is a statement that assigns a value to a style property. To compute the value that is assigned to the style property a JS expression is used.

The production rule in the grammar that corresponds to a style statement is the rule named `statement`. This production rule is shown in syntax example 5.9. A style statement consists of a style property followed by the equality character and a JS expression. A semicolon ends the style statement. The syntax of a style property is described in section 5.2.2.

```
1 statement
2   : property '=' js_expression ';'
3   ;
```

Syntax example 5.9: *Style statements*

It is only possible to set CSS properties through the style property and not the rendered properties.

Style statements are used inside style blocks, and the effect of combining these are described in section 5.2.5. The syntax of PDSS allows style statements to be placed outside style blocks, but the behavior of doing this is undefined, and could yield a runtime error.

Example

Some examples of style statements are shown in code example 5.10.

```
1 //A style statement that uses a simple JS value to calculate the style property value.
2 @color = "#000000";
3
4 //A style statement that uses aritmethic in JS to calculate the style property value.
5 @width = 800 / 3;
6
7 //A style statement that uses a value selector to calculate the style property value.
8 @height = ${div#container}@offsetHeight + 100;
```

Code example 5.10: *Three individual style statements.*

5.2.5 Style Block

A style block is an abstraction over style statements and JS statements, and corresponds to a JS function in that:

- It has its own body of code surrounded by curly braces.

- The code in the body of the style block is not executed before explicitly stated.
- It is a first-class value.

The style block can contain multiple style- and JS statements in the body. Whereas a single style statement only specifies a single aspect of the visual appearance of a DOM element, the composition of style statements in a style block can constitute a whole style/theme. The execution of the individual style statements in the body can be controlled by control structures e.g. the conditional "if" statement of JS.

There is no single production rule in the grammar that corresponds to a style block. So the following part of the grammar (syntax example 5.11), shows the rule `expression` where the style block corresponds to one of the alternatives in the rule. The other alternatives in the rule are left out. A style block consists of JS statements enclosed in curly braces, and the whole block is prepended with an `@` character. Note that JS statements includes style statements, which, as later described in section 5.2.6, are important for the style block. The rule `JS` is used to match arbitrary JavaScript code where `;` is an empty JS statement. The syntax of a style statement is described in section 5.2.4.

```
1 expression
2   : '@' '{' js_statements '}'
3   ;
4
5 js_statements
6   : (js|rule|expression|statement|';')*
7   ;
```

Syntax example 5.11: *Style blocks*

A style block can be used in any place where a JS expression can be used. An example of creating a simple style block is shown in code example 5.12 line 1-2. An example of creating a style block that uses conditional JS statements is shown in code example 5.12 line 4-20.

Executing the code in a style block is later just called executing the style block. It is not possible to explicitly execute a standalone style block. The style block first has to be paired with a style selector to form a rule. When the formed rule is executed the style block is executed as a part of this. This will be described in more details in section 5.2.6.

It is possible to declare two style blocks and then combine them to one style block. This is achieved by using the `append` method on a style block. This method takes another style block as argument, and the body of that style block is appended to the body of the style block on which the `append` method is called. An example of appending a style block to another style block is shown in code example 5.12 line 22-26.

Example

Some examples of style blocks are shown in code example 5.12.

```
1 //A style block is a first class JS value
2 var styleBlock = @{@width=400;@height=400};
3
4 //A style block that uses conditional JS to declare style properties
5 @{
6   var largeSize = true;
7
8   if (largeSize)
9   {
10    @width = 400;
11    @height = 400;
12    @fontSize = 20;
13  }
14  else
```

```
15   {
16     @width = 200;
17     @height = 200;
18     @fontSize = 10;
19   }
20 }
21
22 //One style block is appended to another
23 var styleBlock1 = @{@width=600;};
24 var styleBlock2 = @{@height=200;};
25
26 styleBlock1.append(styleBlock2);
```

Code example 5.12: *Three individual code examples of using style blocks.*

5.2.6 Style Rule

A style rule is a statement that combines a style selector and a style block. Combining a style selector and a style block causes the DOM elements that match the style selector to be styled by the style block. This process is later called executing the style rule.

The production rule in the grammar that corresponds to a style rule is the rule named `rule`. This production rule is shown in syntax example 5.13. A style rule is started with the keyword `style`, which is followed by a style selector. Finally a style rule has JS statements enclosed in curly braces. This part of the style rule is treated as a style block. Note that the syntax of this part is equivalent to that of a style block with the exception of the `@` character, which is only used to distinguish a style block from regular JS. The syntax of style selectors and style blocks is described in section 5.2.1 and section 5.2.5 respectively.

```
1 rule
2   : 'style' selector '{' js_statements '}'
3   ;
```

Syntax example 5.13: *Rules*

A style rule can be used in any place where a JS statement can be used. An example of a simple style rule is shown in code example 5.14 line 1-5.

When a style rule is executed the whole style block is executed once for each of the DOM elements that match the style selector.

Executing a style block for a DOM element causes all assignments to style properties in the style block to be applied to the DOM element. The new values of the style properties can be accessed later using value selectors.

When executing a style rule it is not guaranteed that the style block is executed sequentially for all the matching elements. Other style blocks may be executed in between.

A style block in a style rule can depend on other style rules. This is the case if the style block contains a value selector that reads the value of a style property that has been set by another style rule. The dependency relation between a style block and a style rule is more precisely defined in definition 5.1.

Definition 5.1 (**Dependency**)
Given a style block B and a style rule A, B depends on the execution of A's style block for the DOM element e if there exists a style property p such that:

- A's selector matches, among others, e.
- A's style block has a style statement that sets p to some value.
- B has a style statement that uses a value selector to get the value of p or the corresponding rendered property*.
- The selector in the value selector in B matches, among others, e.

* The rendered property `offsetWidth` corresponds to the CSS property `width`, `offsetHeight` to `height`, `offsetLeft` to `left` and `offsetTop` to `top`.

If a style block depends on some style rules, then the style blocks of these style rules are executed first for the concerning DOM element. The order in which the style blocks in all the style rules are executed is undefined, except from the above mentioned condition. An example of a style block that depends on a style rule is shown in code example 5.14 line 7-16.

It is also possible to create a style rule by calling the built-in `createRule` function with a style selector and a style block as parameters. An example of creating a style rule with the function is shown in code example 5.14 line 18-22. There is also a `destroyRule` which has the opposite effect of `createRule`, which can be used to remove the effect of the created rule.

A style block in a style rule can depend on itself. This is the case if the style block depends on a style rule that has a style block that depends on the style rule of the first style block. If a style block depends on itself, then the styling process should halt and emit a runtime error describing the problem. How a style block can depend on itself is defined more precisely in definition 5.2. An example of a style block that depends on itself is shown in code example 5.14 line 24-29.

Definition 5.2 (**Self-dependency**)
A style block B in style rule A is depended on itself if there exists a sequence of style rules r such that:

- B depends on r_0, and
- the style block in r_0 depends on r_1, and
- the style block in r_1 depends on r_2, and
- ...
- the style block in r_n depends on A, where n is the number of rules in the sequence.

Example

Some examples of style rules are shown in code example 5.14.

```
1  //This style rule sets the height of all img elements to 400.
2  style img
3  {
4    @height = 400;
5  }
6
7  //The style block in the first rule (below this line) depends on the execution of the
       style block in the second rule for the div element with id "someID".
8  //So the style block of the second rule will be executed for the div element with id "
       someID" before the style block in the first rule is executed for any element.
9  style table
10 {
11   @width = ${div#someID}@offsetWidth;
12 }
13 style div
14 {
15   @width = 400;
16 }
17
18 //Creating a style rule by using the createStyle function.
19 //This is equivalent to the first example in this code example block.
20 var styleSelector = ${div};
21 var styleBlock = @{@width=400;};
22 createRule(styleSelector, styleBlock);
23
24 //A style rule that contains a style block that depends on itself.
25 //This will yield an error!
26 style div#first
27 {
28   @width = max(${div}@offsetWidth);
29 }
```

Code example 5.14: *Four individual code examples of using style rules.*

5.3 Library Functions

In addition to the language itself, there are a number of library functions tailored to be used with some of the language constructs. While the library functions presented here are already contained in the PDSS library, it is quite easy for a developer to create more of this kind of functions.

The following lists contain all the functions that are available in the library for PDSS.

The four functions in the following list are intended to be used inside predicates for selectors. That also means that all of these functions take a DOM element as their first (and possibly only) argument.

- The `child()` function returns an integer representing the index of the child, in relation to its siblings. This can be used for a number of things, like selecting every n'th child or a range of children.
- The `hover()` function returns true if the element itself or one of its descendants are hovered. This function makes it easy to react to user interaction.
- The `content()` function returns the text contained within an element. This function eases the task of selecting elements based on specific content.
- The `contains()` function takes an additional argument. The additional argument must be a regular expression formatted in the same way as a standard JS regular expression. If the text contained within the element matches the regular expression, the function returns true.

The three functions in the following list are intended to be used with value selectors that match multiple elements. It is however also possible to use them in regular JS expressions. Each of the functions takes a list of values as input. Remember that value selectors return a list of values when they match multiple elements.

- The `max()` and `min()` functions return the highest and lowest value in the list. They use the default comparison operators in JS (`<` and `>`) to find the value.
- The `avg()` function returns the average value of the values in the list. It does this by dividing the sum of the values with the number of values.

When using the three functions above, it is the developer's responsibility only to use them on values from style properties that are comparable/numeric.

Chapter 6

Implementation

Now that the syntax and semantics of the language have been defined, it is possible to do the actual implementation. This chapter will therefore be concerned with details about how PDSS is implemented.

As mentioned in chapter 5, there will be a library accompanying the generated code. The generated code in itself is useless as it relies completely on the library. In fact, the PDSS compiler mainly generates calls to the library. First, an overview of the solution is presented, and later some of the more notable implementation details will be presented and discussed.

6.1 Overview

The PDSS compiler is written in C#. The compiler basically consists of three parts: a lexer, a parser, and a code generator. The lexer and parser in the PDSS compiler was generated by ANTLR, with the grammar in appendix A.2 as input.

When this generated parser is fed a suitable file, it will generate a parse tree. In the code generation process, the parse tree is traversed. All the tokens that do not match rules concerned with PDSS will be outputted directly as is. The grammar has been constructed in a way that allows it to recognize JS tokens but not construct a sensible parse tree of JS, since the JS is only supposed to be output verbatim. In this prototype, the generated JS, as well as the JS in the library, is written only for use with the JS engine in Mozilla Firefox.

The library contains a number of elements that can roughly be partitioned into four components.

Style Selector prototype: Used for representing first class selector values, as well as computing the set of elements a given selector matches.

Style Block prototype: Used for representing first class style block values, and for computing the dependencies of the style block.

Style Table object: Used for maintaining the styling of elements. Creating a new rule can affect other rules, and therefore a number of previously created rules might need to be applied again, because they have changed. Therefore the style table needs to keep track of all existing rules.

Style selectors and style blocks will both be turned into expressions, because both can be put where JS expressions normally are allowed. A style rule will be turned into a JS statement, because style rules can only occur where JS statements are allowed.

The code generated by the compiler consists primarily of instantiations of the prototypes that represent style selectors and style blocks. These prototypes are defined in the PDSS library. The generated code also calls functions defined in the library. So when executing the generated JS, the style selector and style block is both represented by object instances. The style rule is basically represented by a call to the style table, where the arguments of that call are an instance of the selector of the rule and an instance of the style block of the rule.

As described in chapter 5, the style selector or style block alone does not cause anything to be styled. The style rule can however cause elements to be styled, in the case where the style rule actually matches some elements and sets some properties. Whenever the flow of execution passes a style rule, a call to the function `createRule` will occur, and the style rule will be registered in the style table.

The style table basically consists of a list of rules that have been registered/created. This style table will initially be empty, and will grow only when the flow of execution passes a style rule construct, or when a first class style selector is paired with a first class style block to create a rule with `createRule`. Whenever a new rule is registered and added to the style table, all the rules in the style table will be applied.

The style table is only applied when some events occurs. The events are for instance then a new rule is being registered or the mouse moves. Before all the styles are applied, all the current styles are reset. Then, all style selectors and style blocks are evaluated, which means that the style selector computes all the elements it matches, and the style block computes the style values and its dependencies. Then, a dependency graph over all element-property pairs is generated, and is immediately turned into a topological ordering. If the graph contains any cycles, this will be detected, and result in a runtime error. Lastly, the styles are applied in the found order.

Applying a style consists of computing the style values, and then assigning these computed style values to properties on selected elements.

6.2 Details

In the following sections, some of the more notable implementation details will be presented.

6.2.1 Selectors

The most important task of selectors is to compute the set of elements the selector represents. This task is divided into subtasks. There is one subtask for each operator in the selector. As described in chapter 5, selectors have two kinds of operators, match and set operators.

The match operators can be thought of as taking a tree as input, and then outputting all the trees that the operator matches. Each of these outputted trees will then be tested to see if the predicates are true, and if so, the trees are fed into the next operator in the chain, until there are no more operators.

As an example, the selector `div / p` will now be evaluated with the tree in figure 6.1 as the document tree. The selector will select all `div` elements in the document tree, and then it will select all descendants of these `div` elements. The result will be the set: $\{p(5), p(8), p(10)\}$.

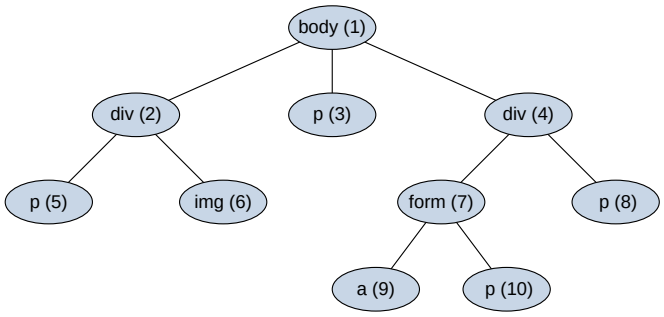


Figure 6.1: *A simple tree.*

The algorithm is:

- Add all elements of the document tree to the set A.
- For each element r in A, test if r is a `div` element, if so, add r to the set B. B then contains $\{div(2), div(4)\}$.
- For each element s in B, find all descendants of s and add them to the set C_s. $C_{div(2)}$ then contains $\{p(5), img(6)\}$ and $C_{div(4)}$ contains $\{form(7), p(8), a(9), p(10)\}$.
- Combine all the sets C_n where n is in B, into C.
- For each element t in C, test if t is a `p` element, if so, add t to the set D. D then contains $\{p(5), p(8), p(10)\}$.
- return D.

This approach with evaluating one operator at a time is very easy to perform, but it can result in some huge intermediate results. For instance, in the above example, there have been created six sets (A, B, C1, C2, C, D), in total containing 26 elements (9+2+2+4+6+3).

The first operation of finding all the `div` elements can clearly be optimized. Instead of putting everything into a set, and first then begin filter the elements by type, the elements can simply be tested while the tree is iterated. This will cause the set A to never be created. In fact, testing predicates and the type of the element can always be done before inserting the element into an intermediate result. This will in many cases reduce the intermediate result, but it never causes it to be larger.

If the elements are eliminated as early as possible, the steps would look like:

- For each element r in the document tree, test if r is a `div` element, if so, add r to the set B.
- For each element s in B, and for each descendant t of s, test if t is a `p` element, if so, add t to the set C_s. $C_{div(2)}$ then contains $\{p(5)\}$ and $C_{div(4)}$ contains $\{p(8), p(10)\}$.
- Combine all the sets C_n where n is in s, into C
- return C.

Now we are down to 4 sets (B, C1, C2, C), but only a total count of 8 elements (2+1+2+3). The actual implementation reduces this even more, it evaluates an entire chain of match operators at once, no intermediate results will be created. For every single element matched at a given step in the chain of operators, the next operator will be applied immediately, recursively. An example:

- For each element r in the document tree, test if r is a `div` element, if so, pass r to the next operator in the chain. When passing the r to the next operator in the chain, each descendant s of r are tested for being a `p` element, if so, s is added to the set A.
- return A.

So when an item is added to the result in the implementation, the call stack will contain all the operators of the selector (assuming that only match operators were used in the selector).

Examples of input and output code for selectors can be seen in the code examples 6.1 and 6.2 respectively.

```
1 var predicate = false;
2 var sel = ${
3   img .(1) div[predicate == false] /(1) p
4 };
```

Code example 6.1: *PDSS example of a selector literal, the generated output can be seen in code example 6.2.*

```
1 var predicate = false;
2 var sel = new Selector(
3   collect,
4
5   new SelectorTerm(
6     children, // operator
7     null, // level
8     "IMG", // tag
9     null, // pred
10    new SelectorTerm(
11      parent, // operator
12      (1), // level
13      "DIV", // tag
14      function(e){ return ( (  predicate == false ) );}, // pred
15      new SelectorTerm(
16        children, // operator
17        (1), // level
18        "P", // tag
19        null, // pred
20        null
21      )
22    )
23  ),
24
25  null
26 );
```

Code example 6.2: *Generated JS output of the PDSS selector literal in code example 6.1.*

As can be seen in code example 6.2, `SelectorTerm` contains a recursive structure of itself, which is basically a linked list of operators.

When the outer (first) of these `SelectorTerm`s is evaluated, it will have access to the subsequent `SelectorTerm`s. As soon as one element has been found to be in the intermediate result of the outer selector, that single element is immediately given as input to the next `SelectorTerm` operator. So each operator recursively calls the next operator in the sequence while it enumerates the intermediate elements.

As mentioned earlier, there are two kinds of operators, match operators and set operators. All the set operators are first applied when the match operators they depend on has been computed. The set operators work in a similarly recursive manner as the match operators.

When a selector is fully evaluated it returns a list of elements as the result.

6.2.2 Style Blocks

Apart from representing style blocks, this component has two other important tasks. The first is holding a list of functions to be executed when the style block is to be applied. These functions just contain the JS that was actually written inside the style blocks. When a style block is created, it only contains one of these functions. The reason there is a list of these functions, is because it is possible to append style blocks to each other.

The second important task for style blocks is to calculate the dependencies a style block has. As discussed earlier, one style block may rely on other style blocks being evaluated in advance. Whenever a property is assigned to in a style block, the compiler turn this into a function call, which allows us to register that the property is set to a given value. See code example 6.3 and code example 6.4 for input and output source of the compiler, respectively.

```
1 var coolStyle = @{
2   if (haha)
3   {
4     @border = "red thick dashed";
5
6     @color = ${ para }@color;
7   }
8 };
```

Code example 6.3: *PDSS example of a style block literal, the generated output can be seen in code example 6.4.*

```
1  var coolStyle = new StyleBlock(
2    function(sb){
3      if (haha) {
4        sb.addProperty(new Property(
5          'border',
6          "red thick dashed"
7        ));
8
9        sb.addProperty(new Property(
10         'color',
11         new ValueSelector(SELECTOR(PARA),'color', sb).value
12       ));
13     }
14   }
15 );
```

Code example 6.4: *Generated JS output of the PDSS style block literal in code example 6.3.*

Based on the output as can be seen in code example 6.4, the function starting on line 2 basically contains the statements of the style block in code example 6.3.

Line 4 and 9 represent the assignment of the properties. When this function gets executed (initiated by the style table), the parameter `sb` (line 2) will reference the `StyleBlock` instance itself (created on line 1). As can be seen on line 11, the constructor of the `ValueSelector` is handed this instance. What the `ValueSelector` does, apart from computing the value to assign to the style property, also adds a dependency to the `StyleBlock` instance through `sb`.

So basically, when a style block is evaluated, its dependencies are computed as well as the values for its style properties. Pseudo code that describes the structure of a style block can be seen in code example 6.5.

```
style block:
  functions: [function],
  properties: [
    prop_name: string,
    value: string,
    dependencies: [
      element: DOM_ref,
      prop_name: string,
    ],
  ],
```

Code example 6.5: *Pseudo structure of a style block. Members in square brackets represent repeated structures.*

A style block has two lists, one of functions and one of properties. Each item in the list of properties contains three values; `prop_name`, `value`, and a list of dependencies. Each item in the list of dependencies consists of two values, `element` and `prop_name`.

The `properties` list basically represents the dependencies in terms of what element-property pair depends on what element-property pair. These dependencies are used by the style table to generate a dependency graph, which will be explained in section 6.2.3.

6.2.3 Style Table

The style table is the entity that keeps all styles up to date, and ensures that the style rules are applied in an order that ensures the document is styled in the way the author of the styling constructs intended.

When the developer decides to apply a style, he either forms a rule by combining a first class selector with a first class style block by calling `create_rule(<selector>, <style block>);`, or he declares a rule `style <selector> <style block>`.

When the flow of execution reaches the creation or declaration of a rule, that rule will be added to a list of rules called the style table. This style table will start out as being empty, and then grows as the execution of the application takes place. It is possible to remove styles from this table again if the developer decides to do so.

Whenever a rule is added to the style table, all styles are re-applied immediately. This is necessary in case where one of the existing styles relies on properties set by the recently added style.

Every time the style rules in the style table need to be applied, there have to be computed a dependency graph as discussed earlier in section 4.3. The nodes of the graph are identified by two values, one is a DOM element, and the other is a string representing the name of a CSS property.

The first step of generating a dependency graph is to evaluate the selector and the style block for all rules in the table. For every rule, all element-property pairs are inserted as nodes in the graph, if they not already exist. At the same time, the dependencies are added. Once this is done, the graph is completed.

A topological sorting of the nodes in the graph is performed, which will yield the order in which to style the elements. This causes every style rule to be applied $e * p$ times, where e is the number of elements matched by the selector of that rule, and p is the number of properties set in the style block of the rule. This is because that is how many times that rule is referenced in the graph.

The style block is however not evaluated this many times, because it originally cached the style values while it was computing the dependencies.

When the mouse is moved, the cursor could hover a new element, and this could cause a predicate to change its value. Therefore, the styles will be redone when the mouse hovers

a new element. There are a lot of other events that should cause the style to be redone, for instance any change of the DOM. But no other events are registered in the implementation.

Chapter 7

Test

A prototype of PDSS has been implemented as described in the previous chapter. In this chapter, the prototype will be tested. First, it is tested whether PDSS can implement all the use cases presented in section 2.3. These use case implementations in PDSS are then compared with the use case implementations in the other existing styling solutions, which were described in section 2.4. The test is also used to test the functionality of PDSS, and the result of this will be presented at last in the chapter.

7.1 Use Case Implementations

In section 2.3 some use cases were presented that represented separate features of web pages. In this section, the use cases will be implemented in the prototype of PDSS. For each of the implementation there will be a description of the implementation, a code example of the HTML document that is styled, a code example of the PDSS code, and a screenshot of the resulting visual styling of the HTML with the PDSS. These use case implementations will later be compared to the implementations of the use cases in the existing solutions.

7.1.1 Multi Level Menu

This use case is about styling an item differently if the item contains sub items. The menu is made from a HTML list (`ul` and `li` elements) containing anchor elements. When a submenu is present, a new menu is nested inside.

The HTML that is styled is shown in code example 7.1. The PDSS code used for the implementation is shown in code example 7.2, and the following line numbers will refer to that code example.

The source consists of four style rules. Line 1-5 set up the container for the menu. The menu is selected by its id and styled through assigning style values to style properties. Line 7-12 make sure that the default styles of the browser are overridden. This rule make use of the union operator to select both `ul` and `li` elements. The actual use case is implemented on the remaining lines. The rule on line 14-17 selects the `li` elements that have a `ul` element as child (using the ancestor operator), and gives these elements a red border. When a `li` element has an `ul` as child, then it contains sub-items. Just to justify the use case, the item that contains the sub items has been styled too (line 19-24). This is done by selecting the `a` elements that are one step to the left of (immediately preceding) an `ul` element. As can be

seen in code example 7.1 on line 4-5, the title of the submenu is contained in the preceding `a` element, and therefore this is styled.

Figure 7.1 shows how the multi level menu use case looks like when styled with PDSS.

```
1     <ul id="menu">
2       <li><a href="#">1</a></li>
3       <li>
4         <a href="#">2</a>
5         <ul>
6           <li><a href="#">2.1</a></li>
7           <li><a href="#">2.2</a></li>
8           <li><a href="#">2.3</a></li>
9         </ul>
10      </li>
11      <li><a href="#">3</a></li>
12      <li><a href="#">4</a></li>
13      <li>
14        <a href="#">5</a>
15        <ul>
16          <li><a href="#">5.1</a></li>
17          <li><a href="#">5.2</a></li>
18        </ul>
19      </li>
20      <li><a href="#">6</a></li>
21      <li><a href="#">7</a></li>
22    </ul>
```

Code example 7.1: *HTML for Multi Level Menu use case implementation.*

```
1  style *#menu
2  {
3    @width = "100px";
4    @border = "medium black solid";
5  }
6
7  style ul + li
8  {
9    @margin = "0";
10   @padding = "0";
11   @listStyle = "none";
12 }
13
14 style ul .(1) li
15 {
16   @border = "medium red solid";
17 }
18
19 style ul <(1) a
20 {
21   @display = "block";
22   @width = "100%";
23   @backgroundColor = "red";
24 }
```

Code example 7.2: *Multi Level Menu in PDSS.*

7.1.2 Submenu Positioning

This use case is about positioning a drop-down submenu next to the menu item from which it originates. The submenu positioning use case has been implemented by styling `ul` and `li` elements. An `ul` element represents the main menu, which has a `li` element for each menu item. For each menu item, an `ul` element is present if the menu item has a submenu. The `li` elements in this `ul` element are menu items in the submenu.

The HTML that is styled is shown in code example 7.3. The PDSS code used for the implementation is shown in code example 7.4, and the following line numbers will refer to that code example.

First on line 1-5, all `li` elements in the menu are styled such that they look like boxes. This is achieved by a style rule that has a style selector that selects all `li` elements that

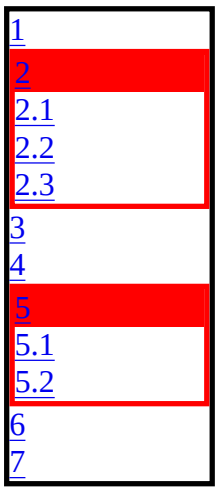


Figure 7.1: *Screenshot of Multi Level Menu use case implementation in PDSS.*

are descendants of the `ul` element with the id "menu". To create the box layout, the two CSS properties `border` and `listStyle` are set. Setting `listStyle` to "none" ensures that the list items not will have any bullet points. Then, the size of each menu item in the main menu is specified to a fixed size on line 7-11. Note that the style selector used in the style rule is identical to the one used in the previous style rule, with one exception. For the descendant operator (`/`), it is specified that the maximum number of links are one. By doing that, only the menu items in the main menu are selected and not the menu items in the submenu, which sizes we do not care about. On line 13-21, all submenus are hidden, and they are positioned beside the menu item in which they are contained. The CSS property `visibility` is used to hide the elements, and the CSS property `display` is set to "none", which temporary removes the elements from the frame tree (see section 2.5.2). When setting the CSS property `position` to "relative", an element is positioned relative to the top-left corner of the parent element when setting the CSS properties `left` and `top`. The reason for setting the CSS property `top` to -21 is that the submenus should be placed above the text "Menu item #", which have a height of 21 pixels. If each menu item in the main menu not had a fixed size, a value selector could have been used to get the size.

```
1  <ul id="menu">
2    <li>
3      Menu item 1
4      <ul>
5        <li>Sub menu 1</li>
6        <li>Sub menu 2</li>
7        <li>Sub menu 3</li>
8      </ul>
9    </li>
10   <li>
11     Menu item 2
12     <ul>
13       <li>Sub menu 1</li>
14       <li>Sub menu 2</li>
15       <li>Sub menu 3</li>
16     </ul>
17   </li>
18   <li>
19     Menu item 3
20     <ul>
21       <li>Sub menu 1</li>
22       <li>Sub menu 2</li>
23       <li>Sub menu 3</li>
24     </ul>
```

```
25    </li>
26 </ul>
```

Code example 7.3: *HTML for Submenu Positioning use case implementation.*

The rest of the code is concerned with styling when a menu item in the main menu is hovered. First, it is defined on line 23-26, that hovered menu items in the main menu should have a silver background color. The style selector used in this style rule is the same as that used to style the menu item, with the exception of the predicate at the end. In the predicate, the JS `hover` function from the PDSS library is called, which returns true when the element given as input is hovered. Then, it is defined on line 28-33 that any submenu in a menu item should be visible when the menu item is hovered. The style selector in this style rule is the same as the one used in the previous style rule, except that an extra descendant operator selects the submenu (`ul` element) of the menu item rather than the menu item.

Figure 7.2 shows how the submenu positioning use case looks like when styled with PDSS.

```
 1 style ul#menu / li
 2 {
 3   @border = "1px solid black";
 4   @listStyle = "none";
 5 }
 6
 7 style ul#menu /(1) li
 8 {
 9   @width = 100;
10   @height = 21;
11 }
12
13 style ul#menu /(2) ul
14 {
15   @visibility="hidden";
16   @display="none";
17   @position="relative";
18   @padding=0;
19   @left=100;
20   @top=-21;
21 }
22
23 style ul#menu /(1) li [hover(e)]
24 {
25   @backgroundColor="silver";
26 }
27
28 style ul#menu /(1) li [hover(e)] / ul
29 {
30   @visibility="visible";
31   @display="block";
32   @backgroundColor="transparent"; // no apparent effect , but fixes a bug
33 }
```

Code example 7.4: *Submenu Positioning use case in PDSS.*

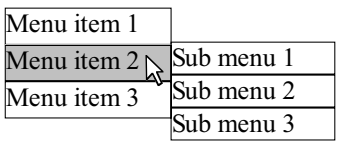


Figure 7.2: *Screenshot of the implemented Submenu Positioning use case.*

7.1.3 Cell Content

This use case is about styling table cells based on their content. The table is a standard HTML table with arbitrarily chosen numbers in the cells.

The HTML that is styled is shown in code example 7.5. The PDSS code used for the implementation is shown in code example 7.6, and the following line numbers will refer to that code example.

The first rule on line 1-4 just draws borders on the cells. The second rule on line 6-9 selects all `td` elements with a predicate on the `td`. Inside the predicate (that consists of a JS expression), the `content` function from the PDSS library is used to retrieve the content of the `td` element. This content is then compared to the value 5 using the "less than" JS comparison operator. The fetched content will automatically be converted to a numerical value according to the semantics of JS, and the operator will yield true if the converted value is less than 5. That means that the selector will match all `td` elements of which the content represents a value less than five. These elements are then given a pink background.

Figure 7.3 shows how the cell content use case looks like when styled with PDSS.

```
1     <table id="first">
2       <tr><td>1</td><td>23</td><td>3</td><td>4</td></tr>
3       <tr><td>5333</td><td>6</td><td>7</td><td>8</td></tr>
4       <tr><td>9</td><td>0</td><td>33</td><td>27</td></tr>
5       <tr><td>3</td><td>4</td><td>5</td><td>63</td></tr>
6       <tr><td>7</td><td>38</td><td>9</td><td>0</td></tr>
7     </table>
```

Code example 7.5: *HTML for Cell Content use case implementation.*

```
1 style table / td
2 {
3   @border = "black thin solid";
4 }
5
6 style table / td[content(e) < 5]
7 {
8   @backgroundColor = "hotpink";
9 }
```

Code example 7.6: *Cell Content in PDSS.*

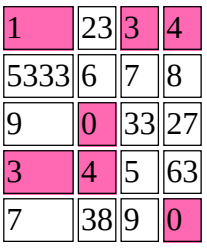


Figure 7.3: *Screenshot of Cell Content use case implementation in PDSS.*

7.1.4 Aspect Maintenance

This use case is about having an element that maintains its aspect when it is resized. In this use case there are two `div` elements. The first is the *content* box, and the second is the element that is to be *auto scaled* once the width of the window changes.

The HTML that is styled is shown in code example 7.7. The PDSS code used for the implementation is shown in code example 7.8, and the following line numbers will refer to that code example.

The aspect of the auto scaled element is set on line 1, which does not involve any of the PDSS features. Line 3-7 styles the content box that is meant to have a fixed width. Line 9-15 styles the image that is supposed to scale based on the available space. Line 11 grabs the available width through the JS expression `document.body.clientWidth`, which is available through the DOM. The width of the fixed width content box is then retrieved through a value selector. The available space is subtracted the value of the value selector as well as the integer `4`. The subtraction of `4` is for taking the width of the borders into account. Line 12-13 then sets the dimensions of the auto scaled image, while taking the aspect ratio of the image into account. The height is computed with the JS multiplication operator.

Figure 7.4 and 7.5 show how the aspect maintenance use case looks like when styled with PDSS.

```
1     <div id="fixed">
2     Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla posuere pharetra
          tortor sit amet fermentum.
3     </div>
4
5     <div id="scale">
6     Donec sit amet urna at nisl mattis tincidunt. Ut vitae urna magna.
7     </div>
```

Code example 7.7: *HTML for Aspect Maintenance use case implementation.*

```
1  var HeightWidthRatio = 1.5;
2
3  style *#fixed
4  {
5    @width = "200";
6    @border = "thin blue solid";
7  }
8
9  style *#scale
10 {
11   var width = document.body.clientWidth - ${ *#fixed }@offsetWidth - 4;
12   @width = width;
13   @height = width * HeightWidthRatio;
14   @border = "thin red solid";
15 }
```

Code example 7.8: *Aspect Maintenance in PDSS.*

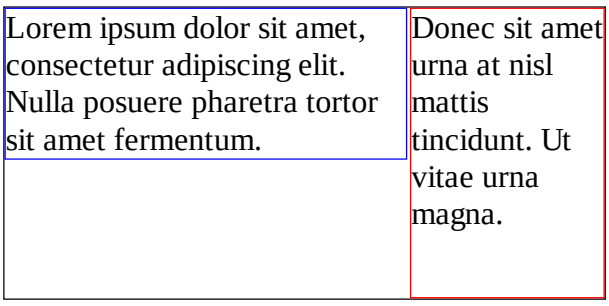


Figure 7.4: *Screenshot of Aspect Maintenance use case implementation in PDSS.*

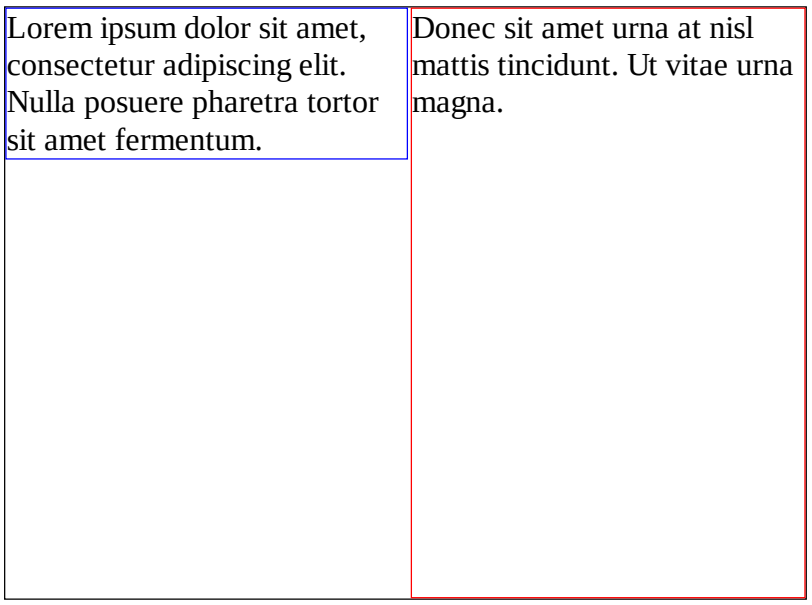


Figure 7.5: *Screenshot of Aspect Maintenance use case implementation in PDSS, when the window is scaled.*

7.1.5 Column Layout

This use case is about making a column layout where all the columns expand to the height of the tallest column. The implementation consists of three `div` elements. One of those elements is the *container*, which holds the two other `div` elements. These two `div` elements are considered to be the columns in the layout. Each of the `div` elements have an id attribute such that they are easy to select.

The HTML that is styled is shown in code example 7.9. The PDSS code used for the implementation is shown in code example 7.10, and the following line numbers will refer to that code example.

On line 1-18, three style rules are used to set up the three `div` elements in terms of width, background color, and floating direction. To select the elements, the selectors in the style rules use the shorthand notation for selecting elements based on id. Note that the sum of the widths of the two `div` elements that represent the columns is almost the same as the width of the container `div` element. The floating direction is specified by the CSS property `float`, and causes the first `div` element inside the container to float to the left side of the container, and the second `div` element inside the container to float to the right side of the container. This causes the `div` elements inside the container to be placed side by side, like two columns. Floating is a concept from CSS that causes an element to float from its original position to the left or right text boundary. Line 20-24 style both columns. Both columns should have the same style, so the selector selects both columns by using the union selector operator between the two selectors that select each of the columns. On line 22, the rendered heights of the two columns are stored in a variable `heights` by using a value selector. The selector in the value selector is the same as the selector of the style rule, so we are styling the columns based on the columns own style. The value that is returned by the value selector and stored in the variable `heights` is a list of heights. On line 23, the

height of both columns is set to the height of the tallest column. The value is calculated by using the PDSS library function `max` on the list of columns heights in the variable `heights`.

Figure 7.6 shows how the column layout use case looks like when styled with PDSS.

```
1     <div id="container">
2
3       <div id="left">
4       Donec sit amet urna at nisl mattis tincidunt. Ut vitae urna magna.
5       </div>
6
7       <div id="right">
8       Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla posuere pharetra
            tortor sit amet fermentum.
9       </div>
10
11    </div>
```

Code example 7.9: *HTML for Column Layout use case implementation.*

```
1  style *#container
2  {
3    @width = "210px";
4  }
5
6  style *#left
7  {
8    @float = "left";
9    @width = "100px";
10   @backgroundColor = "red";
11 }
12
13 style *#right
14 {
15   @float = "right";
16   @width = "100px";
17   @backgroundColor = "blue";
18 }
19
20 style *#left + *#right
21 {
22   var heights = ${ *#left + *#right }@offsetHeight;
23   @height = max( heights );
24 }
```

Code example 7.10: *Column Layout in PDSS.*

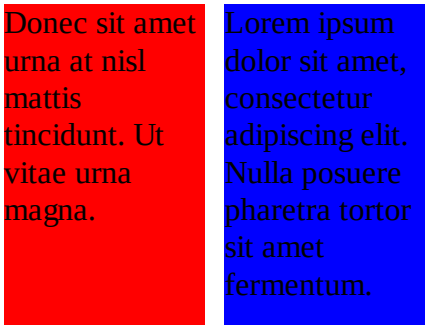


Figure 7.6: *Screenshot of Column Layout use case implementation in PDSS.*

7.1.6 Table Alignment

This use case is about aligning the columns of two tables, such that the columns of one table have the same widths as the columns of another table. The use case implementation

consists of two standard HTML tables. A snippet of text is inserted to separate the two tables.

The HTML that is styled is shown in code example 7.11. The PDSS code used for the implementation is shown in code example 7.12, and the following line numbers will refer to that code example.

```
1     <table id="second">
2       <tr><td>Lorem</td><td>ipsum</td><td>dolor</td></tr>
3       <tr><td>sit</td><td>amet,</td><td>consectetur</td></tr>
4     </table>
5
6     Ordinary text
7
8     <table id="first">
9       <tr><td>ante</td><td>ultricies</td><td>placerat</td></tr>
10      <tr><td>eu</td><td>ristique</td><td>lorem.</td></tr>
11      <tr><td>Quisque</td><td>luctus</td><td>rhoncus</td></tr>
12      <tr><td>sem,</td><td>ac</td><td>mattis</td></tr>
13    </table>
```

Code example 7.11: *HTML for Table Alignment use case implementation.*

```
1  style *
2  {
3    @padding = "0";
4  }
5
6  style table
7  {
8    //@borderCollapse = "collapse";
9    @border = "thin black solid";
10 }
11
12 style *#first / td
13 {
14   @backgroundColor = "red";
15 }
16
17 style *#second / td
18 {
19   @backgroundColor = "blue";
20 }
21
22 function alignCol(column)
23 {
24   style table / tr / td[child(e) == column]
25   {
26     var widths = ${ table / tr / td[child(e) == column] }@offsetWidth;
27     @width = max( widths );
28   }
29 }
30
31 alignCol(1);
32 alignCol(2);
33 alignCol(3);
```

Code example 7.12: *Table alignment in PDSS.*

Line 1-20 remove all padding of all elements, give the tables some borders, and style the cells of the two tables with red and black backgrounds. This is all done through four trivial rules. Line 22-29 define a JS function containing a style rule. This function is created to make it convenient to choose what columns should get the same width. The parameter to this function defines what column to be styled. Line 24-28 select all `td` elements (cells) in a given column. The cells of the column are found through the use of a predicate. The predicate makes use of the PDSS library function `child`, which returns an integer corresponding to the position of the `td` element. This position corresponds to the column where the given `td` happens to be the cell. Therefore, the function parameter `column` is compared to the result of the `child` function with the JS equality operator. If the result is true, then the predicate

is true and the corresponding `td` element is selected. Notice that the selector selects all cells in this column on both tables. By using a value selector, all cells in the given column of both tables are selected, and a list of their computed widths is retrieved (line 26). The list of values is assigned to the JS variable `widths`. The variable `widths` now contains a list of all the widths of the cells of the column designated by the variable `column`. The larges value in the list is found on line 27 and assigned to the `@width` property. That causes all the cells in the column of both tables to be styled. Their widths are set to be as wide as the widest of the cells across both tables. The function is then called once for each column on line 31-33, which causes all of the columns of the two tables to be aligned with each other.

Figure 7.7 shows how the table alignment use case looks like when styled with PDSS.

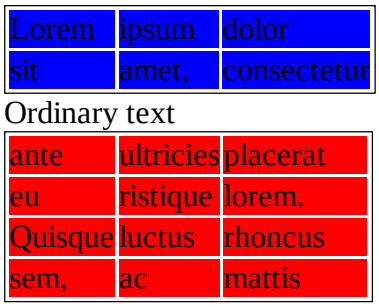


Figure 7.7: *Screenshot of Table Alignment use case implementation in PDSS.*

7.1.7 Ordinal Selection and Selection Intersection

The ordinal selection and selection intersection use cases are merged into one implementation because each of them is simple, and it makes good sense to use them together. The ordinal selection use case is about selecting every n'th element, and the selection intersection use case is about finding the intersecting elements of two selectors. The ordinal selection use case is represented by selecting rows and columns in a table. The selection intersection use case is represented by intersecting these two selections.

The HTML that is styled is shown in code example 7.13. The PDSS code used for the implementation is shown in code example 7.14, and the following line numbers will refer to that code example.

Two style rules are used to perform some styling on line 1-13, with the only purpose of making the table look good. This code is not very interesting. On line 15-18 and 20-23, every third row and second column in the table, respectively, are styled by means of different background colors. To select the cells in the rows (line 15) and columns (line 20), the function `child` from the PDSS library is called inside a predicate, with the `tr`/`td` element as argument (through the *e* symbol). The `child` function returns the index of the element (in relation to the parent element). This value is used with the modulus operator in JS, to test if the row is in every third row and if the column is in every second column, which is the case then the reminder of the division is 0.

The part of the implementation until now represented ordinal selection, and the next part of the implementation covers selection intersection. On line 25-28, a new style rule is created. The style selector of this style rule consists of the selectors on line 15 and 20, but separated by an `^` symbol. The `^` symbol is the intersection operator in PDSS, so the

intersection of the elements that the two selectors match is selected in the new selector. Finally, the elements (cells) matched by the selector in the new rule (every third row and every second column) have their background color set to green (line 27).

Figure 7.8 shows how the ordinal selection and selection intersection use case looks like when styled with PDSS.

```
1 <table>
2   <tr>
3     <td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td>
4   </tr>
5   <tr>
6     <td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td>
7   </tr>
8   <tr>
9     <td>m</td><td>n</td><td>o</td><td>p</td><td>q</td><td>r</td>
10  </tr>
11  <tr>
12    <td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td>
13  </tr>
14  <tr>
15    <td>y</td><td>z</td><td>a</td><td>b</td><td>c</td><td>d</td>
16  </tr>
17  <tr>
18    <td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td>
19  </tr>
20 </table>
```

Code example 7.13: *HTML for Ordinal Selection and Intersection Selection use case implementation.*

```
1  style table
2  {
3    @border = "1px solid black";
4    @width = 150;
5    @height = 150;
6    @borderCollapse = "collapse";
7    @borderSpacing = 0;
8  }
9
10 style td
11 {
12   @textAlign = "center";
13 }
14
15 style tr[child(e)%3==0] / td
16 {
17   @backgroundColor="blue";
18 }
19
20 style td[child(e)%2==0]
21 {
22   @backgroundColor="red";
23 }
24
25 style tr[child(e)%3==0] / td ^ td[child(e)%2==0]
26 {
27   @backgroundColor="green";
28 }
```

Code example 7.14: *Ordinal Selection and Intersection Selection use case in PDSS.*

7.1.8 Centering

This use case is about centering one element inside another element. The centering use case has been implemented as one `div` element that is centered inside another `div` element.

The HTML that is styled is shown in code example 7.15. The PDSS code used for the implementation is shown in code example 7.16, and the following line numbers will refer to that code example.

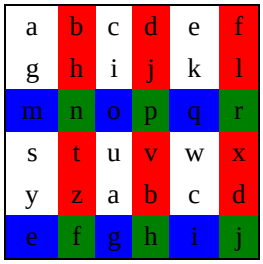


Figure 7.8: *Screenshot of the implemented Ordinal Selection and Intersection Selection use case.*

The first style rule styles the outer `div` element (line 1-6). This code is not so interesting. The second style rule styles the inner `div` element, which is the centered one (line 8-16), and the code for this is the most interesting. The CSS property `position` of the inner `div` element is set to `relative` (line 11), such that later setting the CSS properties `left` and `top` means relative to the parent element (the outer `div` element).

To center the inner `div` element inside the outer `div` element, the width and height of both `div` elements are needed. When having these values, the left position of the inner `div` element is calculated as $(outerDivWidth - innerDivWidth)/2$, and the top position of the inner `div` element is calculated as $(outerDivHeight - innerDivHeight)/2$. These two calculations take place on line 14-15, where the height and width of the outer `div` element is accessed through value selectors. As the inner `div` element has a fixed width of 100, as declared on line 12, this value is used directly in the computation, whereas the rendered height (`offsetHeight`) is accessed through a value selector. By using the rendered height, the inner `div` element is ensured to be centered even though the text content of the `div` element is changed, which often is the case in RIAs.

Figure 7.9 shows how the centering use case looks like when styled with PDSS.

```
1 <div id="container">
2   <div id="centeredDiv">
3     The div element that this text is inside, is centered into another div element.
4   </div>
5 </div>
```

Code example 7.15: *HTML for Centering use case implementation.*

```
1  style div#container
2  {
3    @width=300;
4    @height=200;
5    @backgroundColor="silver";
6  }
7
8  style div#centeredDiv
9  {
10   @backgroundColor="gray";
11   @position="relative";
12   @width = 100;
13
14   @left=(${div#container}@width-100)/2;
15   @top=(${div#container}@height-${div#centeredDiv}@offsetHeight)/2;
16 }
```

Code example 7.16: *Centering use case in PDSS.*

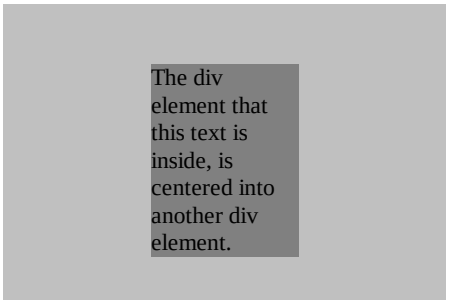


Figure 7.9: *Screenshot of the implemented Centering use case.*

7.2 Comparison of Use Case Implementations

All the use cases have successfully been implemented in PDSS. In section 2.4, each of the use cases were implemented in one of the existing styling solution. In this section, implementations of the use cases in PDSS will be compared to the implementation of the same use case in the existing styling solution it was implemented in. Not all the use cases will be compared, only the ones that shows something interesting. Like in the analysis, the implementations will be compared in groups of features (element selection, inter-element constraints and expressions). The comparison is used to conclude whether or not PDSS is of benefit to web development with regards to the feature that the use case represents. Some of the parameters that will be taken into consideration in the comparison are expressive power, code length and readability.

7.2.1 Element Selection

The ordinal selection and the multi level menu use cases are just as easy to implement in the existing solutions as in PDSS. For instance, to perform ordinal selection in CSS 3 (see code example 2.12 on page 28) and PDSS, the two following pieces of code are used: `tr:nth-child(3n)` in CSS 3 and `tr[child(e)%3==0]` in PDSS. What is important here is that `nth-child` is one of the predefined constraints in CSS 3, and it is not possible to create user-defined constraints, unlike PDSS. In PDSS, if the developer wants to select the rows that correspond to prime numbers, he can use a `isPrime` function inside a predicate. which returns true if the argument is a prime. The row index is then given as argument. The predicates in PDSS can therefore express any possible condition, unlike the limited expressiveness of CSS 3. PDSS also makes it intuitive to express conditions, as all features of the UPL is available, and the developer does not need to use some predefined functionality in predicates.

The cell content use case that was implemented in XSLT (see code example 2.10 on page 27) requires a lot more code than the implementation in PDSS. In PDSS, it is only necessary to create the two selectors `td[content(e) < 5]` and `td[content(e) > 5]` to split the cells into two groups. In XSLT, all the cells must first be selected, and then afterwards filtered by a lengthy choose-when-otherwise construct. So PDSS has a clear advantage, as it can express the condition in a concise manner. With regards to powerfulness it is a tie between XSLT and PDSS.

The Selection intersection use case, which was also implemented in XSLT (see code example 2.13 on page 29), also required a considerably amount of code in contrast to PDSS.

In PDSS an intersection selection was simply created by using the built-in intersection operator between selectors.

While it is possible to express arbitrary predicates, it is not possible to express arbitrary selection operators in the PDSS prototype. That puts a limit on how complex a tree it can match. At most one of the other solutions is more powerful in this regard, namely DSSSL. We have however been unable to determine its powerfulness.

7.2.2 Interelement Constraints

With regard to the submenu positioning use case, which was implemented in the existing solution SASS (see code example 2.14 on page 30), there is no great difference between the implementations. To implement this use case, one must select the submenu element (`ul`), and position this relative to the parent element, when the parent element is hovered. All the solutions that build on top of CSS can perform this. The implementation in SASS however has the advantage that it uses nested rules. This was also an idea for PDSS that was never implemented. This means that many of the selectors in the PDSS implementation start with `ul#menu /(1) li`. This could have been avoided.

The table alignment use case was implemented in CCSS (see code example 2.15 on page 31). This was the only use case where one of the existing solutions clearly beats PDSS. In addition to being significantly more concise, the CCSS solution is also conceptually significantly easier to implement. In CCSS, we only needed to declare one variable `first_col`, select all the cells that should have the same width, and then specify the constraint `width = first_col`. In PDSS, we needed to select all the cells that should have the same width, and then get the widths of all the cells by using a value selector, and then assigning the maximum of these widths to the cell. The concept of constraints in CCSS is very powerful, and makes it easy to create a simple and short implementation of the use case. The implementation of the use case however shows one of the general advantages of PDSS, namely the fact that styling can be abstracted away into a function. In the CCSS implementation, three rules are specified, one for each column in the table. In the PDSS implementation, only one rule is specified inside a function. Executing this function instantiates the rule every time it is called, and by the use of parameters on the function, the rule is instantiated with different values. This makes it possible to use the same abstraction to style any number of columns. CCSS requires that a new rule and variable is declared for each column.

The use case centering was implemented in PSL (see code example 2.16). The way this use case was implemented in PSL and PDSS is quite similar. The element that should be centered is selected, and then the two properties left and top are calculated and set, based on the size of the parent element. The selector are however more interesting. What is more interesting is the selectors. In PDSS, the selector is `div#centeredDiv` whereas the following code is needed to select the element in PSL: `DIV {if (getAttribute(self, "id") == "innerDiv") then ...}`. In PSL, it is not possible to perform other selections but on tag names, and then filtering the selected elements by conditional statements. It is quite clear that the selectors in PDSS are simpler and more powerful than the ones in PSL.

7.2.3 Expressions

The last use case that has been implemented in PDSS is aspect maintenance which has also been implemented in CCSS (see code example 2.17). Both the implementations in CCSS and PDSS are simple and based on the same concept. The CCSS implementation however requires some more statements than in the PDSS implementation. This is due to

the constraint variables that must the declared and constrained to the width of the body and div element. This is not necessary in PDSS, where the width of the div element is read using a value selector directly in the rule.

Summary

All the use cases have been implemented in PDSS, and only in one case is the use case implementation in one of the existing styling solutions more concise. This is the case in the implementation of the table alignment use case in CCSS. In three of the use cases PDSS is considered more concise or conceptually simpler than the existing styling solution. This is the case with the ordinal selection and Selection intersection use case which was implemented in XSLT and the aspect maintenance use case implemented in PSL. In the remaining use cases there were no definite overall advantage.

7.3 Implementation Problems

A few bugs were found during the implementation of the use cases in PDSS. One of them is related to setting a styling property of an element, where the value to set is derived partly from the property itself. For instance when the height of an element is based on the height of multiple elements, of which one is the element itself. This causes a circular dependency with the element, and this circular dependency is caught by our engine that emits an error. The logic solution to this problem is to make the engine allow circular dependencies that involve only a single element. This issue has been avoided in our use case implementations by using an additional/intermediate variable.

Another rather serious issue is that the scroll position and text markings may disappear when the styles are reapplied. For instance, when hovering the mouse over an element, or when resizing the browser window. This issue is caused by all the styles on all DOM elements being reset to their defaults and then reassigned the style dictated by the PDSS code. The resetting of styles to their defaults is necessary in order to determine the initial dimensions of elements. Whether or not this is possible to fix is uncertain.

Other bugs have surfaced, but some seem to be related to the browser rather than our implementation. The issue is that, sometimes, the setting of style properties on DOM elements seems to be ignored until another style property is set. Our engine does not treat these properties differently, and therefore it seems to be a problem with the browser.

Apart from the mentioned bugs, the solution seems to be quite stable.

Chapter 8

Evaluation

This evaluation chapter is split into four parts. First, the analysis of the report is evaluated, which includes some shortcomings. Then the main contribution of the project, PDSS, is evaluated, and this is the most comprehensive part of the evaluation. Finally, some ideas for possible extensions and further work are presented.

8.1 Analysis

In the analysis a number of styling solutions was researched, and we became acquainted with different styling concepts, which have helped us to design our own solution.

The least helpful of these solutions have been XSLT. The reason for this is that it has no features concerned with styling the appearance of graphical elements. Instead, it relies completely on CSS for handling the appearance, which basically results in a static styling.

It was argued that Silverlight and its language XAML would not be looked into. But there might have been some interesting concepts we could have learned from XAML rather than looking into XSLT.

One of the most interesting of the existing solutions was DSSSL. Unfortunately the solution is so old and unused that it has been extremely hard to find relevant information about it, which has left some open questions with regards to its capabilities.

With respect to the use cases, they could have covered a wider range of functionality. More use cases about interaction and dynamic styling would have been relevant. The only kind of dynamic presented in the use cases is the resizing of the window. There could have been use cases covering functionality with respect to being able to add and remove styles dynamically, under different kinds of events.

Because of the prominence of CSS, some deeper research was done. This included both the current CSS 2.1 standard, but also CSS 3 which is yet to become a standard. This gave some insight in what the future brings to browsers. In addition to that, some research into the internals of a browser and the connection to JS has yielded some understanding of how it is possible to perform styling dynamically.

8.2 PDSS

On basis of the analysis, a problem statement was created in section 3, which defined the objectives of the rest of the report. The main objective was to create a new powerful

dynamic styling solution. The styling solution has been created, and in this section it is evaluated. First, it is evaluated whether or not PDSS fulfills the requirements stated in the problem statement. Then the language of PDSS is evaluated in more details. This includes the syntax of the language, and how the different languages used in PDSS (JS and CSS) work together. The evaluation of the functionality of PDSS took place in section 7.2, and will only be mentioned briefly in this section.

Requirements

In the following list, it will be described if PDDS fulfills each of the requirements from the problem statement and to which extent.

Powerful element selectors: The demands for the selectors were that it should be possible to select elements based on the content and context of the element. In PDSS, it is possible to select elements by using the descendant, ancestor, left sibling and right sibling operators. Furthermore, it is possible to define a number of links for each of these operators. So it is indeed possible to do contextual selections in PDSS. Selecting elements based on content is not directly possible in the selectors, it is only possible to use class names and ids as selection parameters. It is however possible to do content based selection by using the predicates, which were another demand for the selectors. In the predicates, arbitrary constraints can be specified in JS, among others the content of an element.

Interelement constraints: One of the demands was that it should be possible to base the style of one element on the style of other elements. This is possible in PDSS by means of the value selectors, which can be used in the JS expressions that are used to set the style properties of an element. Furthermore, PDSS ensures that a style will be updated if the style is dependent on the style of another element, and this style is changed. The properties that can be accessed on other elements through the style property abstraction are the CSS properties but also the rendered properties. So interelement constraints has been implemented in PDSS.

Powerful expressions: As JS expressions are used as expressions in PDSS, it is possible to use arithmetic, variables and function calls in all the expressions. And as JS is a very powerful language, this also gives very powerful expressions in PDSS.

Implement all use cases: It was a requirement that PDSS should be able to implement all the use cases presented in section 2.3. This has succeeded, and the implementation of all the use cases in PDSS is shown in section 7.1.

More powerful than CSS: Another requirement was that PDSS should be more powerful than CSS. And as all use cases have been implemented in PDSS, whereas only three can be implemented in CSS (see table 2.1), it is reasonable to conclude that PDSS is more powerful than CSS.

Run natively in the browser: The last demand for PDSS was that it should run natively in the browser. PDSS is implemented using JavaScript as the UPL and CSS to set the styling properties. Both of these are integrated into all modern web browsers, so PDSS code will execute natively in all modern browsers, when it has been compiled by the PDSS compiler.

To sum up, all the demands for PDSS that were stated in the problem statement have been fulfilled.

The Language

For the parser to distinguish PDSS constructs from JS constructs, the characters `$` and `@` followed by curly brackets is used to form some kind of *PDSS construct container*. The syntax for the PDSS construct container is easy to use, as it only requires that the developer writes three symbols to create a PDSS construct. The meaning and use of `$` and `@` is not intuitive, and requires familiarity with the syntax and semantics of PDSS. This is because the characters do not have any common meaning among programming languages in general. One solution for this could be to use some keywords instead of special characters, this will increase the readability but reduce the writeability.

With regards to the contents of the PDSS construct container, the code is quite intuitive. The style selector operators are similar to how a path in a file system structure is navigated, and otherwise works a lot like the selectors in CSS and other solutions. The style blocks are also quite intuitive, as they are similar to an ordinary function scope in JS, but with the possibility of setting style properties. The biggest difference is that the evaluation of a style block is beyond the control of the developer.

Besides all the `$` and `@` symbols and braces in PDSS, the syntax is rather simple and intuitive.

PDSS code is a mixture of CSS properties, JS and PDSS constructs. In the following sections, each of them will be described in more details. It is described which role each of them plays, how they work together with each other, and what this means for the developer that uses PDSS.

CSS Properties

When doing styling in PDSS, this is done by means of setting style properties to some values. Behind the scene, this is implemented by using CSS properties, and the names of the style properties that correspond to the CSS properties. This means that it is required that the developer knows the CSS properties in order to be able to use PDSS. Some abstractions could have been put on top of the CSS properties, such that PDSS had its own styling properties. These could however not be more powerful than the properties in CSS, as PDSS builds on top of CSS. And CSS is the only styling solution that can be used to style elements in a web browser, at least if it should run natively without additional plug-ins, so PDSS has to be based on CSS. With this in mind, the use of CSS properties is unavoidable and thereby the best choice.

All style property names are denoted by a `@` symbol, such that the parser can distinguish these from regular JS variables. This however also increases the readability of the code, as it is easy to identify the style properties that are used in the code.

Generally the integration with CSS is unavoidable, and as most web developer most likely already knows the CSS properties, this is not a problem. So the style property part of PDSS is as it should be. Many of the existing styling solutions including XSLT, SASS and CCSS also use the CSS properties.

JavaScript

JS is the language used to add dynamics and expressive power to PDSS. It is possible to use JS within many of the PDSS language constructs. This includes style blocks, predicates in style selectors and expression that are used to assign values to CSS properties. It is also possible to use PDSS constructs like style rule and style statement inside JS constructs such as conditional statements and functions. This makes PDSS very dynamic.

All these possibilities give the developer a lot of freedom to code the styling aspect as he sees fit. Some styling challenges might be possible to solve in multiple ways in PDSS, so the developer must choose how to do it. This can be both an advantage but also a disadvantage. It is an advantage because it pleases the developer that he can do it the way he wants to, but it is a disadvantage because other developers might have trouble understanding the PDSS code. We think that the developer should have as must expressive power as possible, and PDSS certainly provide this, so JS has been the right choice.

Most web developers, at least those that have done client side programming before, know how JS works. So it should be relative easy for them to start using PDSS.

PDSS Constructs

The PDSS constructs are used to glue JS and CSS style properties together. For instance, the style statement is used to glue together a style property and a JS expression. And a style block is used to glue multiple style statements and JS together. But the PDSS constructs also provides additional features that neither JS nor CSS provides. This is for instance the powerful selectors and the functionality that style rules are evaluated in an order such that dependencies are respected.

There are six different language constructs in PDSS (style selector, style property, value selector, style statement, style block and style rule). Most of these, with the exception of style selector and style rule, are very simple constructs. The style selector is a more comprehensive construct in that many different operators can be used to select elements. The style rule is a syntactically simple construct, but the semantics of it is harder to understand. This is because of the dependency constraints, which means that the style block of a style rule is not necessarily executed sequentially for all the elements that match the style selector. All in all, if a new developer should learn the language, and he can understand the style selector and style rule constructs, then he would probably not have any problems using the other constructs. Therefore, we claim that PDSS is a language that is easy to learn.

As all the constructs in PDSS are compiled to JS and uses a JS library to implement some of the functionalities, one may argue that a developer do not need PDSS and could just have used JS. And this is true, but the abstractions in PDSS over CSS properties and JS make it much easier and concise to do styling.

Prototype State

The implementation of PDSS is not in a state that is acceptable for production sites. As mentioned in section 7.3, the implementation has some significant bugs, which needs to be fixed first. The bugs were found during a test (implementation of the use cases in section 7.1), but a more thorough test of the functionality of PDSS will be necessary in order to ensure the stability of PDSS.

But as stated in the problem statement, only a prototype of the styling language should be implemented, and this was done. In the prototype, all the important features of PDSS

have been realized. All the use cases have been implemented in PDSS, and as the use cases were designed to cover many aspects of styling, we claim that PDSS can be used to perform all the styling that is necessary in web development today.

PDSS is only one step in the direction of a new powerful and dynamic styling solution, the next step will be to adjust and finalize the solution. Besides the mentioned bugs, the most important thing to be changed is better integration into the UPL, in the sense that the PDSS constructs should fit more natural into JS. The major reason this was not done in the prototype, is because it requires the compiler to be fully aware of the JS syntax. In this way, many of the `$` and `@` symbols and braces could be removed, yielding a cleaner solution syntactically. Some other ideas for possible extensions are presented later.

Most web developers know how to use the existing and very popular languages CSS and JS. And as these languages are used in PDSS (only the properties from CSS), the learning curve of PDSS will be quite flat, maybe with a little bump in the beginning. The bump is because the developers must learn to use the style selectors, and understand the semantics of style rules, as described earlier.

The use of JS and CSS, which are implemented natively in all modern browsers, should also ensure that PDSS can be used in most browsers in the future.

8.3 Possible Extensions

The PDSS solution is only a prototype. Therefore there are a number of features that it lacks. The following list contains some of the more notable non-implemented features. Some of these were already presented in chapter 4. The three first features require modification of the implementation, where as the fourth does not.

Element reference: Being able to acquire a reference inside a style rule, to the element the rule is styling, would make it possible to get and set data unrelated to styling. An example could be to set various events, or to insert sub elements.

Nested rules: Nested rules would make it possible to create sub rules that styles elements relative to the outer rule. This could depend on the element reference. An example of the use could be when wanting to style a group of elements in close proximity in the DOM. A rule could then be used to select the location of such a group, and then nested rules could be used to style different members of the group of elements. Such a group could be a `div` with a `p` as child. A rule could then select the `p`, and then style is as intended, but inside the style block, a nested style block could reside. This nested block would then select the child `p` of the `div` element. This nested rule would then style the `p` element, and could possibly do so with access to the properties of the outer rule (parent `div`).

Selector concatenation: Currently, it is possible to combine style blocks. But the same concept can be implemented for selectors, and will make PDSS more expressive.

Predicate helpers: While the library already contains functions such as `content(e)` and `hover(e)` there is plenty of room for enhancements. An example could be a `last(e)` which would return true for all elements that are the last child.

8.4 Further Work

In addition to the extensions mentioned in previous section, there are a number of areas that can be researched further. Some of these topics have been touched in this report, especially the performance related topics.

One of the things that can be researched is, to what extent it is possible to streamline our extension into the UPL. For instance removal of the dollar- and at-signs, as well as the brackets around selectors.

Research into dedicated querying languages could also yield ideas for further extension of the selectors.

With the goal of performance we have identified a number of areas worth looking into.

- The dependency graph could be partially based on static analysis of the style rule declaration, rather than at runtime.
- Eliminating the need to perform redundant styling of elements.
- Analysis of the style rules applied at a given point in time, with the purpose of translating them into equivalent CSS which could be loaded into the browser, and thereby offloading the JS engine.

Chapter 9

Conclusion

When dealing with web development there are a number of interesting concepts, and one of these is styling. The problem of styling becomes more demanding as the world moves more and more towards RIAs. This report builds on a previous report[1], where nine solutions to unified web development were examined. In this project, these solutions were examined further, and all of them were found to have limited styling capabilities with regards to styling mechanisms. A number of additional solutions, specifically intended for styling, were found. Of these styling solutions, the most promising eight solutions were further investigated. These were: CSS 2, CSS 3, CCSS, DSSSL, XSLT, PSL, SASS, and JSSS. All of these were investigated with regards to origin, general features and market penetration.

In order to assess the practical capabilities of these eight solutions more thoroughly, nine use cases were created. The use cases were concerned with three groups of features, namely Element Selection, Interelement Constraint, and Expressions. Based on the knowledge from the analysis of the styling solution, a table of capabilities was created. To verify the capabilities, and to get practical experience of using the solutions, some of the use cases were implemented in some of the solutions. Most of the more expressive and generally promising solutions were the ones with the least market penetration. This was shown by the table of capabilities, in combination with the general research of the solutions and the use case implementations.

Because of the lack of an expressive and widespread styling solution, the problem of developing a new expressive styling solution was defined. The solution had to be at least as powerful as the currently most widespread styling solution, CSS. Three requirements to the expressiveness of the solution were defined, which are recapitulated here:

Element selectors: It should be possible with some predefined operators to select elements based on their content and the context in which they exist. To give even more expressive power to the selectors, it should also be possible for a developer to define his own arbitrary constrains in form of predicates.

Interelement constraints: When styling an element, the solution should have the possibility to access properties of other elements. These properties should be the ones that are set by other styling rules, but also properties that represent the actual position and size of elements in the browser.

Expressions: To increase the expressiveness of expressions, it should, besides accessing properties of other elements, be possible to use arithmetic, variables and functions.

Further, the solution should be able to implement all of the previously created use cases, integrated on top of a general purpose programming language, and being able to run natively in web browsers.

A new styling solution (PDSS) was designed, and a prototype of this was implemented. PDSS turned out to be an extension to JavaScript, which means that it integrates some styling constructs into JavaScript. To assess whether or not the prototype matches the requirements, it was tested by implementing the nine use cases.

The result was that the prototype indeed lives up to the requirements. It has much more powerful selectors than CSS, and many of the other solutions. It has the possibility of specifying that some properties should be constrained to the properties of other elements. At the same time, it is possible to use all of the features of JavaScript to set the property values. This includes arithmetic expressions, variables and functions. All of the use cases were also successfully implemented in PDSS, and it turned out that PDSS has a number of advantages compared to the existing solutions.

The solution does however have a few bugs, one of which is quite serious. The bug prevents the solution from being widely useful, as it will be a great annoyance for people not to be able to keep a scroll position or to select text. If this issue is solved, the solution could be used in production with advantage. There is still some work ahead in order to get an optimal solution. Ideas for future work have been presented, which aims at enhancing the solution. In conclusion a reasonable solution to the problem of styling has successfully been developed.

Bibliography

[1] Daniel S. Korsgård, Morten Bøgh, Michael S. Knudsen, and Markus Krogh. Unified multitier web development, 2008. Dat5-project by computer science group d525a at Aalborg University.

[2] NextApp, Inc. Echo web framework, 2008. http://echo.nextapp.com/site/, Visited 05/12/08.

[3] NETiKA Technologies. Official web site for goa winforms, 2008. http://www.netikatech.com/, Visited 05/12/08.

[4] Google. Official gwt web site, 2008. http://code.google.com/webtoolkit/, Visited 05/12/08.

[5] Official web site for helma, 2008. http://dev.helma.org/, Visited 05/12/08.

[6] The hop project site, 2008. http://hop.inria.fr/, Visited 05/12/08.

[7] Edinburgh University. Links: Linking theory to practice for the web, 2008. http://groups.inf.ed.ac.uk/links/, Visited 05/12/08.

[8] Nikhil Kothari. The official script# site, 2008. http://projects.nikhilk.net/ScriptSharp/, Visited 05/12/08.

[9] Microsoft. The official microsoft silverlight site., 2008. http://silverlight.net/, Visited 05/12/08.

[10] Stephen Chong, Andrew Myers, K. Vikram. The swift project site, 2008. http://www.cs.cornell.edu/jif/swift/, Visited 05/12/08.

[11] Ezra Cooper and Sam Lindley and Philip Wadler and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, pages 266–296, 2006.

[12] Official web site for haxe, 2009. http://haxe.org/, Visited 26/05/08.

[13] M. Hostetter and D. Kranz and C. Seed and C. Terman and S. Ward. Curl: a gentle slope language for the Web. *World Wide Web J.*, 2(2):121–134, 1997.

[14] W3C (Bert Bos, Tantek Çelik, Ian Hickson and Håkon Wium Lie). Cascading style sheets level 2 revision 1 (css 2.1) specification, w3c candidate recommendation 23 april 2009. http://www.w3.org/TR/CSS2/, Visited 26/05/09.

[15] W3C (Bert Bos). Cascading style sheets - current work. http://www.w3.org/Style/CSS/current-work, Visited 26/05/09.

[16] Greg J. Badros and Alan Borning and Kim Marriott and Peter J. Stuckey. Constraint cascading style sheets for the web. In *ACM Symposium on User Interface Software and Technology*, pages 73–82, 1999.

[17] Joint Technical Commitee ISO/IEC JTC 1, Information technology. Information technology - processing languages - document style semantics and specification language (dsssl). `http://www.ibiblio.org/pub/sun-info/standards/dsssl/draft/`, Visited 06/04/09.

[18] W3C (Anders Berglund). Extensible stylesheet language (xsl) version 1.1, w3c recommendation 05 december 2006. `http://www.w3.org/TR/xsl11/`, Visited 26/05/09.

[19] Ethan V. Munson. A new presentation language for structured documents. electronic publishing: Origination, dissemination, and design. In *the Sixth International Conference on Electronic Publishing, Document Manipulation, and Typography*, pages 8–125, 1995.

[20] Louis Weitzman and Kent Wittenburg. Relational grammars for interactive design. In *VL*, pages 4–11. IEEE Computer Society, 1993.

[21] The Haml Team (Hampton Catlin the driving force behind it and the brains of the operation). Module: Sass. `http://haml.hamptoncatlin.com/docs/rdoc/classes/Sass.html`, Visited 26/05/09.

[22] Lou Montulli and Brendan Eich and Scott Furman and Donna Converse and Troy Chevalier (all Netscape). Javascript-based style sheets, initial proposal, 1996. `http://www.w3.org/Submission/1996/1/WD-jsss-960822`, Visited 09/03/09.

[23] Håkon Wium Lie and Bert Bos. Cascading style sheets, level 1, w3c recommendation 17 dec 1996, revised 11 apr 2008. `http://www.w3.org/TR/CSS21/`, Visited 11/03/09.

[24] Bert Bos and Tantek Çelik and Ian Hickson and Håkon Wium Lie. Cascading style sheets level 2 revision 1 (css 2.1) specification, w3c candidate recommendation 19 july 2007. `http://www.w3.org/TR/CSS21/`, Visited 11/03/09.

[25] Eric A. Meyer and Bert Bos (World Wide Web Consortium). Introduction to css3, w3c working draft, 23 may 2001. `http://www.w3.org/TR/css3-roadmap/`, Visited 09/03/09.

[26] Ian Jacobs, Head of W3C Communications. About the world wide web consortium (w3c). `http://www.w3.org/Consortium/`, Visited 26/05/09.

[27] Philip M. Marden Jr. and Ethan V. Munson. Psl: An alternate approach to style sheets for the web. In Hermann A. Maurer and Richard G. Olson, editor, *WebNet*. AACE, 1998.

[28] Vlad Alexander (xhtml.com). Conversation with css 3 team. `http://xhtml.com/en/css/conversation-with-css-3-team/`, Visited 16/03/09.

[29] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[30] W3C and the WAM (Web, Adaptation and Multimedia) project at INRIA. Amaya home page - w3c's editor/browser. http://www.w3.org/Amaya/, Visited 26/05/09.

[31] Toshimitsu Suzuki and Masatomo Goto. Xml projects in japan and fujitsu's approach to xlink/xpointer. http://www.fujitsu.com/downloads/MAG/vol36-2/paper09.pdf, Visited 06/04/09.

[32] Open Source Community. Openjade distribution page. http://openjade.sourceforge.net/, Visited 06/04/09.

[33] Organization for the Advancement of Structured Information Standards (OASIS). Official website for docbook. http://www.docbook.org/, Visited 26/05/09.

[34] Paul Prescod. Introduction to dsssl, 1997. http://www.prescod.net/dsssl/, Visited 01/06/09.

[35] W3C (James Clark). Xsl transformations (xslt) version 1.0, w3c recommendation 16 november 1999. http://www.w3.org/TR/xslt, Visited 22/03/09.

[36] W3C (Michael Kay - Saxonica). Xsl transformations (xslt) version 2.0, w3c recommendation 23 january 2007. http://www.w3.org/TR/xslt20/, Visited 22/03/09.

[37] The Haml Team (Hampton Catlin the driving force behind it and the brains of the operation). #haml. http://haml.hamptoncatlin.com/, Visited 18/03/09.

[38] Haml — google groups. http://groups.google.com/group/haml?hl=en, Visited 18/03/09.

[39] Peter Belesis. Dhtml lab: Hierarchical menus, iii; frames - javascript style sheets. http://www.webreference.com/dhtml/column18/menuFrJSS.html, Visited 16/03/09.

[40] Jan Roland Eriksson. About ns4x and jsss, how one specific implementation of css came to be. http://www.dev-archive.net/articles/About-JSSS.html, Visited 16/03/09.

[41] Net Applications. Market share for browsers, operating systems and search engines. http://marketshare.hitslink.com/, Visited 11/03/09.

[42] Dave Raggett, Arnaud Le Hors and Ian Jacobs (World Wide Web Consortium). Html 4.01 specification, w3c recommendation 24 december 1999. http://www.w3.org/TR/html401/present/graphics.html, Visited 06/04/09.

[43] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, John Williams. Selectors level 3, w3c working draft 10 march 2009. http://www.w3.org/Style/CSS/current-work, Visited 01/06/09.

[44] W3C (Chris Lilley) and Opera Software(Håkon Wium Lie). Css3 values and units, w3c working draft 19 september 2006. http://www.w3.org/Style/CSS/current-work, Visited 01/06/09.

[45] Mozilla Foundation. Gecko - mozilla developer center. https://developer.mozilla.org/en/Gecko, Visited 11/03/09.

[46] L. David Baron (Mozilla Corporation). Mozilla's layout engine, 2006. http://www.mozilla.org/newlayout/doc/layout-2006-07-12/master.xhtml, Visited 03/06/09.

[47] L. David Baron (Mozilla Corporation). Mozilla style system documentation, 2008. http://www.mozilla.org/newlayout/doc/style-system.html, Visited 03/06/09.

[48] Mozilla. Stylesheet object - mozilla developer center, 2007. https://developer.mozilla.org/en/DOM/stylesheet, Visited 03/06/09.

[49] JavaScript Kit (A web site with JavaScript tutorials). Changing external style sheets using the dom. http://www.javascriptkit.com/dhtmltutors/externalcss.shtml, Visited 03/06/09.

[50] Terence Parr (Professor at the University of San Franciso) and many others (see http://www.antlr.org/credits.html). Antlr parser generator. http://www.antlr.org/, Visited 15/05/09.

Appendix A

Appendix

A.1 Code Examples

This appendix contains more detailed versions of the code examples that have been presented throughout the report. The code examples in the report only presents the most important parts, whereas the whole picture of the code examples can be seen in this section.

A.1.1 XSLT Code Example

Source tree

```
<?xml version="1.0"?>
<?xml-stylesheet href="XSLT.code" type="text/xsl" ?>
<people>
  <person>
  <tal>-1</tal>
    <fullname>Delora Jed</fullname>
    <mail>delora@jed.com</mail>
    <affiliation>Student</affiliation>
  </person>
  <person>
    <fullname>Laurine Giselle</fullname>
    <mail>laurine@giselle.com</mail>
    <affiliation>Student</affiliation>
  </person>
  <person>
    <fullname>Davina Aleta</fullname>
    <mail>davina@aleta.com</mail>
    <affiliation>Employee</affiliation>
  </person>
  <person>
    <fullname>Devyn Tracey</fullname>
    <mail>devyn@tracey.com</mail>
    <affiliation>Employee</affiliation>
  </person>
  <person>
    <fullname>Trev Dana</fullname>
    <mail>trev@dana.com</mail>
    <affiliation>Student</affiliation>
  </person>
  <person>
    <fullname>Madlyn Karyn</fullname>
    <mail>madlyn@karyn.com</mail>
    <affiliation>Student</affiliation>
  </person>
</people>
```

Code example A.1: *Source tree*

Style Sheet

```
1 <?xml version="1.0" ?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns="
      http://www.w3.org/1999/xhtml">
3   <xsl:output method="xml" indent="yes" encoding="UTF-8"/>
4
5   <xsl:variable name="headerRepeatInterval" select="5"/>
6   <xsl:variable name="header">
7     <tr style="background-color: silver; font-size: 1.4em;">
8       <td>Full name</td>
9       <td>Affiliation</td>
10      <td>Mail</td>
11    </tr>
12  </xsl:variable>
13
14  <xsl:template match="/people">
15    <html>
16      <head></head>
17      <body style="font-size: 11px; font-family: verdana;">
18        <table style="border: 1px solid black; width: 400px;">
19          <xsl:copy-of select="$header" />
20          <xsl:for-each select="person">
21            <xsl:sort select="affiliation"/>
22            <xsl:sort select="fullname"/>
23            <xsl:sort data-type="number" select="string-length(mail)"/>
24
25            <xsl:if test="position() mod $headerRepeatInterval = 0">
26              <xsl:copy-of select="$header" />
27            </xsl:if>
28
29            <tr>
30              <td><xsl:value-of select="fullname" /></td>
31              <td><xsl:value-of select="affiliation" /></td>
32              <td><xsl:value-of select="mail" /></td>
33            </tr>
34          </xsl:for-each>
35        </table>
36      </body>
37    </html>
38  </xsl:template>
39 </xsl:stylesheet>
```

Code example A.2: *Style sheet*

Result Tree

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
      DTD/xhtml1-strict.dtd">
2 <html xmlns="http://www.w3.org/1999/xhtml">
3   <head>
4     <title></title>
5   </head>
6   <body style="font-size: 11px; font-family: verdana;">
7     <table style="border: 1px solid black; width: 400px;">
8       <tr style="background-color: silver; font-size: 1.4em;">
9         <td>Full name</td>
10        <td>Affiliation</td>
11        <td>Mail</td>
12      </tr>
13      <tr>
14        <td>Davina Aleta</td>
15        <td>Employee</td>
16        <td>davina@aleta.com</td>
17      </tr>
18      <tr>
19        <td>Devyn Tracey</td>
20        <td>Employee</td>
21        <td>devyn@tracey.com</td>
22      </tr>
23      <tr>
24        <td>Delora Jed</td>
25        <td>Student</td>
26        <td>delora@jed.com</td>
27      </tr>
```

```
28        <tr>
29          <td>Laurine Giselle</td>
30          <td>Student</td>
31          <td>laurine@giselle.com</td>
32        </tr>
33        <tr style="background-color: silver; font-size: 1.4em;">
34          <td>Full name</td>
35          <td>Affiliation</td>
36          <td>Mail</td>
37        </tr>
38        <tr>
39          <td>Madlyn Karyn</td>
40          <td>Student</td>
41          <td>madlyn@karyn.com</td>
42        </tr>
43        <tr>
44          <td>Trev Dana</td>
45          <td>Student</td>
46          <td>trev@dana.com</td>
47        </tr>
48      </table>
49    </body>
50  </html>
```

Code example A.3: *Result tree (when using the built-in XSLT processors in Mozilla Firefox version 3.0.8)*

Screenshot

Full name	Affiliation	Mail
Davina Aleta	Employee	davina@aleta.com
Devyn Tracey	Employee	devyn@tracey.com
Delora Jed	Student	delora@jed.com
Laurine Giselle	Student	laurine@giselle.com
Full name	Affiliation	Mail
Madlyn Karyn	Student	madlyn@karyn.com
Trev Dana	Student	trev@dana.com

Figure A.1: *Example of what the result tree in code example A.3 looks like when rendered in Mozilla Firefox version 3.0.8*

A.1.2 Cell Content Use Case in XSLT

Source tree

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet href="elementSelection.xslt" type="text/xsl" ?>
3  <people>
4    <person>
5      <fullname>Delora Jed</fullname>
6      <income>89</income>
7      <expense>3</expense>
8    </person>
9    <person>
10     <fullname>Laurine Giselle</fullname>
11     <income>24</income>
12     <expense>74</expense>
13   </person>
14   <person>
15     <fullname>Davina Aleta</fullname>
16     <income>22</income>
17     <expense>48</expense>
18   </person>
19   <person>
```

```
20      <fullname>Devyn Tracey </fullname>
21      <income>21</income>
22      <expense>77</expense>
23    </person>
24    <person>
25      <fullname>Trev Dana </fullname>
26      <income>93</income>
27      <expense>89</expense>
28    </person>
29    <person>
30      <fullname>Madlyn Karyn </fullname>
31      <income>37</income>
32      <expense>16</expense>
33    </person>
34  </people>
```

Code example A.4: *Source tree*

Style Sheet

```
1  <?xml version="1.0" ?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns="
       http://www.w3.org/1999/xhtml">
3    <xsl:output method="xml" indent="yes" encoding="UTF-8"/>
4
5    <xsl:template match="/people">
6      <html>
7        <head>
8        </head>
9        <body style="font-size: 11px; font-family: verdana;">
10         <table style="border: 1px solid black;">
11           <tr style="background-color: silver; font-size: 1.4em;">
12             <td>Full name</td>
13             <td>Income</td>
14             <td>Expense</td>
15             <td>Total</td>
16           </tr>
17
18           <xsl:for-each select="person">
19             <xsl:sort select="fullname"/>
20             <tr>
21               <td><xsl:value-of select="fullname" /></td>
22               <td><xsl:value-of select="income" /></td>
23               <td><xsl:value-of select="expense" /></td>
24
25               <xsl:choose>
26                 <xsl:when test="income - expense &lt; 0">
27                   <td style="color: red;">
28                     <xsl:value-of select="income - expense" />
29                   </td>
30                 </xsl:when>
31                 <xsl:otherwise>
32                   <td>
33                     <xsl:value-of select="income - expense" />
34                   </td>
35                 </xsl:otherwise>
36               </xsl:choose>
37             </tr>
38           </xsl:for-each>
39         </table>
40       </body>
41     </html>
42
43   </xsl:template>
44
45 </xsl:stylesheet>
```

Code example A.5: *Style sheet*

Result Tree

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
    DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title></title>
  </head>
  <body style="font-size: 11px; font-family: verdana;">
    <table style="border: 1px solid black;">
      <tr style="background-color: silver; font-size: 1.4em;">
        <td>Full name</td>
        <td>Income</td>
        <td>Expense</td>
        <td>Total</td>
      </tr>
      <tr>
        <td>Davina Aleta</td>
        <td>22</td>
        <td>48</td>
        <td style="color: red;">-26</td>
      </tr>
      <tr>
        <td>Delora Jed</td>
        <td>89</td>
        <td>3</td>
        <td>86</td>
      </tr>
      <tr>
        <td>Devyn Tracey</td>
        <td>21</td>
        <td>77</td>
        <td style="color: red;">-56</td>
      </tr>
      <tr>
        <td>Laurine Giselle</td>
        <td>24</td>
        <td>74</td>
        <td style="color: red;">-50</td>
      </tr>
      <tr>
        <td>Madlyn Karyn</td>
        <td>37</td>
        <td>16</td>
        <td>21</td>
      </tr>
      <tr>
        <td>Trev Dana</td>
        <td>93</td>
        <td>89</td>
        <td>4</td>
      </tr>
    </table>
  </body>
</html>
```

Code example A.6: *Result tree (when using the built-in XSLT processors in Mozilla Firefox version 3.0.8)*

Screenshot

Full name	Income	Expense	Total
Davina Aleta	22	48	-26
Delora Jed	89	3	86
Devyn Tracey	21	77	-56
Laurine Giselle	24	74	-50
Madlyn Karyn	37	16	21
Trev Dana	93	89	4

Figure A.2: *Example of what the result tree in code example A.6 looks like when rendered in Mozilla Firefox version 3.0.8*

A.1.3 Selection Intersection Use Case in XSLT

Source tree

```
1 <?xml version="1.0"?>
2 <?xml-stylesheet href="selectionIntersection.xslt" type="text/xsl" ?>
3 <table>
4   <tr>
5     <td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td>
6   </tr>
7   <tr>
8     <td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td>
9   </tr>
10  <tr>
11    <td>o</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td>
12  </tr>
13  <tr>
14    <td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>a</td><td>b</td>
15  </tr>
16 </table>
```

Code example A.7: *Source tree*

Style Sheet

```
 1 <?xml version="1.0" ?>
 2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns="
       http://www.w3.org/1999/xhtml">
 3   <xsl:output method="xml" indent="yes" encoding="UTF-8"/>
 4
 5   <xsl:template match="/table">
 6     <html>
 7       <head>
 8       </head>
 9       <body style="font-size: 11px; font-family: verdana;">
10         <table style="border: 1px solid black;">
11           <xsl:for-each select="tr">
12             <tr>
13               <xsl:choose>
14                 <xsl:when test="position() mod 2  = 0">
15                   <xsl:for-each select="td">
16                     <xsl:choose>
17                       <xsl:when test="position() mod 2 = 0">
18                         <td style="background-color: silver;">
19                           <xsl:value-of select="."/>
20                         </td>
21                       </xsl:when>
22                       <xsl:otherwise>
23                         <td>
24                           <xsl:value-of select="."/>
25                         </td>
26                       </xsl:otherwise>
27                     </xsl:choose>
28                   </xsl:for-each>
29                 </xsl:when>
30                 <xsl:otherwise>
31                   <xsl:for-each select="td">
32                     <td>
33                       <xsl:value-of select="."/>
34                     </td>
35                   </xsl:for-each>
36                 </xsl:otherwise>
37               </xsl:choose>
38             </tr>
39           </xsl:for-each>
40         </table>
41       </body>
42     </html>
43   </xsl:template>
44 </xsl:stylesheet>
```

Code example A.8: *Style sheet*

Result Tree

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title></title>
  </head>
  <body>
    <table style="border: 1px solid black;">
      <tr>
        <td>a</td>
        <td>b</td>
        <td>c</td>
        <td>d</td>
        <td>e</td>
        <td>f</td>
        <td>g</td>
      </tr>
      <tr>
        <td>h</td>
        <td style="background-color: silver;">i</td>
        <td>j</td>
        <td style="background-color: silver;">k</td>
        <td>l</td>
        <td style="background-color: silver;">m</td>
        <td>n</td>
      </tr>
      <tr>
        <td>o</td>
        <td>p</td>
        <td>q</td>
        <td>r</td>
        <td>s</td>
        <td>t</td>
        <td>u</td>
      </tr>
      <tr>
        <td>v</td>
        <td style="background-color: silver;">w</td>
        <td>x</td>
        <td style="background-color: silver;">y</td>
        <td>z</td>
        <td style="background-color: silver;">a</td>
        <td>b</td>
      </tr>
    </table>
  </body>
</html>
```

Code example A.9: *Result tree (when using the built-in XSLT processors in Mozilla Firefox version 3.0.8)*

Screenshot

Figure A.3: *Example of what the result tree in code example A.9 looks like when rendered in Mozilla Firefox version 3.0.8*

A.2 PDSS grammar

This appendix contains the full ANTLR grammar for PDSS.

```
source
  : js_statements EOF;

js_statements
  : (js|rule|expression|statement|SEMI_COLON)*;

js_expression
  : (js|rule|expression)*;

js
  : MATCH_ALL_ELSE
  | JS_OPERATORS
  | NAME
  | L_CURLY js_statements R_CURLY
  | L_SQUARE js_statements R_SQUARE
  | L_ROUND js_statements R_ROUND
  | L_ANGLE
  | R_ANGLE
  | ADD
  | SUB
  | MUL
  | DIV
  | CARET
  | DOT
  | EQUAL
  | COMMA
  | HASH;


rule
  : STYLE selector '{' js_statements '}';

expression
  : '@' '{' js_statements '}'
  | '$' '{' selector '}' property?;

statement
  : property EQUAL js_expression SEMI_COLON;

property
  : AT NAME;

selector
  : selector_Term_First (
      ( CARET // intersection
      | ADD // union
      | SUB // complement
      )
      selector_Term_First
    )*;

selector_Term_First
  : (NAME | '*') selector_Predicate* selector_Term?;


selector_Term
  : selector_Operator (NAME | '*') selector_Predicate* selector_Term?;


selector_Predicate
  : L_SQUARE js_expression R_SQUARE
  | COMMA NAME
  | HASH NAME;

selector_Operator
  : DIV (L_ROUND js_expression R_ROUND)? // child
  | DOT (L_ROUND js_expression R_ROUND)? // parent
  | L_ANGLE (L_ROUND js_expression R_ROUND)? // left sibling
  | R_ANGLE (L_ROUND js_expression R_ROUND)? // right sibling;


/* Tokens */

STYLE
  : 'style';

```

```
 77 NAME
 78   : AplhaNum+;
 79
 80 fragment
 81 AplhaNum
 82   : 'a'..'z'
 83   | 'A'..'Z'
 84   | '0'..'9';
 85
 86 L_CURLY
 87   : '{';
 88
 89 R_CURLY
 90   : '}';
 91
 92 L_SQUARE
 93   : '[';
 94
 95 R_SQUARE
 96   : ']';
 97
 98 L_ROUND
 99   : '(';
100
101 R_ROUND
102   : ')';
103
104 L_ANGLE
105   : '<';
106
107 R_ANGLE
108   : '>';
109
110 ADD
111   : '+';
112
113 SUB
114   : '-';
115
116 MUL
117   : '*';
118
119 DIV
120   : '/';
121
122 DOT
123   : '.';
124
125 CARET
126   : '^';
127
128 EQUAL
129   : '=';
130
131 AT
132   : '@';
133
134 SEMI_COLON
135   : ';';
136
137 COMMA
138   : ',';
139
140 HASH
141   : '#';
142
143 WS
144   : (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;};
145
146
147 /* Additional tokens used in javascript */
148
149 JS_OPERATORS
150   : '=='
151   | '!='
152   | '==='
153   | '!=='
154   | '!'
155   | '<='
156   | '>='
157
```

```
158     | '+='
159     | '-='
160     | '*='
161     | '/='
162     | '%='
163     | '<<='
164     | '>>='
165     | '>>>='
166     | '|='
167     | '&='
168     | '%'
169     | '//'
170     | '++'
171     | '--'
172     | '<<'
173     | '>>'
174     | '&&'
175     | '||'
176     | '|'
177     | '&'
178     | '>>>'
179     | ':'
180     | '?'
181     | '\\'
182     | '~';
183
184 MATCH_ALL_ELSE
185     //: ~('"'|'\x00'..' ')?
186     : '"' (~'"'|'\\"')* '"'
187     | '\'' (~'\''|'\\\'')* '\''
188
189     | (('0'..'9')* ('.' ('0'..'9')+) | ('0'..'9')+ ('.' ('0'..'9')+)?) (('E'|'e')('+'|'-
            ')('0'..'9')+)? /* floats */
190     //| (s/foo/bar/g) /* regex */;
191
192 COMMENTS
193     : '/*' (~'*'|'*' ~'/')* '*/'  {$channel=HIDDEN;} /* C-style block comment */
194     | '//' ~('\r'|'\n')* ('\r'|'\n'|'\r\n')  {$channel=HIDDEN;};
```

Code example A.10: *PDSS grammar*