Attribute Grammars: An Executable Specification for CSS Layout

By

Scott Buckley

A thesis submitted to Macquarie University
for the degree of Master of Research
Department of Computing
October 2014

Typeset in LaTeX 2_ε.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Scott Buckley

Abstract

The World Wide Web is a technology used daily by billions of people internationally. The core technologies that make up the Web are specified by the World Wide Web Consortium, and implemented independently by various organisations in the form of web browsers, which are tools to retrieve and view HTML/CSS documents. The task of retrieving, parsing, and rendering an HTML document is very important and very complex. The specification that defines how an HTML page should be rendered (CSS) is more than 150,000 words long. It is well known that all web browsers render HTML differently in subtle ways, and we argue that the size and non-deterministic nature of the specification contributes to this problem. This project demonstrates a new method of a) describing the CSS specification, and b) implementing a CSS-compliant HTML renderer, that employs dynamic attribute grammars to achieve both tasks. Attribute grammars assist in specification by describing the relationship between the Document Object Model and its visual representation deterministically, and greatly reduce the gap between specification and implementation by writing the specification in an executable format. Furthermore, we show informally that this executable specification is human-readable, which we argue is because attribute grammars are the *natural* language to describe CSS.

Contents

1 Introduction

This chapter provides a brief introduction to the technologies that make up the Web, and a thorough introduction to attribute grammars, including an explanation of the particular attribute grammar notation used in this thesis.

Chapter 2 gives an overview of related work.

Chapter 3 gives a thorough analysis of the CSS layout problem, and proposes that attribute grammars are an ideal language for describing its semantics.

Chapter 4 explains the semantics of a representative subset of CSS, and expresses those semantics using our attribute grammar notation.

Chapter 5 demonstrates that the attribute grammar specification shown in Chapter 4 is executable, showing screenshots of simple documents rendered using attribute grammars. Chapter 5 also explores the human-readability of an attribute grammar specification for CSS.

Chapter 6 gives directions for future work, and provides a very brief overview of the work contained in Chapters 1 to 5.

The key ideas in this thesis

- An attribute grammar based specification for CSS would be more human-readable than the existing (prose) specification, when the reader is a web or browser developer.
- Such a formalisation would be deterministic and executable.
- It would be feasible to create an attribute grammar formalisation for the whole CSS layout specification.

The key outcomes of this work

- A demonstration that a significant subset of the CSS layout specification can be formalised using attribute grammars.
- A comparison of the existing CSS layout specification with our attribute grammar alternative.
- A demonstration of an executable attribute grammar formalisation.

1.1 The Web

The Web is a group of specifications that describe a way to transfer and interact with documents over the internet. The core technologies that make up the Web are HTTP, HTML, CSS, and JavaScript.

The transfer protocol used in the Web is HTTP (Hyper-Text Transfer Protocol). The standard document format is HTML (Hyper-Text Markup Language), which describes structured documents. CSS (Cascading Style Sheets) is a language and standard for specifying how HTML documents are rendered. JavaScript is a language embedded in browsers to allow an HTML document to dynamically modify its structure.

HTTP

HTTP (Hyper-Text Transfer Protocol) is an application protocol for the stateless transfer of documents over TCP. The HTTP protocol is a light-weight text-based standard made up of requests and responses, designed to allow users to interact with documents described by a URI.

The initial version of HTTP only included the GET method [1], and described HTTP requests as idempotent. The current version of HTTP (1.1) states that the methods GET, HEAD, PUT, and DELETE should be idempotent, and have no side-effects.

HTML

HTML [2, 3] (Hyper-Text Markup Language) was developed alongside HTTP, and is the standard markup language used for creating web pages. HTML is similar to XML, and can be queried by XQuery (Section 2.2.1) under the right circumstances.

JavaScript

ECMAScript [4] (informally known as JavaScript) is unofficially the 'language of the Web'. JavaScript is supported by all modern web browsers (including mobile web browsers), and is used to implement most interactive web pages on the Web today.

JavaScript is a prototype-based dynamically typed scripting language with first class functions. JavaScript is is not purely functional because side-effects are supported, however there is strong support for functional-style programming in JavaScript [5].

One of the core requirements for a web-scripting language is the ability to interact with the Document Object Model (DOM) [6]. One of the most ubiquitous JavaScript libraries on the Web is jQuery [7]. jQuery introduces a syntax for selecting and transforming elements in the DOM that is reminiscent of term rewriting. While not in the scope of this thesis, the relationship between DOM manipulation and term-rewriting is relevant to the broader theory of a 'Functional Web', and is a topic for future research following this thesis.

CSS

CSS (Cascading Style Sheets) [8, 9] is a stylesheet language for specifying the way in which an HTML page is rendered. CSS is covered in detail in Section 1.2.

1.2 The CSS specification

The official HTML standard states "User agents are not required to present HTML documents in any particular way". The de facto standard in modern web browsers is to use the CSS layout method. As a result, the CSS standard is fundamental in a browser's ability to represent an HTML document visually.

There are two core components of the CSS standard:

The stylesheet language. Most web authors know CSS as a tool for specifying the particular display properties of various elements in an HTML document.

CSS as a stylesheet language is well researched and well understood in the web development community. This thesis does not focus on CSS's stylesheet language, although the way that a CSS stylesheet's rules change the properties of a document bears a striking resemblance to operations carried out in term-rewriting (see Section 6.1). The fact that these rules and jQuery both lend themselves to the same DSL is no surprise, given that jQuery's element selection method is inspired by CSS[1].

The display standard. While the stylesheet language offers a web author a list of display properties to adjust, the CSS standard is also responsible for specifying exactly how a compliant browser must use these rules to create the visual representation of the document.

The display standard provided by CSS is of central importance to this thesis. This standard defines the specific relationship between the CSS properties of an element

[1] The official jQuery website states "Borrowing from CSS [versions] 1-3, and then adding its own, jQuery offers a powerful set of tools for matching a set of elements in a document." `http://api.jquery.com/category/selectors/` (June 2014).

and its on-screen representation. This thesis examines the standard, and demonstrates a new formalisation of the standard based on attribute grammars.

The CSS display specification is provided informally in a number of technical reports published by the W3 Consortium [10]. Not only are the semantics of CSS-compliant layout written in informal prose, this prose is spread over many separate technical reports, and amounts to roughly 150,000 words[2] in total. The CSS layout specification is in no way concise or deterministic, with some aspects of the layout being explicitly undefined, and multiple interpretations existing of the aspects that do intend to be well-defined. It is well-recognised that the current form of the specification is ambiguous.

It is well-known that different layout engines can produce inconsistent results when using CSS layout rules to render the same HTML document. We suggest that the non-deterministic nature of the CSS specification contributes to this problem. Currently, no alternative formalisations exist, although Meyerovich *et al.* [11] provide a formalisation of a subset of CSS in attribute grammar form.

1.2.1 CSS layout implementations

All modern web browsers implement the CSS display standard for rendering HTML documents. Given the magnitude of this task, a number of 'layout engines' exist as stand-alone projects. Each major web browser delegates the document rendering task to one of these engines.

Some of the more popular layout engines include:

- Gecko (employed by Mozilla's Firefox)
- Trident (employed by Microsoft's Internet Explorer)
- KHTML (employed by KDE's Konqueror)
- WebKit (employed by Apple's Safari)
- Presto (employed by Opera Software's Opera)
- Blink (employed by Google's Chrome)

[2] A rough word count of the following documents: CSS Level 2 Revision 1, Selectors Level 3, CSS Namespaces, CSS Color Level 3, Media Queries, CSS Style Attributes taken from http://http://www.w3.org/Style/CSS/

1.3 Attribute grammars

1.3.1 Production-based attribute grammars

Attribute Grammars (AGs) were first introduced by Knuth in 1968 as a tool for decorating trees created by context-free grammars in compilers [12]. Originally attribute grammars were written as equations alongside CFG production rules, and were only applicable to trees created by a CFG (therefore trees that conform to a specific 'shape').

A static attribute grammar evaluator will analyse the rules of an attribute grammar and generate a tree traversal algorithm to find values for all attributes on all nodes. The evaluator will try to find the most efficient traversal for the given dependency graph, as well as checking for dependency loops and other non-terminating configurations. Some evaluators will generate a traversal algorithm that will perform on any tree in the grammar (at the compiler's compile time), and others will generate a traversal algorithm for the given tree (at the compiler's run time) [13].

Given that attribute grammars were historically written alongside CFG production rules, there have been two distinct ways to define an attribute grammar rule. If an attribute equation for an `E` node is written alongside a production rule for `E`, this is known as a *synthesized* grammar rule. `E`'s production rule defines `E`'s children, so the associated equation has access to the attributes of `E`'s children (see listing 1.1). Since no CFG production rules refer to the parent of a node, the only way to write attribute grammar rules with 'upward' dependencies is by writing the attribute equation in the parent's production rule. Assuming `P` is a parent of `E`, an *inherited* attribute equation for an `E` node will be written alongside `P`'s production rule (see listing 1.2).

Listing 1.1: A synthesized attribute grammar rule.

```
E → A + B [E.total = A.total + B.total]
```

Listing 1.2: An inherited attribute grammar rule.

```
P → E + F [E.depth = P.depth + 1]
```

1.3.2 Tree-based attribute grammars

While pairing attribute grammar rules with production rules has its advantages in the context of compiler design, separating the two allows for an attribute grammar to operate on a tree of any shape. Some modern evaluators allow for attribute grammar rules to be written without making any assumptions about the structure of the tree. Such systems require notations for referencing a node's parents and/or children. The result is a much wider scope for the application of attribute grammars. See Figure 1.1 in Section 1.3.3 for an example of a tree-based attribute grammar.

When attribute grammar rules are no longer paired with production rules, the distinction between inherited and synthesized grammar rules becomes less important. CFG-based attributes could only have either upward *or* downward dependences (not both), because a production rule only spans two 'generations' of a tree, and therefore there are only symbols available for two generations. When an evaluator allows for special right-hand-side (evaluation) notations to access parent and child elements, it becomes possible to have both upward *and* downward dependencies.

The absence of a strong tree structure and the presence of simultaneous upward and downward dependencies makes it considerably more difficult for an evaluator to find an optimal tree-walk for evaluating all attribute values. One solution to this problem is to defer attribute evaluation to runtime, and to allow the dependency graph to dictate the tree-walk, only generating the values of attributes when they are needed. The result is that attributes behave like functions or lazily-evaluated variables. Memoisation becomes an important optimisation in these instances.

As with CFG-based attribute grammars, the evaluator is responsible for detecting dependency loops. Occasionally dependency loops are terminating loops however, and some evaluators can accommodate this. One of the most important jobs of an evaluator is to account for complex dependency graphs.

When an attribute grammar language is given a powerful right-hand-side expression language and allowed to operate on a tree of any shape, it might understandably be confused with a general purpose programming language. In an object-oriented programming context, if objects in a tree have methods that call upon the methods of parent and child objects, how are these different to attribute grammars? There are three reasons that tree-based attribute grammars should still be considered a domain-specific language:

Memoisation Attribute grammars are not executed, but evaluated by an attribute grammar evaluator. One of the tasks carried out by this evaluator is memoisation, which prevents attribute values to be needlessly recalculated.

Pure Functionality In most instances, an attribute equation is merely an expression with no side-effects. An evaluator will often use side-effects for optimisation purposes (such as memoisation), but an attribute equation is usually required to be idempotent.

Dependency Analysis An evaluator does not blindly execute the equations in attribute grammar rules, but first performs dependency analysis to ensure that the dependency graph is sensible. When the value of an attribute is externally requested, the evaluator will use the equations provided by attribute grammar rules to find an attribute's value.

1.3.3 Our attribute grammar notation

A number of attribute grammar notations exist, mostly in executable form. For a CSS formalisation, the attribute grammar notation used does not need to be *directly* executable,

but must be deterministic and unambiguous. We also aim to make our notation easily translatable into an executable attribute grammar notation.

We will demonstrate our notation with a solution to a modified 'repmin' problem. The repmin problem usually requires creating a clone of a binary tree, with all leaf values being replaced with the minimum leaf value from the whole tree. In this instance, instead of cloning the tree, we will merely decorate all nodes with an attribute called `treemin` that represents the global minimum of all leaf nodes `value` attributes. We call this the 'treemin' problem.

We assume a tree of `Node` elements, with subtypes of `Node` being `Fork` and `Leaf`. `Leaf` elements have a `value` property, and `Fork` elements have a `children` property which is an iterable list of `Node` elements.

Figure 1.1: A treemin solution in attribute grammar notation

```
node.globmin : Int
  (e : Fork) => min(c.globmin for c in e.children)
  (e : Leaf) => e.value

node.treemin : Int
  (e is root) => e.globmin
  (e)         => e.parent.treemin
```

Figure 1.1 shows an attribute grammar solution to the treemin problem. The `globmin` attribute calculates the minimum value for all leaves in an element's subtree. The first line in Figure 1.1 denotes that `globmin` is an attribute that evaluates to an integer.

`(e : Fork)` means 'if the current element is a `Fork` node'. For all `Fork` elements, the `globmin` attribute is evaluated as `min(c.globmin for c in e.children)`, which is the minimum value of its children's `globmin` values. We call the line beginning with '`(e : Fork)`' the first *rule* of the attribute grammar. To evaluate the `globmin` attribute, you consider the rules from top to bottom. If the expression on the left of the `=>` matches, the expression on the right is returned.

If the first rule does not match, the second is considered. `(e : Leaf)` means 'if the current element is a `Leaf` node'. For `Leaf` elements, the `globmin` attribute evaluates to that element's `value` property; `e.value`.

`treemin` should resolve to the global minimum value at `Leaf` nodes. The root element of the entire tree will have the global minimum stored in its `globmin` attribute. To access this value, all nodes should recursively recall their parent's `globmin` value. The first rule in `treemin` begins with `(e is root)`, which means 'if the current element is the root element'. The root element's `treemin` is simply its `globmin` value.

For all other nodes, `treemin` should reference its parent's `treemin` attribute value. A rule that begins with the simple expression `(e)` will match any element. This is known as the *default rule*. `treemin` has one expression for the root element, and one expression for all other elements.

If a node could have a type other than `Fork` and `Leaf`, then the `globmin` attribute would not be well-defined, as the behaviour for that third type would be undefined. For this reason, it is a good idea to make the last rule the *default rule* so that an attribute can always be evaluated. It might be appropriate to change the attribute grammar in Figure 1.1 to make it more typesafe. However, some attribute grammar evaluators (such as Kiama, as discussed in Section 5.1) will automatically detect grammars that are not well-defined, and report an error if a third child of `Node` exists.

The following points summarise the features of our attribute grammar notation, including some that have not yet been explored.

- An attribute is defined by a number of rules, which are tested from top to bottom.
- If the expression on the left-hand-side of the `=>` symbol matches, the right-hand-side equation is evaluated and returned.
- `(e : MyType)` matches if the element is a `MyType` node.
- `(e is root)` matches if the element has no parent, and `(e is first)` matches if the element has no previous siblings.
- `(e.globmin > 4)` matches if the element's globmin is greatar than 4. Any simple boolean expression can be placed on the left-hand-side of a grammar rule.
- `(e)` matches any element.
- The identifier found in the left-hand-side boolean expression can be used in the right-hand-side expression to refer to the element under scrutiny. In the examples shown in this document, this identifier is always `e`.
- The identifier has in-built reference attributes `prev`, `next`, and `parent`, which provide access to the the previous, next, and parent nodes of the given element respectively. `children` returns an iterable list of a node's children.

2 Related Work

Neither the Web nor functional programming are new ideas. Both have been around for decades and have a strong research and industry backing. A number of systems exist that attempt to bring declarative principles into the Web stack.

2.1 Functional frameworks for the Web

2.1.1 Melchior

Melchior [14] is a Haskell based functional reactive programming (FRP) framework for declaratively building web applications. The tool essentially allows programmers to write FRP code in a Haskell-like language, which will compile to JavaScript. The focus of Melchior is to provide event handling in a declarative manner.

2.1.2 Happstack

In the same spirit as Melchior, Happstack's [15, 16] goal is to allow web developers to build systems in a declarative manner, although Happstack developers are able to perform server-side scripting in Haskell, as well as client-side scripting assistance via Jmacro. Happstack is a stand-alone HTTP server, which allows server-side developers to interact closely with HTTP.

2.1.3 Yesod

Yesod [17, 18] is another web framework built on Haskell. A key feature is compile-time typechecking, which ensures type safety, helping reduce the frequency of common Web application runtime errors. Yesod also makes use of template Haskell for metaprogramming.

Yesod allows developers to define application logic and to construct HTML responses, both in a declarative manner.

2.1.4 Other Haskell-based frameworks

A number of compile-to-JavaScript frameworks exist beyond Melchior. These include Fay [19], GHCJS [20], UHC [21], and Elm [22]. There are also a number of complete Haskell-driven web application frameworks that exist beyond Happstack and Yesod. These include Snap [23], Scotty [24], MFlow [25], Salvia [26], and Miku [27].

2.1.5 Relevance

The projects discussed above share a general aim of 'combining' functional programming with Web stack technologies. This thesis proposes that CSS (a Web stack technology) is most naturally described using a declarative DSL: attribute grammars. We recommend that the CSS standard should be written using declarative languages, whereas the discussed projects seek to use declarative programming languages to interface with the existing Web stack technologies.

Another contrasting factor is that instead of proposing a new way to develop *new* systems, we are proposing a new way to redevelop *old* systems, namely web browsers. More specifically CSS layout engines.

2.2 Functional Web-related standards

2.2.1 XQuery

XQuery [28, 29] is a functional DSL for querying and transforming XML documents. This technology is relevant to the Web stack because XML and HTML are closely related[1], and XQuery can be used to query and transform XHTML[2] (or any HTML that is XML compliant).

Being a functional Web-related technology, XQuery holds some relevance to this thesis. XQuery is an example where a functional style has proven appropriate when accessing and transforming documents, a task that is also performed in the application of CSS rules to a document (Section 1.2). This thesis aims to apply the declarative programming style to the specification and implementation of CSS.

[1] XML and HTML are both derivatives of SGML.

[2] XHTML [30] is a standard that aims to be both valid HTML and well-formed XML.

2.3 Core web technologies implemented functionally

2.3.1 Using attribute grammars for CSS layout

Kastens *et al.* [31] demonstrate the applicability of attribute grammars in HTML rendering, specifically in the computation of the width and height of a textual table. The demonstrated solution is implemented in LIDO [32] for LIGA [33], which is a language-independent generator for attribute grammars.

Swierstra *et al.* [34] also demonstrate the applicability of attribute grammars to the table layout problem. This solution is put forward as part of an argument for Haskell as "one of the most promising candidates for the 'ideal extensible language' ". The demonstrated solution is implemented in a un-named proprietary notation.

Meyerovich *et al.* [11] explore the use of attribute grammars as an aid to parallelising the CSS layout process. The authors implement a subset of the CSS layout scheme (which they name BSS) using attribute grammars, in an effort to speed up render time. This work bears a resemblance to the work of this thesis, but is motivated by a different goal. While Meyerovich desires attribute grammars for their parallelisation benefits, we desire attribute grammars for their concise expressiveness. See the beginning of Chapter 4 for a deeper discussion of the differences between Meyerovich's work and ours.

The three works listed above demonstrate how attribute grammars can be applied to subsets of the CSS layout problem. This thesis aims to build upon the listed works to demonstrate that attribute grammars not only *can* be used for CSS layout, but are an ideal language for implementation due to their expressiveness. This thesis also aims to show that CSS *semantics* can be expressed in attribute grammar form, thus closing the gap between specification and implementation that causes so many compatibility problems between modern web browsers. The primary focus of the work presented in this thesis is that attribute grammars are an ideal language for specifying the semantics of CSS layout.

3

Why CSS needs attribute grammars

3.1 An analysis of the CSS layout problem

3.1.1 From a tree to a rendered page

The structure and presentation of a web document are entirely separate. The HTML 5.1 standard states "User agents are not required to present HTML documents in any particular way" [35], making it clear that the HTML spec does not officially cover visual display. However, the same spec also says "In general, user agents are expected to support CSS, and many of the suggestions in this section are expressed in CSS terms"; CSS is the de facto standard for rendering HTML documents.

CSS is a standard for styling an HTML document using cascading style sheets. The CSS spec lists a number of legal CSS properties, and describes how these properties are used to create a visual representation of an HTML document. The specification describes methods to calculate the exact way an element should be displayed in a compliant browser.

An HTML document has a nested nature, so it is appropriate to represent it as a tree. The Document Object Model (DOM) [6] is a tree representation of an HTML (or XML) document, and is relied upon by the HTML spec.[1] Similarly, the CSS spec instructs a user agent to treat an HTML document as a tree[2]. The document tree is then decorated, such that every node has a value for every CSS property[3].

[1] "The conformance criteria of HTML implementations are defined, in this specification, in terms of operations on the DOM" (2.2.2 Dependencies, in [36].)

[2] "In this model, a user agent processes a source by going through the following steps: 1. Parse the source document and create a document tree." (2.3 The CSS 2.1 processing model, in [9].)

[3] "Once a user agent has parsed a document and constructed a document tree, it must assign, for every element in the tree, a value to every property that applies to the target media type." (6.1 Specified, computed,

An HTML document may contain thousands[4] of elements, each of which is decorated with hundreds[5] of CSS properties. That means hundreds of thousands of CSS properties must be evaluated for every page render. These properties are things like `border-width`, `display`, and `text-align`. The values for these properties can be specified by style sheets provided by the document author, the user, or the user agent. There are cascading and inheritance rules that define how different style sheets interact to come to a 'final' property value, but this process is outside the scope of this thesis. We are more concerned with the way these properties are used to create a visual representation of the document tree.

3.1.2 Different kinds of properties

Let us call the visual representation of an HTML document (according to CSS) a page. All visible elements in a document take up a rectangle on the page. Their positions on the page are determined by their position in the document tree, their CSS property values, and the property values of other elements in the tree. While there are CSS properties named `top`, `left`, `width` and `height`, these properties do not directly define the element's exact size and position on the page.

The algorithm for deciding the element's exact size and position is very complex, and broken up into many intermediary steps. These steps are detailed by the CSS spec, and describe a number of properties of an element, such as its *box type*, *positioning scheme*, *containing block*, *margin edge*, etc. We will call these properties "hidden properties", as they are part of the inner workings of a renderer and not able to be directly set by a document or style sheet. The spec describes how to determine an element's hidden properties, and how these hidden properties determine an element's position and appearance on the page. In many cases, the values of these hidden properties are determined by the *context* of the element; not merely the properties of the element: properties of ancestors, descendants, and siblings must be examined to determine a value.

Non-hidden CSS properties can be given a value by a style sheet or by the user or user agent, but that's not the whole story. To quote the spec:

> "The final value of a property is the result of a four-step calculation: the value is determined through specification (the 'specified value'), then resolved into a value that is used for inheritance (the 'computed value'), then converted into an absolute value if necessary (the 'used value'), and finally transformed according to the limitations of the local environment (the 'actual value')." ([9] Section 6.1)

These four separate values are resolved by fixed transformations or by consulting the values of related elements in the tree. In the case of resolving inheritance (computed value)

and actual values, in [9].)

[4]The page at `http://reddit.com` at the time of this writing contained 1730 elements (counted by tags).

[5]The CSS 2.1 property index table lists 94 properties [9], while Jens Meiert's index (which is referenced by the W3C) lists 331 [37].

the parent element is considered, and in the case of resolving percentage values into pixel values (used value), an ancestor known as the *containing block* is considered. In some instances, the computed and used values of a property are affected by child elements in the tree.

There are also cases where both parents *and* children must be considered to resolve a single property value. Consider the following example, where a single hidden property has dependencies on parent, child, and sibling elements.

> "If a child C of a tabular container P is an anonymous inline box that contains only white space, and its immediately preceding and following siblings, if any, are proper table descendants of P and are either 'table-caption' or internal table boxes, then it is treated as if it had 'display: none'. A box D is a proper table descendant of A if D can be a descendant of A without causing the generation of any intervening 'table' or 'inline-table' boxes." ([9] Section 17.2.1)

3.1.3 A dependency graph

The values of all properties of an element are calculated using values from associated elements. The four stages for a non-hidden property (specified, computed, used, and actual) have mostly upward dependencies (depending on parents or higher ancestors), with some downward dependencies. Hidden properties have more complex dependencies, with a single property often relying upon properties from children and siblings as well as parents.

It's also worth noting that a property's dependees are not always in adjacent nodes (dependees being the properties that are depended upon). There are a number of hidden properties in CSS that refer to a non-adjacent dependee node, such as *containing block* or *offsetParent*. The spec has created hidden properties that we will call "reference properties" to describe a dependee element that is not always adjacent to the source element.

Computing a single property value with complex dependencies is one thing, but the problem becomes considerably more difficult when those dependencies have their own complex dependencies. The result is a complex dependency graph. If all dependencies were to adjacent nodes then the dependency graph would look like a tree, but this is not the case. There are a number of problems raised by a complex dependency graph. The order in which properties are evaluated becomes important, as all of a property's dependencies must be resolved before a value can be found. A spec can be (and is) written without explicitly being concerned with the order of property evaluations, but implementations need to be careful to always evaluate a dependee property before the property that has the dependency.

A complex dependency graph brings with it a high risk of non-terminating dependency loops. In a project the size of CSS, it is difficult to know if dependency loops exist, given the informal nature of the specification. The nature of the problem makes a specification hard to write in a manner that is both human readable and deterministic. The specification authors

have chosen to use prescriptive language in an attempt to make the spec deterministic, and have broken the problem down into intermediary stages (using what we call *hidden properties*) to make the problem more approachable.

3.2 An alternative formalisation

We assert that attribute grammars are the natural notation for formalising CSS layout.

3.2.1 Specification

A summary of the explored aspects of the CSS layout spec:

- A document is a tree.
- At every node, the four stages of each CSS property must be evaluated, as well as a number of hidden properties.
- Evaluating a property involves examining properties of related nodes.
- Some (hidden) properties are references to nonadjacent nodes in the tree.

Attribute grammars are a perfect match for a problem like this. AGs are designed for decorating trees with values that depend on values from nearby nodes. Further, AG languages exist that support reference attributes [38], which enable a property to be a reference to a nonadjacent node.

The size of the CSS layout problem has caused the authors of the specification to break the entire process down into manageable steps. These steps usually involve defining some property or relationship that has been outlined in isolation, and then used in the definition of further properties and relationships. This strategy is similar to the attribute grammar strategy, meaning the original spec can often be translated into attribute form without redefining the intermediate steps.

3.2.2 Dependency resolution

The spec does not need to worry about the order in which different properties are evaluated, but an implementation does. When dealing with implementing a system with a complex dependency graph, a number of important questions are raised, including:

1. Which properties need to be calculated in which order, to make sure that all data is available when required?
2. Will a single property be recalculated when its value is requested multiple times?
3. How can we be sure that there are no dependency loops in the specification?

These questions don't have trivial answers in a procedural setting. However, attribute grammars were designed specifically for solving problems of this nature. Attribute evaluator order is handled by the attribute grammar evaluator, so that all of an attributes dependencies are available when they are needed. Memoisation (covered in Section 1.3.2) is employed by some evaluators to prevent attribute recalculation. Dependency loops are not resolved by evaluators, but are pointed out to the user when they occur, at either compile- or run-time.

3.2.3 An example

Here is an example of a rule in the CSS spec, that describes the `offsetTop` property of a DOM element. `offsetTop`, in simple terms, describes the y-coordinate of an element. This property defined by W3C as follows:

> " The `offsetTop` attribute must return the result of running these steps:
>
> 1. If the element is the HTML `body` element or does not have any associated CSS layout box return zero and terminate this algorithm.
> 2. If the `offsetParent` of the element is null return the y-coordinate of the top border edge of the first CSS layout box associated with the element, relative to the initial containing block origin, ignoring any transforms that apply to the element and its ancestors, and terminate this algorithm.
> 3. Return the result of subtracting the y-coordinate of the top padding edge of the first CSS layout box associated with the `offsetParent` of the element from the y-coordinate of the top border edge of the first CSS layout box associated with the element, relative to the initial containing block origin, ignoring any transforms that apply to the element and its ancestors. " [39]

Consider the specification above. It reads as a procedural algorithm ('return the result of running these steps...'), but examine the third point in the definition. The `offsetTop` property is dependent upon the `paddingEdge` property of the `offsetParent` element. To find this value, the `offsetParent` element must first be identified, and then the `offsetParent`'s padding edge must be identified. Further, to calculate the `offsetParent`'s padding edge, all of the child elements of the `offsetParent` must be examined.

Figure 3.1: `offsetTop` in attribute grammar form

```
elem.offsetTop : Int
  (e:Body)                  => 0
  (e.offsetParent = null)   => e.boxBorderTop -
                               ICB.top
  (e)                       => e.boxBorderTop -
                               e.offsetParent.boxPaddingTop - ICB.top
```

`offsetTop` is an example of a CSS property, `offsetParent` is an example of a reference property, and `paddingEdge` is an example of a hidden property. The definition of `offsetParent` is similarly complex, and not shown here. This small sample of the spec is a demonstration of how many different nodes and properties must be considered to evaluate a single property. The prose specification itself is quite complex, and a procedural implementation is likely to be even more complex. In contrast, the attribute grammar specification of the same property (shown in Figure 3.1) contains all of the relevant information from the prose specification while being considerably more concise.

Note the use of `ICB` in Figure 3.1. `ICB` refers to the initial containing block, which has been implemented as an external variable in this instance (and in all attribute grammars shown in this document). For evaluators that do not allow the use of external variables, `ICB` could be implemented as a reference attribute or an attribute that is computed at the root and passed down to all other nodes.

4

CSS LAYOUT WITH ATTRIBUTE GRAMMARS

In this chapter we build a layout specification from scratch, using attribute grammar notation. This specification is a small but representative subset of the CSS layout problem. This spec is broken down into stages, beginning with the smallest possible subset of the layout scheme and building up from there. This is designed to be a translation of the original spec, but does not claim to conform to all of the subtle details of the CSS layout system. This subset is a proof-of-concept implementation of a CSS layout specification.

The five stages of CSS are: block boxes, inline boxes, anonymous line boxes, the box model, and right floats. Each stage builds upon the previous stage, with the exception of stage 5 (right floats) which builds upon stage 3 for the sake of clarity. In each of the following sections, an aspect of the CSS layout system will be explained and relevant excerpts from the prose CSS specification will be presented. An attribute grammar notation of each stage will also be shown and explained in each section from 4.1 (inline boxes) to 4.5 (right floats).

This work is similar to that presented by Meyerovich *et al.* [11] in that CSS broken into stages and presented as attribute grammars. The primary difference between this work and Meyerovich's is its purpose; Meyerovich *et al.* explores the use of attribute grammars as a tool for parallelisation when implementing web browsers. We explore the use of attribute grammars as a tool for clear and deterministic specification of the layout problem. The three stages that Meyerovich *et al.* implemented were:

- **Vertical and horizontal stacking boxes.** Vertical stacking boxes are block-level boxes, and horizontal stacking boxes are inline-level boxes. Meyerovich *et al.* specify that a vertical box can contain either vertical boxes *only* or horizontal boxes *only*. Also

they specify that horizontal box can contain vertical boxes only. In the official HTML specification, any box can be a sibling of any other box, and this creates complications such as anonymous line boxes, which are dealt with in our system but not Meyerovich's. We allow the document tree to contain any nodes in any order, which is truer to the HTML specification.

- **Shrink-to-fit sizing.** We implement shrink-to-fit sizing in stage 1 (Section 4.1), where a block-level box with a specified height of 'auto' will have a computed height just large enough to contain all of its children.

- **Left floats.** We agree with Meyerovich when he states that floating elements are the most complex aspect of the CSS specification. Meyerovich implemented left floats only (not right). We implement right floats only (not left). In this regard, our works are complementary. However, the type of language that Meyerovich describes as 'attribute grammars' differs greatly from the attribute grammars used in the work presented here. This disparity is caused by our greatly different goals; Meyerovich aims to write fast and parallelisable CSS layout engine, and we aim to write a clear and deterministic specification for CSS layout, which happens to be executable (but not necessarily fast).

The rest of this chapter will explain our CSS stages and their attribute grammar specifications. Chapter 5 will demonstrate that these specifications are executable.

4.1 Stage 1: Block-level boxes only

Stage 1 only contains block-level boxes. Sibling block-level boxes stack vertically and can be nested. Given that the CSS spec does not provide guidelines for implementing a simplified layout machine such as this, a number of assumptions have been made to fill in the gaps. The following specs are not intended to demonstrate a direct translation of the original spec (as was demonstrated for `offsetTop` above), but rather to demonstrate that the nature of the layout problem can be fully specified using attribute grammars. See Figure 5.5 in Section 5.1.2 for a visual representation of stage 1.

4.1.1 The existing specification

The exact position and size of an element can not be narrowed down to one section of the spec, as there are so many variations on the behaviour caused by more advanced features. The following is a collection of quotes from the spec that outline the basic behaviour of block-level boxes:

- "Block-level boxes participate in a block formatting context. [...] In a block formatting context, boxes are laid out one after the other, vertically, beginning at the top of a containing block." ([9] Section 10.8)

- "The following constraints must hold among the used values of the other properties: 'margin-left' + 'border-left-width' + 'padding-left' + 'width' + 'padding-right' + 'border-right-width' + 'margin-right' = width of containing block"[1] ([9] Section 10.3.3)

- " 'Auto' heights for block formatting context roots: [...] If it has block-level children, the height is the distance between the top margin-edge of the topmost block-level child box and the bottom margin-edge of the bottommost block-level child box." ([9] Section 10.6.7)

4.1.2 In AG form

Figure 4.1: CSS stage 1 in attribute grammar form

```
e.boxX : Int
  (e) => ICB.left

e.boxWidth : Int
  (e) => ICB.width

e.boxRelY : Int
  (e is first) => 0
  (e)          => e.prev.boxRelY + e.prev.boxHeight

e.boxY : Int
  (e is root) => ICB.top + e.boxRelY
  (e)         => e.parent.boxY + e.boxRelY

e.boxHeight : Int
  (e.height = auto) => sum(c.boxHeight for c in e.children)
  (e)               => e.height
```

Figure 4.1 shows an attribute grammar notation for how block-level boxes should be displayed on the page. In this spec, `boxX`, `boxY`, `boxWidth`, and `boxHeight` provide the exact size and position of a box that represents the element on-screen. `boxRelY` is a temporary attribute that specifies a block's y-position relative to its parent. `ICB` references the *initial containing block*, which in this context is the display window or canvas.

The horizontal size and position of all block-level boxes in this model are the entire width of the initial containing block, as you might expect from an un-styled `div` (which is a block-level box). All boxes sit below their previous sibling. The height of the box can be defined by the `e.height` property (externally), and an `e.height` value of `auto` will result in a `boxHeight` value just large enough to contain all of its children. A definition for `e.height` is not shown in the above grammar, as it should be set by a stylesheet, which is outside the scope of this work.

[1] Since stage 1 does not implement margin/border/padding (see Section 4.4), this equation can be simplified to `width = width of containing block`.

4.2 Stage 2: Inline-level boxes

Stage 2 includes inline-level boxes. Sibling inline-level boxes stack horizontally, and 'spill' onto a new line when necessary. Block-level and inline-level boxes can be siblings.

4.2.1 The existing specification

As with block-level boxes, the following collection of quotes represents an outline of the existing CSS spec's positioning scheme for inline-level boxes:

- "Inline-level elements are those elements of the source document that do not form new blocks of content; the content is distributed in lines." ([9] Section 9.2.2)
- "In an inline formatting context, boxes are laid out horizontally, one after the other, beginning at the top of a containing block." ([9] Section 9.4.2)
- "When an inline box exceeds the width of a line box, it is split into several boxes and these boxes are distributed across several line boxes." ([9] Section 9.4.2)

4.2.2 In AG form

Figure 4.3 shows an attribute grammar notation for both block-level and inline-level boxes in the same document. In this spec, the on-screen rectangle representing an element is defined by `boxAbsLeft`, `boxAbsTop`, `boxWidth`, and `boxHeight`. Temporary attributes such as `boxRelTop` define intermediary values, such as a box's position relative to its parent's origin. `isLineHead` is used to decide when an inline-level box must be placed below its previous sibling, due to 'line overflow'. All inline-level boxes that are not `isLineHead` sit to the right of their previous sibling.

There is an issue with this design. If you examine `boxRelTop`, the second formula (where `e.isLineHead = true`) uses `e.prev.boxRelBottom` (the bottom of the previous sibling's box) to define the top of a box on a 'new line'. But what if boxes in the first line aren't all of the same height? This layout model will produce a result as shown in Figure 4.2. Stage 3 resolves this problem.

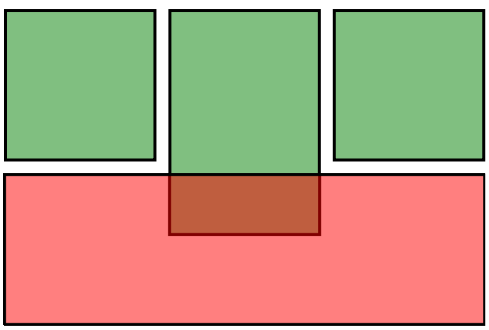


FIGURE 4.2: Incorrect inline-level box placement

Figure 4.3: CSS stage 2 in attribute grammar form

```
e.isLineHead : Boolean
  (e is first)       => true
  (e : Block)        => true
  (e.prev : Block)   => true
  (e.prev : Inline)  => (e.prev.boxRelRight + e.boxWidth) >
                            e.parent.boxWidth

e.boxRelLeft : Int
  (e : Block)              => 0
  (e.isLineHead = true)    => 0
  (e)                      => e.prev.boxRelRight

e.boxRelRight : Int
  (e) => e.boxRelLeft + e.boxWidth

e.boxRelTop : Int
  (e is first)             => 0
  (e.isLineHead = true)    => e.prev.boxRelBottom
  (e)                      => e.prev.boxRelTop

e.boxRelBottom : Int
  (e) => e.boxRelTop + e.boxHeight

e.boxWidth : Int
  (e : Block)  => ICB.width
  (e : Inline) => e.width

e.boxHeight : Int
  (e.height = auto) => e.height
  (e)               => max(c.boxRelBottom for c in e.children)

e.boxAbsLeft : Int
  (e is root) => ICB.left + e.boxRelLeft
  (e)         => e.parent.boxAbsLeft + e.boxRelLeft

e.boxAbsTop : Int
  (e is root) => ICB.top + e.boxRelTop
  (e)         => e.parent.boxAbsTop + e.boxRelTop
```

4.3 Stage 3: Line boxes

Stage 3 introduces an attribute that mimics the behaviour of a line box. Overflowing inline boxes no longer cause overlaps as shown in Figure 4.2.

4.3.1 The existing specification

The spec describes a sequence of inline blocks as being made up of 'line boxes', which are anonymous block-level boxes that tightly wrap a sequence of inline boxes.

- "The rectangular area that contains the boxes that form a line is called a line box." ([9] Section 9.4.2)
- "A line box is always tall enough for all of the boxes it contains." ([9] Section 9.4.2)
- "The line box height is the distance between the uppermost box top and the lowermost box bottom." ([9] Section 10.8)

4.3.2 In AG form

Figure 4.5 shows an attribute grammar notation for both block-level and inline-level boxes in the same document, with sequential inline-level boxes being contained by an anonymous line box. In this spec, the `lineBottom` attribute has been introduced and used by `boxRelTop`. While no 'line box' element has been explicitly created, the `lineBottom` attribute represents the bottom of a theoretical line box that contains a sequence of inline boxes. `lineBottom` is a running maximum of the `boxRelBottom` attribute, so that the `lineBottom` value of *the last inline box in a line* represents the bottom edge of the theoretical line box. The theoretical line box is represented by the blue dotted rectangle in Figure 4.4. This rendering model assumes that colinear inline boxes should have their top edges aligned.

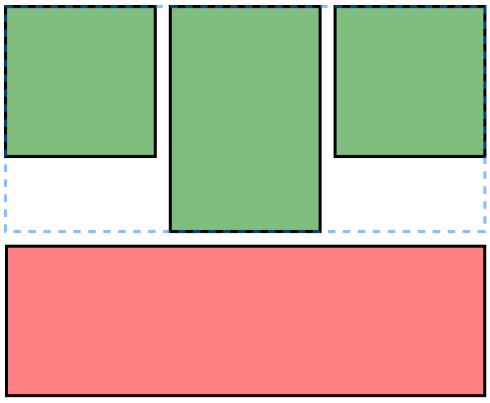


FIGURE 4.4: Corrected inline-level box placement

Figure 4.5: CSS stage 3 in attribute grammar form

```
e.isLineHead : Boolean
  (e is first)       => true
  (e : Block)        => true
  (e.prev : Block)   => true
  (e.prev : Inline)  => (e.prev.boxRelRight + e.boxWidth) >
                         e.parent.boxWidth

e.lineBottom : Int
  (e.isLineHead = true) => e.boxRelBottom
  (e.prev : Inline)     => max(e.prev.lineBottom, e.boxRelBottom)
  (e)                   => e.boxRelBottom

e.boxRelLeft : Int
  (e : Block)              => 0
  (e.isLineHead = true)    => 0
  (e)                      => e.prev.boxRelRight

e.boxRelRight : Int
  (e) => e.boxRelLeft + e.boxWidth

e.boxRelTop : Int
  (e is first)           => 0
  (e.isLineHead = true)  => e.prev.lineBottom
  (e)                    => e.prev.boxRelTop

e.boxRelBottom : Int
  (e) => e.boxRelTop + e.boxHeight

e.boxWidth : Int
  (e : Block)  => ICB.width
  (e : Inline) => e.width

e.boxHeight : Int
  (e.height = auto) => max(c.boxRelBottom for c in e.children)
  (e)               => e.height

e.boxAbsLeft : Int
  (e is root) => ICB.left + e.boxRelLeft
  (e)         => e.parent.boxAbsLeft + e.boxRelLeft

e.boxAbsTop : Int
  (e is root) => ICB.top + e.boxRelTop
  (e)         => e.parent.boxAbsTop + e.boxRelTop
```

4.4 Stage 4: The box model

Stage 4 introduces the box model, which decorates every visible element with a border, interior padding and an exterior margin.

4.4.1 The existing specification

The box model is described in Section 8 of the CSS 2.1 spec [9]. Every element has a (possibly zero) margin, border and padding around its central 'content area'. Figure 4.6 demonstrates how the different areas described by the border, margin and padding fit together.

- "The content edge surrounds the rectangle given by the width and height of the box, which often depend on the element's rendered content. The four content edges define the box's content box." ([9] Section 8.1)
- "[For block-level elements in normal flow] The following constraints must hold among the used values of the other properties: 'margin-left' + 'border-left-width' + 'padding-left' + 'width' + 'padding-right' + 'border-right-width' + 'margin-right' = width of containing block" ([9] Section 10.3.3)
- "In CSS, the adjoining margins of two or more boxes (which might or might not be siblings) can combine to form a single margin. [...] Horizontal margins never collapse. [...] When two or more margins collapse, the resulting margin width is the maximum of the collapsing margins' widths." ([9] Section 8.3.1)

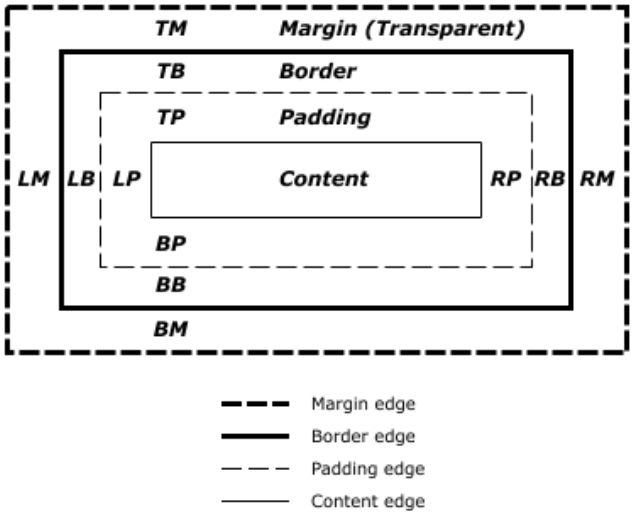


FIGURE 4.6: The CSS 2.1 box model ([9] Section 8)

4.4.2 In AG form

Figures 4.7 and 4.8 show an attribute grammar notation for both block-level and inline-level boxes in the same document, with all boxes conforming to the CSS box model, and vertical margins between two consecutive block-level boxes collapsing. This spec assumes that an element has the same margin for all edges (top, bottom, left, and right).

Now that elements potentially have 'thickness' to their edges, the amount of space a box takes up (`boxOuterWidth`) can be different to the space available to its children (`boxInnerWidth`). `mbp` is a helper attribute, which represents the sum of an element's `margin`, `border` and `padding`. In CSS 2.1, it is possible for margins to *collapse* (share the same space) under the right circumstances. Figure 4.7 implements the most simple example of margin collapsing; consecutive block-level boxes. This behaviour is present in the last rule for the `boxRelTop` attribute.

Figure 4.7: CSS stage 4 in attribute grammar form

```
e.isLineHead : Boolean
  (e is first)       => true
  (e : Block)        => true
  (e.prev : Block)   => true
  (e.prev : Inline)  => (e.prev.boxRelRight + e.boxOuterWidth) >
                         e.parent.boxInnerWidth

e.lineBottom : Int
  (e.isLineHead = true) => e.boxRelBottom
  (e.prev : Inline)     => max(e.prev.lineBottom, e.boxRelBottom)
  (e)                   => e.boxRelBottom

e.boxRelLeft : Int
  (e : Block)              => 0
  (e.isLineHead = true)    => 0
  (e)                      => e.prev.boxRelRight

e.boxRelRight : Int
  (e) => e.boxRelLeft + e.boxOuterWidth

e.boxRelTop : Int
  (e is first)           => 0
  (e.isLineHead = true)  => e.prev.lineBottom
  (e : Inline)           => e.prev.boxRelTop
  (e.prev : Inline)      => e.prev.lineBottom
  (e.prev : Block)       => e.prev.boxRelBottom -
                            min(e.margin, e.prev.margin)

e.boxRelBottom : Int
  (e) => e.boxRelTop + e.boxOuterHeight
```

FIGURE 4.8: CSS stage 4 in attribute grammar form (continued)

```
e.boxInnerWidth : Int
  (e : Inline) => e.width
  (e is root)  => ICB.width - 2*e.mbp
  (e)          => parent.boxInnerWidth - 2*e.mbp

e.boxInnerHeight : Int
  (e.height = auto) => max(c.boxRelBottom for c in e.children)
  (e)               => e.height

e.boxOuterWidth : Int
  (e) => e.boxInnerWidth + 2*e.mbp

e.boxInnerHeight : Int
  (e) => e.boxInnerHeight + 2*e.mbp

e.mbp : Int
  (e) => e.margin + e.border + e.padding

e.boxAbsLeft : Int
  (e is root) => ICB.left + e.boxRelLeft
  (e)         => e.parent.boxAbsLeft + e.parent.mbp + e.boxRelLeft

e.boxAbsTop : Int
  (e is root) => ICB.top + e.boxRelTop
  (e)         => e.parent.boxAbsTop + e.parent.mbp + e.boxRelTop


e.boxMarginLeft     : Int   (e) => e.boxAbsLeft
e.boxMarginRight    : Int   (e) => e.boxMarginLeft + e.boxOuterWidth
e.boxMarginTop      : Int   (e) => e.boxAbsTop
e.boxMarginBottom   : Int   (e) => e.boxAbsTop     + e.boxOuterHeight

e.boxBorderLeft     : Int   (e) => e.boxMarginLeft   + e.margin
e.boxBorderRight    : Int   (e) => e.boxMarginRight  - e.margin
e.boxBorderTop      : Int   (e) => e.boxMarginTop    + e.margin
e.boxBorderBottom   : Int   (e) => e.boxMarginBottom - e.margin

e.boxPaddingLeft    : Int   (e) => e.boxBorderLeft   + e.border
e.boxPaddingRight   : Int   (e) => e.boxBorderRight  - e.border
e.boxPaddingTop     : Int   (e) => e.boxBorderTop    + e.border
e.boxPaddingBottom  : Int   (e) => e.boxBorderBottom - e.border

e.boxContentLeft    : Int   (e) => e.boxPaddingLeft   + e.padding
e.boxContentRight   : Int   (e) => e.boxPaddingRight  - e.padding
e.boxContentTop     : Int   (e) => e.boxPaddingTop    + e.padding
e.boxContentBottom  : Int   (e) => e.boxPaddingBottom - e.padding
```

4.5 Stage 5: Floats

Stage 5 introduces floating elements, which escape the normal flow of layout to sit on the far left or right edge of the containing box. Floats break many of the rules of 'normal' element layout, and might appear to be ill-suited to an attribute grammar formalisation. We have chosen to examine floats for precisely this reason.

Meyerovich *et al.* [11] describe floats as "the most complicated and powerful elements in CSS2.1. [...] Floats are ambiguously defined and inconsistently implemented between browsers; our specification of them took significant effort and only supports left (not right) floats". Meyerovich's solution to the float problem involves 'speculative evaluation' and reevaluation of attributes, which is an unusual addition to an attribute grammar evaluator, and not possible for demand-driven evaluators. This approach stretches the definition of 'attribute grammar', and is not likely to be compatible with existing attribute grammar evaluators.

We chose to implement right floats only (not left), but achieved this using the same attribute grammar techniques as in previous stages. Note also that stage 5 does not contain box model properties as found in stage 4, for the sake of clarity. See the Figures in Section 5.1.5 for visual examples of floating elements.

4.5.1 The existing specification

Floats are described in Sections 9 and 10 of the CSS 2.1 spec [9].

- "A float is a box that is shifted to the left or right on the current line. The most interesting characteristic of a float (or 'floated' or 'floating' box) is that content may flow along its side." ([9] Section 9.5)
- "A floating box must be placed as high as possible. A left-floating box must be put as far to the left as possible, a right-floating box as far to the right as possible. A higher position is preferred over one that is further to the left/right." ([9] Section 9.5.1)
- "Since a float is not in the flow, non-positioned block boxes created before and after the float box flow vertically as if the float did not exist. However, the current and subsequent line boxes created next to the float are shortened as necessary to make room for the margin box of the float." ([9] Section 9.5)

Note that a float is vertically positioned "on the current line". This implies that a float should first be placed like an inline element, and then shifted to the left or right. However, an inline element's vertical position is determined by which line box contains it, and line boxes are 'shortened' by nearby floating elements. Therefore, the vertical position of a floating element depends upon the layout of surrounding line boxes, and the layout of those line boxes is directly affected by vertical position of said floating elements.

This dependency cycle is the reason floats are considered so complex. The CSS 2.1 seeks to be deterministic by describing an algorithm for placing floating elements. Some excerpts from this algorithm:

- "A floated box is shifted to the left or right until its outer edge touches the containing block edge or the outer edge of another float." ([9] Section 9.5)
- "If there is not enough horizontal room for the float, it is shifted downward until either it fits or there are no more floats present." ([9] Section 9.5)
- "A line box is next to a float when there exists a vertical position that satisfies all of these four conditions: (a) at or below the top of the line box, (b) at or above the bottom of the line box, (c) below the top margin edge of the float, and (d) above the bottom margin edge of the float." ([9] Section 9.5)
- "If a shortened line box is too small to contain any content, then the line box is shifted downward (and its width recomputed) until either some content fits or there are no more floats present." ([9] Section 9.5)

The final position of a float is not so much *calculated* as it is *found*. The methods shown above involve "moving" an element until a set of conditions are met. What's more, line boxes are also described as "moving" in this system, to allow floats to be placed. To have an element truly move around the page requires that the positional attributes of an element change over time, which means that an attribute's evaluation is not idempotent, which implies side-effects.

However, we believe that the same *results* can be achieved using standard attribute grammars. While a fully fledged implementation of floating elements is outside the scope of this thesis, Section 4.5.2 describes our contribution; a right-float CSS layout system in attribute grammar form without 'speculative evaluation'.

Figure 4.9: CSS stage 5 in attribute grammar form

```
e.floatRight : Boolean
  (e.float = right) => true
  (e)               => false

e.isLineHead : Boolean
  (e is first)                => true
  (e : Block)                 => false
  (e if e.floatRight = true) => false
  (e.prevInFlow : Block)      => true
  (e.prevInFlow : Inline)     => (e.prevInFlow.boxRelRight + e.boxWidth) >
                                 e.inlineRightMax

e.prevInFlow : Element
  (e is first)                => null
  (e.prev.floatRight = true) => e.prev.prevInFlow
  (e)                         => e

e.lineBottom : Int
  (e : Block)              => e.boxBottom
  (e is first)             => e.boxBottom
  (e.isLineHead = true)    => e.boxBottom
  (e.prevInFlow : Inline) => max(e.prevInFlow.lineBottom, e.boxBottom)
  (e)                      => e.boxBottom

e.prelim_boxRelLeft : Int
  (e : Block)            => 0
  (e.prevInFlow = null) => 0
  (e)                    => e.prevInFlow.boxRelRight

e.boxRelLeft : Int
  (e.floatRight = true) => e.floatRight_boxRelLeft
  (e.isLineHead)        => 0
  (e)                   => e.prelim_boxRelLeft

e.floatRight_boxRelLeft : Int
  (e.belowPrevRightFloat = true) => ICB.width - e.boxWidth
  (e.prevRightFloat = null)      => ICB.width - e.boxWidth
  (e)                            => e.prevRightFloat.boxRelLeft -
                                    e.boxWidth

e.inlineRightMax : Int
  (e is root)               => ICB.width
  (e is first)              => e.parent.boxWidth
  (e.prevRightFloat = null) => e.parent.boxWidth
  (e)    => if (e.prevRightFloat.boxRelBottom > e.preLineBreak_boxRelTop)
            then (e.prevRightFloat.boxRelLeft)
            else (e.prevRightFloat.inlineRightMax)

e.belowPrevRightFloat : Boolean
  (e.floatRight = false)           => false
  (e.prevRightFloatToRight = null) => false
  (e)                              => e.prelim_boxRelLeft + e.width >
                                      e.prevRightFloatToRight.boxRelLeft
```

FIGURE 4.10: CSS stage 5 in attribute grammar form (continued)

```
e.prevRightFloatToRight : Element
  (e is first)                => null
  (e.prev.floatRight = true) => if (e.preFloat_boxRelTop <
                                      e.prev.boxRelBottom)
                                then (e.prev)
                                else (null)
  (e)                         => e.prev.prevRightFloatToRight

e.prevRightFloat : Element
  (e is first)                => null
  (e.prev.floatright = true) => e.prev
  (e)                         => e.prev.prevRightFloat

e.preLineBreak_boxRelTop : Int
  (e is first) => 0
  (e : Block)  => e.prevInFlow.lineBottom
  (e : Inline) => e.prevInFlow.boxRelTop

e.preFloat_boxRelTop : Int
  (e is first)           => 0
  (e.isLineHead = true) => e.prevInFlow.lineBottom
  (e)                    => e.preLineBreak_boxRelTop

e.boxRelTop : Int
  (e.belowPrevRightFloat = true) => e.prevRightFloat.boxRelBottom
  (e.prevInFlow : Block)         => e.prevInFlow.boxRelBottom
  (e.prevInFlow : Inline)        => if (e.prevInFlow.boxRelRight +
                                    e.boxWidth > e.prevInFlow.inlineRightMax)
                                    then (e.prevInFlow.lineBottom)
                                    else (e.preFloat_boxRelTop)

e.boxRelRight : Int
  (e) => e.boxRelLeft + e.boxWidth

e.boxRelBottom : Int
  (e) => e.boxRelTop + e.boxHeight

e.boxWidth : Int
  (e : Block)  => ICB.width
  (e : Inline) => e.width

e.boxHeight : Int
  (e.height = auto) => max(c.boxRelBottom for c in e.children)
  (e)                => e.height

e.boxAbsLeft : Int
  (e is root) => ICB.left + e.boxRelLeft
  (e)         => e.parent.boxAbsLeft + e.boxRelLeft

e.boxAbsTop : Int
  (e is root) => ICB.top + e.boxRelTop
  (e)         => e.parent.boxAbsTop + e.boxRelTop
```

4.5.2 In AG form

Figures 4.9 and 4.10 show an attribute grammar notation for stage 3 extended with floating boxes. This grammar employs reference attributes, whose values resolve to a reference to another element in the document (or `null`). Since floating elements are not 'in the flow', the elements directly before and after a float should interact as if the float did not exist. `prevInFlow` is a reference to the most recent non-floating sibling, or `null` if none exist. `prevRightFloat` and `prevRightFloatToRight` are also reference attributes, with `prevRightFloat` finding the most recent sibling of an element that is a right float, and `prevRightFloatToRight` finding the most recent right floating sibling that occupies the same vertical space as the given element.

To avoid dependency cycles, `boxRelTop` was broken into three steps: `preLineBreak_boxRelTop` is the vertical position of the box assuming that the element is in the same 'line box' as its previous sibling. `preFloat_boxRelTop` is the vertical position of the box assuming that there are no floating boxes influencing its position (possibly making use of `preLineBreak_boxRelTop`). `boxRelTop` is the final vertical position of the box (possibly making use of `preFloat_boxRelTop`). This distinction was important because a box's vertical position depends on its `inlineRightMax`. `inlineRightMax` defines the amount of horizontal space available in the container at a given point, which can be affected by floating elements that lie to the right of a given element. To determine which floats lie to the right of an element, their vertical position is required. Without breaking the vertical position into steps, a dependency cycle would exist between `boxRelTop` and `inlineRight`.

5 Evaluation

5.1 The attribute grammar specification is executable

A number of attribute grammar evaluators exist that could be used to implement an attribute grammar based layout engine. We chose to use Kiama, a Scala based language processing library which supports dynamic attribute grammars [38]. Using Kiama tools, we implemented a simple HTML parser, which will create a document tree from an HTML string. The Java `Graphics2D` library was employed for 2D painting.

FIGURE 5.1: The method used to render an element in Scala

```
def render(node:Doc): Unit = {
  g2d.setStroke(new BasicStroke())
  g2d.drawRect(
      node->absx
    , node->absy
    , node->width
    , node->height
    )

  // render children (recursively)
  node.chren.map(render)
}
```

Figure 5.1 shows the `render` method, which uses the four basic positional attributes (`absx`, `absy`, `width` and `height`) to paint elements on a `Graphics2D` object. This method traverses (depth first) through the document tree, requesting positional information from each node as it goes. As `absx` and `absy` are requested for each node, the attribute grammar evaluator

examines attributes of the given element and related elements to find a final value. The four positional attributes are the entry points to the dependency graph, and trigger the evaluation of all other attributes.

5.1.1 A comparison of attribute grammar notations

In this thesis, primarily in Chapter 4, we have been using the attribute grammar notation described in Section 1.3.3. To execute these grammars, they must first be translated into Kiama attribute grammar notation. Our notation is quite similar Kiama's, so the translation is straightforward. We will examine each of the attributes from CSS stage 1's attribute grammar (shown in Figure 5.2) and compare them with the Kiama forms of the same attributes (shown in Figure 5.3).

`boxX` vs `absx`: Our `boxX` uses `ICB.left`, because a block-level element will (in stage 1) always horizontally span from the left of the initial containing block to the right. In the Kiama implementation, the display window is the initial containing block, so `absx` is always zero.

`boxWidth` vs `width`: Similarly to the previous attribute, the width in the Kiama grammar is the width of the display window, given by the external variable `windowWidth`.

`boxRelY` vs `rely`: It is clear to the naked eye how these two rules are related; just examine them in Figures 5.2 and 5.3. In the first rule, both check if the element is first among siblings on the left hand side of the rule, and return zero on the right. In the second rule, each have a notation for 'default' on the left, and contain semantically identical expressions on the right. Kiama uses the notation `n.prev[Doc]->rely` where our notation uses `e.prev.boxRelY`.

`boxY` vs `absy`: As with the previous attribute, the two notations express the same thing in similar notations. Note that `ICB.top` has been replaced with the constant zero, as the initial containing block is the display window.

`boxHeight` vs `height`: This attribute has the most significant differences. The Kiama implementation uses `Option` values to represent the difference between 'auto' and a numeric value. Pattern matching is used to extract the numeric value from the `Option`, with `None` representing 'auto'. Other than this, the notations are similar. `n.props.height` is the 'height' value specified by a CSS stylesheet, and is returned unless a value of 'auto' is found. In the 'auto' case, the maximum of all children's heights is returned. This is achieved in Kiama by mapping an anonymous function over the list of children (`n.chren`).

Figure 5.2: CSS stage 1 in attribute grammar form

```
e.boxX : Int
  (e) => ICB.left

e.boxWidth : Int
  (e) => ICB.width

e.boxRelY : Int
  (e is first) => 0
  (e)          => e.prev.boxRelY + e.prev.boxHeight

e.boxY : Int
  (e is root) => ICB.top + e.boxRelY
  (e)         => e.parent.boxY + e.boxRelY

e.boxHeight : Int
  (e.height = auto) => sum(c.boxHeight for c in e.children)
  (e)               => e.height
```

Figure 5.3: CSS stage 1 in Scala (Kiama) attribute grammars

```
  val absx: Doc => Double =
  attr {
    case _ => 0
  }

  val width: Doc => Double =
  attr {
    case _ => windowWidth
  }

  val rely: Doc => Double =
  attr {
    case n if n.isFirst => 0
    case n              => (n.prev[Doc]->rely) + (n.prev[Doc]->height)
  }

  val absy: Doc => Double =
  attr {
    case n if n.isRoot => 0 + n->rely
    case n             => (n.parent[Doc]->absy) + (n->rely)
  }

  val height: Doc => Double =
  attr {
    case n => n.props.height match {
      case Some(hgt) => hgt
      case None      => n.chren.map(c=>c->height).sum
    }
  }
```

5.1.2 Stage 1 demonstrated

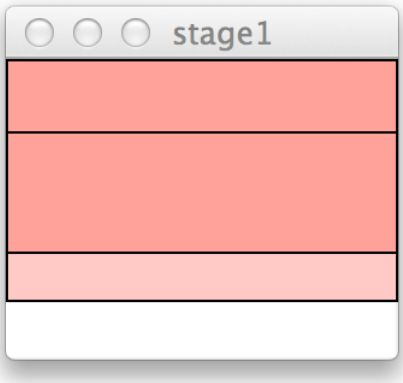
Figure 5.3 shows the attribute grammar from CSS stage 1 (Section 4.1) encoded in Kiama. Figure 5.4 shows a very simple HTML document consisting of nested `div` elements. Note that a `div` is a block-level element.

Figure 5.5 shows the result when Figure 5.4 is rendered. Block-level boxes are filled with a transparent red colour to demonstrate the layering that occurs; a darker red means that multiple block-level elements are layered. The 'outer' `div` is 100 pixels high, while the two inner `div`s are stacked vertically, with a cumulative height of 80 pixels. The black border shown does not indicate a space-consuming border (as described by the CSS box model), but is included only as a visual indicator of the size and position of elements.

FIGURE 5.4: CSS stage 1 example HTML

```
<div height=100>
    <div height=30></div>
    <div height=50></div>
</div>
```

FIGURE 5.5: CSS stage 1 rendered using Kiama attribute grammars



5.1.3 Stage 3 demonstrated

Stage 2 is not explored in detail due to its close similarity to stage 3. Stage 3 (covered in Section 4.3) introduces inline-level boxes, which stack horizontally and wrap. Figure 5.6 shows a simple HTML document with a number of `span`s (`span`s are inline-level elements) as well as `div`s. All `span`s have their width and height set manually, although in a fuller CSS implementation they would likely have their dimensions defined by the text within. We have a prototype implementation of this which demonstrates that attribute grammars pose no particular difficulty for allowing text to determine width and height.

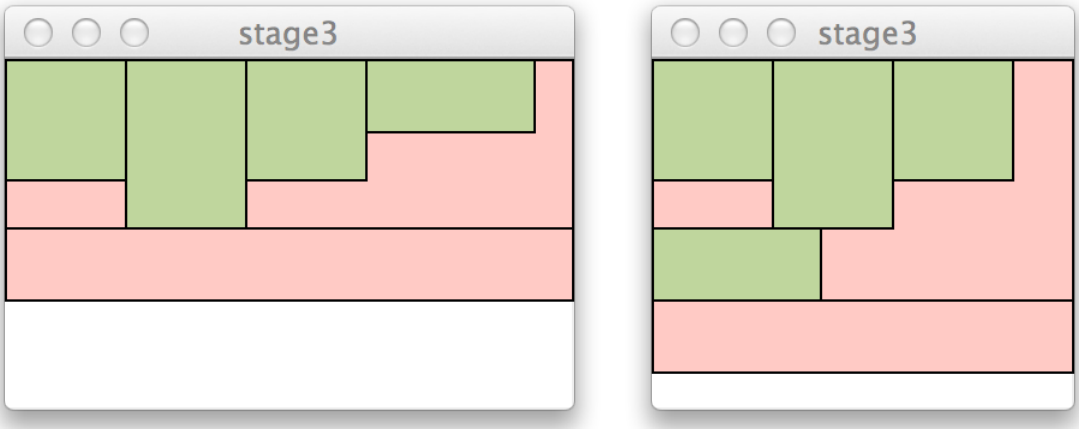
Figure 5.6: CSS stage 3 example HTML

```
<div>
    <span width=50, height=50></span>
    <span width=50, height=70></span>
    <span width=50, height=50></span>
    <span width=70, height=30></span>
</div>
<div height=30></div>
```

Figure 5.7 shows the output when Figure 5.6's document is rendered using Kiama attribute grammars. The first image shows the layout when the display window is wide enough to hold all four (green) inline boxes. The second image shows line-wrapping in action; when the window is not wide enough to fit all four inline elements in one line. Note that the fourth ('wrapped') inline box is not directly below the preceding element (as in stage 2), but has been placed below the anonymous 'line box' that contains all the elements in the first line.

Note that the HTML shown in Figure 5.6 is not strictly valid HTML. The first `span` shown specifies the CSS `width` and `height` properties to `50`. The real HTML notation would be `<span style="width:50;height:50;"><span>`. Our parser accepts a slightly different notation.

Figure 5.7: CSS stage 3 rendered using attribute grammars in Kiama

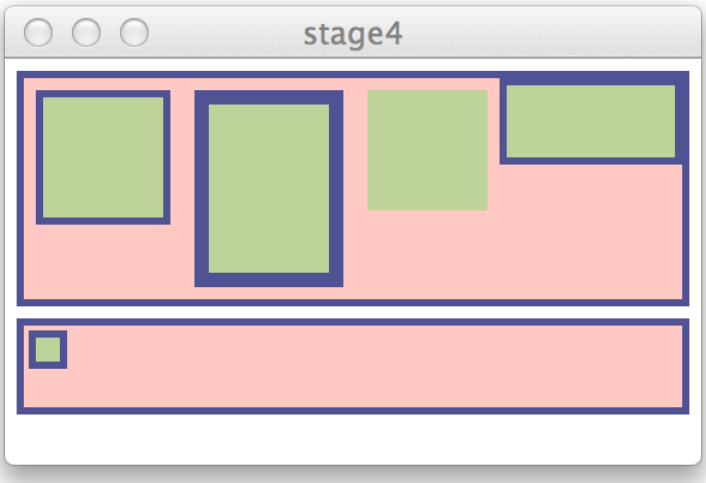


5.1.4 Stage 4 demonstrated

Stage 4 (covered in Section 4.4) implements the CSS box model. All elements now have a `border`, `margin`, and `padding`. An example HTML document is shown in Figure 5.9, and the rendered output is shown in Figure 5.8 . All `margin`, `border`, and `padding` values are 5, 3, and zero respectively, with some exceptions.

- The second `span` has a border of 6 pixels (shown in blue). Note that this causes the box to take up more space. The `outerWidth` of this box is affected by the `border` attribute. `outerWidth = width+margins+padding+borders = 50+2*(5+6+0) = 72`.
- The third `span` has no border (a `border` of zero width). This changes the appearance of the element as well as reducing the space it occupies.
- Note that the first three `span`s are surrounded by pink. This is because their `margin`s prevent them from touching the borders of their parent `div`. The fourth `span`, however, has a `margin` of zero, causing its border to sit directly alongside the parent's border.
- The `span` inside the second (lower) `div` also has a no margin, but it is not 'touching' its parent border. This is because the parent element has a `padding` of 2 pixels. Padding is spacing on the *inside* of the border, whereas margin is the spacing on the *outside* of the border.

FIGURE 5.8: CSS stage 4 rendered using attribute grammars in Kiama



Consider the space between the two (red) `div`s in Figure 5.8. Both `div`s have a 5 pixel margin, but there is only 5 pixels separating the two. There should only be one margin between the `div`s and the left edge of the window, but there should be two margins between the two `div`s (the top `div`'s margin and the bottom `div`'s margin). It is clear to the naked eye that the same amount of space is present to the left as is present between the two `div`s.

Figure 5.9: CSS stage 4 example HTML

```
<div margin=5, border=3, padding=0>
    <span width=50, height=50, margin=5, border=3, padding=0></span>
    <span width=50, height=70, margin=5, border=6, padding=0></span>
    <span width=50, height=50, margin=5, border=0, padding=0></span>
    <span width=70, height=30, margin=0, border=3, padding=0></span>
</div>
<div height=30, margin=5, border=3, padding=2>
    <span width=10, height=10, margin=0, border=3, padding=0></span>
</div>
```

This is because the two margins have *collapsed*, creating a total separation of `max(marginA, marginB)`. Instead of being two 5 pixel margins (totalling 10px of separation), there is only one *collapsed* margin (totalling 5px of separation).

5.1.5 Stage 5 demonstrated

Stage 5 (covered in Section 4.5) extends stage 3 with right floats. An example HTML document is shown in Figure 5.10, and the rendered output is shown in Figure 5.11. Some elements have been given an `id` property, which is printed inside the element when rendered. This is merely to assist in visually identifying elements, and has no effect on the layout system.

The first image in Figure 5.11 shows an element that has floated to the right (element `F`). Although element `F` comes before element `C` in the document structure, it lies to the right because it is a floating element. The element directly preceding `C` is `F`, but `F` is not in normal flow. `C`'s previous *in flow* element is `B`, and as such `B` and `C` are adjacent. The value of `C`'s `prevInFlow` attribute will be a reference to `B`.

Figure 5.10: CSS stage 5 example HTML

```
<div>
    <span id=A, width=50, height=50></span>
    <span id=B, width=50, height=60></span>
    <span id=F, width=45, height=45, float=right></span>
    <span id=C, width=50, height=50></span>
</div>
<div height=30></div>
```

The second image in Figure 5.11 shows the layout of the same document when the window has been resized such that not all inline elements fit horizontally in one line. The presence of element `F` has reduced the horizontal space available to the inline elements, and has caused element `C` to wrap onto a new line.

FIGURE 5.11: CSS stage 5 rendered using attribute grammars in Kiama

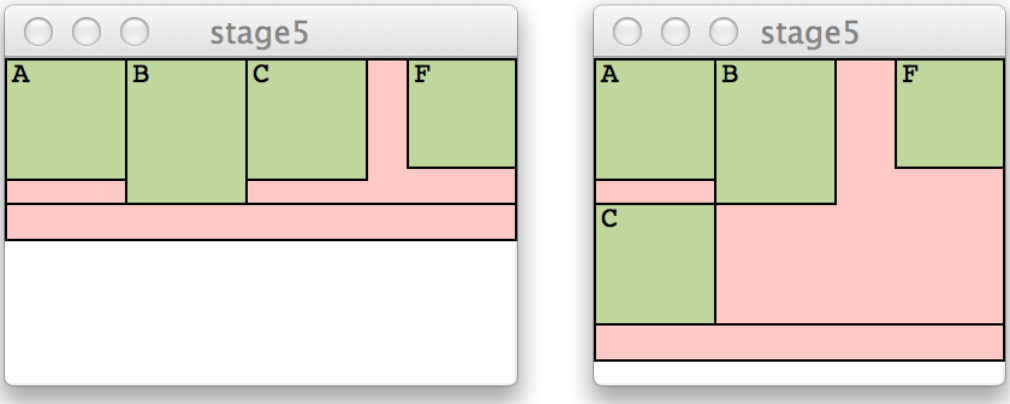
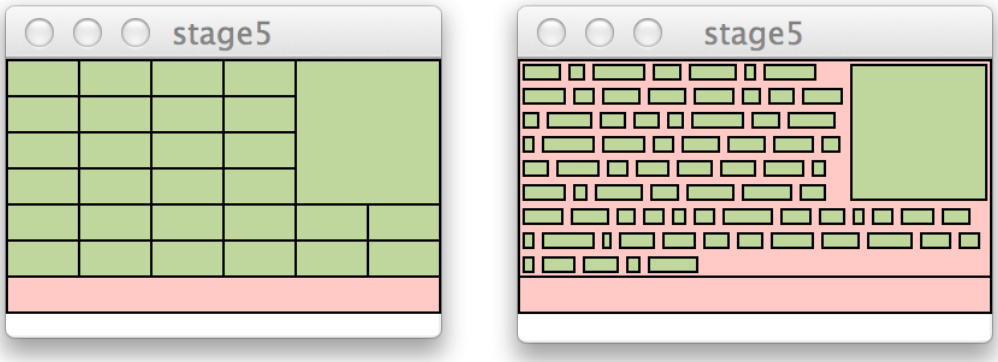


Figure 5.12 shows two examples of a realistic document taking advantage of floating elements. The left image shows a grid of similar elements, with a larger element on the right (which could be a picture for example). The right image shows a similar document, but with inline elements of random sizes to represent text[1], as in a newspaper article with an image accompanying a body of text.

FIGURE 5.12: CSS stage 5 rendered using attribute grammars in Kiama



[1]In this render, a margin of two pixels has been hardcoded into the layout scheme to give inline elements some separation, so that they look more like words of text.

5.2 The attribute grammar specification is human-readable

We have suggested that an attribute grammar specification is more human-readable than the existing prose specification. To a reader not familiar with attribute grammar notations, this is probably not the case. To a reader with no exposure to software development, the prose specification would certainly be easier to read than attribute grammar notation. The existing prose specification should continue to be the source of knowledge for the layman.

We assert that an attribute grammar specification is more readable *for the target audience*: browser developers. The CSS spec exists to show developers how CSS properties should be used to create a visual document. We argue that an attribute grammar gives a developer a clearer picture of the intended output than the prose specification. The attribute grammar has two properties that contribute to its developer-readability: determinism and clarity of expression.

Determinism The fact that an attribute grammar specification is executable is proof of its lack of ambiguity, i.e. its determinism. The grammar is made up of rules and formulae which can be followed resulting in a strictly-typed result. A prose description of a process can potentially be deterministic, but is prone to ambiguity. It is well-recognised that the current form of the specification is ambiguous [11]. A developer who seeks to understand a specification will have a requirement for determinism. The developer can not write software to match the specified output if the output is not concretely specified.

Clarity of Expression Determinism is only useful if it is well-understood. A layout engine written in assembly would be perfectly deterministic, but would not be a useful specification. The specification should relay to the developer an intended result, without its purpose being obfuscated by its notation.

Domain-specific languages are often created for the purpose of removing boilerplate code from an implementation, and attribute grammars are a domain-specific language designed for situations just like CSS layout: to evaluate attributes that decorate the nodes of a tree, whose values are dependent on the attributes of neighbouring nodes. Attribute grammars are a clear expression for the CSS layout problems, because they were designed specifically to be so.

Even with the evidence discussed, human-readability is a subjective quality. To argue for attribute grammars, let us examine an example that was shown in Section 3.2.3.

The `offsetTop` property, as it is described by the prose CSS specification:

> “ The `offsetTop` attribute must return the result of running these steps:
>
> 1. If the element is the HTML `body` element or does not have any associated CSS layout box return zero and terminate this algorithm.

2. If the `offsetParent` of the element is null return the y-coordinate of the top border edge of the first CSS layout box associated with the element, relative to the initial containing block origin, ignoring any transforms that apply to the element and its ancestors, and terminate this algorithm.
3. Return the result of subtracting the y-coordinate of the top padding edge of the first CSS layout box associated with the `offsetParent` of the element from the y-coordinate of the top border edge of the first CSS layout box associated with the element, relative to the initial containing block origin, ignoring any transforms that apply to the element and its ancestors. " [39]

The `offsetTop` property in attribute grammar notation:

```
elem.offsetTop : Int
  (e:Body)                  => 0
  (e.offsetParent = null)   => e.boxBorderTop -
                               ICB.top
  (e)                       => e.boxBorderTop -
                               e.offsetParent.boxPaddingTop - ICB.top
```

The prose spec describes `offsetTop` in three points, making up 150 words. The attribute grammar is a translation of those three points into three rules. A `Body` element yields a value of zero. An element with no `offsetParent` yields a value determined by a subtraction. Any other element yields the result of different expression, which is described by 61 words of prose (point three above) or the expression `e.boxBorderTop - e.offsetParent.boxPaddingTop - ICB.top` in the attribute grammar form.

Objectively, the attribute grammar form has no ambiguity. Subjectively, we the authors think that the attribute grammar form is easier to read, and easier to understand for the target audience (developers). A survey to see if the majority of web developers share this opinion is left to future work.

6 Conclusion

6.1 Future work

The most obvious extension of this work is to explore more of the CSS spec and write attribute grammar formalisations to match it. While a representative subset has been discussed in this document, a larger subset would provide further confirmation of the thesis.

Other than expansion, the most immediate task for future work would be to implement left and right floats together. Meyerovich *et al.* have implemented left floats 'attribute grammar notation', and we have implemented right floats. Unfortunately, the notation used by Meyerovich *et al.* is not compatible for translation into the notation we used, so left floats will need to be implemented manually. Further, the right float system demonstrated in this thesis only performs correctly for relatively simple documents. A new implementation needs to be implemented that conforms more rigorously to the spec.

The reason that left floats haven't been implemented is that they break the "ordered layout" process that exists currently. The reader may have noticed that all dependencies in existing attribute grammar implementations are up, down, or leftward. No element ever considers an element that comes *after* it in the document (excluding children). This means that elements can be placed on the screen one at a time, without being 'moved'. Left floats have the ability to shift their *previous* siblings to the right, whereas right floats only affect siblings that come after itself. It will be necessary to remove this constraint to properly implement left floats, however doing so increases the risk of dependency loops. When using attribute grammars for semantic analysis in a compiler, there are times when the *use* of an identifier must reference the *declaration* of the same identifier, for typechecking. One method used by these systems involves the parent element holding a sequence of references to identifier declarations, which can be accessed through the use of *parameterised attributes*.

This allows an identifier-use node to access the relevant identifier-declaration node without manually searching for it (as in `prevRightFloatToRight` in Section 4.5). It may be appropriate to employ parameterised attributes in a similar fashion to properly implement both left and right floats. It may be necessary for the parameterised attribute (of the parent) to consider all of its children upon evaluation, therefore causing an element to safely access elements that exist later in the document.

To implement floats in the way that is described by the spec, another alternative might be to add a new notation to attribute grammars that allows for attribute re-evaluation. This would need to be carefully constructed so as not to remove the properties that make attribute grammars to functionally attractive, such as being free of side-effects. Perhaps attributes can exist in a mutable state initially, and be made immutable when certain conditions are met, with overall evaluation not considered complete until all attributes were in an immutable state. Perhaps terminating recursion could be allowed (as it is not currently allowed in Kiama attribute grammars) to solve this problem.

Another shortcoming of the existing specification is that we assume that every element has one and only one CSS layout box. The spec formally specifies that a document element can be represented by zero or more boxes on-screen, whereas our implementation assumes zero or one boxes on-screen. This is fine for most situations, but a `<b>` element in HTML might contain text that exists on multiple lines in the visual document. There are cases where there are two or more layout boxes per document box, and this is not accounted for in the existing attribute grammar model.

Once a complete executable specification for CSS is completed, it will be possible to use this for automated testing of other layout engines. If the specification is executable, then there exists a 'perfect' layout engine. Even if attribute grammar layout is too slow for practical use in a browser, it will be possible to then automatically compare the results from optimised browsers to see that they strictly conform to the spec. We do not know yet if a complete browser engine built on attribute grammars will operate at a speed appropriate for direct use in browser implementations, and we do not necessarily expect that it will. Automated testing of browser engines will eliminate the need for CSS testing suites like the Acid Tests.

Outside of CSS layout and attribute grammars, there is related work to be done that explores existing relationships between functional (or declarative) languages and Web stack technologies. As has been mentioned earlier in this paper, there is a remarkable similarity between CSS selector matching and term-rewriting. This relationship is similar to the relationship between CSS layout and attribute grammars; an examination of CSS selector matching suggests that term-rewriting is the *natural* language to describe the specification.

6.2 The work presented

In this thesis, a new method for specifying and implementing a CSS layout engine was proposed, demonstrated, and discussed. We explored the CSS layout problem, outlining the features of the problem that make it difficult to specify and implement. We presented attribute grammars as a notation for specification and implementation. We introduced five subsets of the CSS layout problem, presented an attribute grammar specification for each of these stages, and demonstrated their executability by rendering some sample documents.

The key ideas of this thesis are:

- An attribute grammar based specification for CSS would be more human-readable than the existing (prose) specification, when the reader is a web or browser developer.
- Such a formalisation would be deterministic and executable.
- It would be feasible to create an attribute grammar formalisation for the whole CSS layout specification.

The three key ideas relate to human-readability, executability, and feasibility. Section 5.2 explored the readability of an attribute grammar specification, arguing that determinism is an important aspect of readability for developers. Section 5.1 showed determinism through execution, demonstrating the grammar's ability to correctly render some sample documents.

Chapter 4 developed a subset of the CSS layout problem in five stages, progressing in complexity. The last stage tackled what is arguably the most complex feature of CSS layout; floats (Section 4.5). This set of subproblems is designed to be representative of the larger layout problem, and extends upon the set of subproblems put forward by Meyerovich *et al.* [11], which the authors also claim to be representative. An attribute grammar specification and implementation of a *representative subset* of the CSS layout problem demonstrates that attribute grammars are feasible for application to the problem as a whole.

The key outcomes of this work have been:

- A demonstration that a significant subset of the CSS layout specification can be formalised using attribute grammars. (Chapter 4)
- A comparison of the existing CSS layout specification with our attribute grammar alternative. (Sections 3.2.3, 4, and 5.2)
- A demonstration of an executable attribute grammar formalisation. (Section 5.1)

References

[1] T. Berners-Lee. *HyperText Transfer Protocol*. Technical report, CERN (1992). URL http://www.w3.org/History/19921103-hypertext/hypertext/WWW/Protocols/HTTP.html.

[2] T. Berners-Lee. *HTML*. Technical report, CERN (1992). URL http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/MarkUp.html.

[3] I. Jacobs, A. L. Hors, and D. Raggett. *HTML 4.01 Specification*. W3C recommendation, W3C (1999). URL http://www.w3.org/TR/1999/REC-html401-19991224.

[4] *Standard ECMA-262* (ECMA International, 1999). URL http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[5] B. Lonsdorf. *Hey underscore, you're doing it wrong!* HTML5 Developers Conference, URL https://thenewcircle.com/s/post/1468/hey_underscore_you_are_doing_it_wrong_brian_lonsdorf.

[6] L. Wood, R. S. Sutor, S. Isaacson, C. Wilson, J. Robie, S. B. Byrne, M. Champion, I. Jacobs, G. Nicol, and A. L. Hors. *Document Object Model (DOM) Level 1*. W3C recommendation, W3C (1998). URL http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[7] J. Resig. *Jquery–javascript library* (2009). URL http://jquery.com/.

[8] H. W. Lie. *Cascading HTML style sheets: a proposal*. World Wide Web Consortium (W3C) (1994).

[9] H. W. Lie, T. Çelik, B. Bos, and I. Hickson. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. W3C recommendation, W3C (2011). URL http://www.w3.org/TR/2011/REC-CSS2-20110607.

[10] E. Etemad. *Cascading Style Sheets (CSS) Snapshot 2010*. W3C note, W3C (2011). URL http://www.w3.org/TR/2011/NOTE-css-2010-20110512/.

[11] L. A. Meyerovich and R. Bodik. *Fast and parallel webpage layout*. In *Proceedings of the 19th international conference on World wide web*, pp. 711–720 (ACM, 2010).

[12] D. E. Knuth. *Semantics of context-free languages*. Mathematical systems theory **2**(2), 127 (1968).

[13] H. Alblas and B. Melichar. *Proceedings on Attribute Grammars, Applications and Systems* (Springer-Verlag, 1991).

[14] K. Gorman. *Melchior*. `https://github.com/kjgorman/melchior` (2013).

[15] J. Shaw. *The Happstack Book: Modern, Type-Safe Web Development in Haskell*.

[16] M. Elder and J. Shaw. *Happstack-A Haskell Web Framework* (2011).

[17] M. Snoyman. *Developing Web Applications with Haskell and Yesod* (O'Reilly Media, Inc., 2012).

[18] M. Snoyman. *yesod: Creation of type-safe, RESTful web applications.* `https://hackage.haskell.org/package/yesod-1.2.6` (2014).

[19] C. Done and A. Bergmark. *fay: A compiler for Fay, a Haskell subset that compiles to JavaScript.* `https://hackage.haskell.org/package/fay-0.20.0.4` (2014).

[20] L. Stegeman. *GHCJS*. `https://github.com/ghcjs/ghcjs` (2014).

[21] A. Dijkstra. *The Utrecht Haskell Compiler (UHC).* `https://github.com/UU-ComputerScience/uhc` (2014).

[22] E. Czaplicki. *Elm: The Elm language module.* `http://hackage.haskell.org/package/Elm-0.12.3` (2014).

[23] O. Ataman, D. Beardsley, G. Collins, C. Howells, and C. Smith. *snap: Top-level package for the Snap Web Framework).* `http://hackage.haskell.org/package/snap-0.13.2.7` (2014).

[24] A. Farmer. *scotty: Haskell web framework inspired by Ruby's Sinatra, using WAI and Warp.* `http://hackage.haskell.org/package/scotty-0.8.0` (2014).

[25] A. G. Corona. *MFlow: stateful, RESTful web framework.* `http://hackage.haskell.org/package/MFlow-0.4.5.4` (2014).

[26] S. Visser. *salvia: Modular web application framework.* `http://hackage.haskell.org/package/salvia-1.0.0` (2010).

[27] J. Wang. *miku: A minimum web dev DSL in Haskell.* `http://hackage.haskell.org/package/miku-2014.5.19` (2014).

[28] J. Simeon, J. Robie, S. Boag, D. Florescu, M. Fernandez, and D. Chamberlin. *XQuery 1.0: An XML Query Language (Second Edition).* W3C recommendation, W3C (2010). URL `http://www.w3.org/TR/2010/REC-xquery-20101214/`.

[29] P. Wadler. *XQuery: a typed functional language for querying XML.* In *Advanced Functional Programming*, pp. 188–212 (Springer, 2003).

[30] S. Pemberton. *XHTML 1.0: The Extensible HyperText Markup Language*. W3C recommendation, W3C (2000). URL http://www.w3.org/TR/2000/REC-xhtml1-20000126.

[31] U. Kastens, W. M. Waite, and A. M. Sloane. *Generating Software from Specifications* (Jones & Bartlett Learning, 2007).

[32] U. Kastens. *LIDO-Computations in Trees*. University of Paderborn. Distributed with the Eli Compiler Construction System (1997).

[33] U. Kastens. *LIGA: A language independent generator for attribute grammars*. Tech. rep., Technical Report Bericht.

[34] S. D. Swierstra, P. R. Azero Alcocer, and J. Saraiva. *Designing and Implementing Combinator Languages*. pp. 150–206 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1999).

[35] R. Berjon, S. Faulkner, E. O'Connor, S. Pfeiffer, E. D. Navara, and T. Leithead. *HTML 5.1 - Rendering*. Candidate recommendation, W3C (2014). URL http://www.w3.org/TR/2014/WD-html51-20140617/rendering.html.

[36] E. D. Navara, R. Berjon, S. Pfeiffer, E. O'Connor, T. Leithead, and S. Faulkner. *HTML 5.1*. W3C working draft, W3C (2014). URL http://www.w3.org/TR/2014/WD-html51-20140617/.

[37] J. Meiert. *CSS Properties Index*. http://meiert.com/en/indices/css-properties/ (2014). (Visited on 09/08/2014).

[38] A. Sloane. *Attribution - kiama - Overview of Kiama's support for attribute grammars*. https://code.google.com/p/kiama/wiki/Attribution (2013). (Visited on 10/01/2014).

[39] G. Adams and S. Pieters. *CSSOM View Module*. W3C working draft, W3C (2013). URL http://www.w3.org/TR/2013/WD-cssom-view-20131217/.