

SCSS without the S: Automated refactorings for maintainability using modern CSS features

VLAD NEGARĂ, University of Twente, The Netherlands

Developers migrate away from CSS preprocessors, but find themselves lacking the pure CSS vocabulary to write idiomatic style sheets that are highly compact, readable, and maintainable. This paper closely examines three select features added to the language at different points in the last decade to assess the maintainability improvements they introduce. The main scientific contribution consists of identifying the refactoring opportunities offered by the chosen features, proposing novel algorithms to automatically apply them, and evaluating said algorithms in a small-scale study on real-world style sheets. The results confirm the high potential of the refactorings for improving maintainability, with nesting showing the most promise out of the three features.

Additional Key Words and Phrases: Cascading Style Sheets, maintainability, modernity, refactoring, static analysis

1 INTRODUCTION & BACKGROUND

Cascading Style Sheets (CSS) is the presentation language of the web, used to style documents whose structure is defined in the Hypertext Markup Language (HTML). At its simplest, a style sheet is made up of *rules*, where each rule consists of one or more *selectors* and a *declaration block*.

A selector delineates a set of elements within the HTML document to which the rule's declarations are to be applied. They range from simple selectors (e.g., `p`, which selects all paragraphs on a page) to highly specific, complex selectors (e.g., `#main-content article.recent > footer`, which selects footers that are direct children of articles with the class `recent` that are, in turn, descendants of the element with the unique identifier `main-content`). A rule may have multiple selectors, in which case it targets all elements belonging to at least one of the selected sets.

A rule's declaration block comprises one or more *declarations*: property–value pairs that each describe a certain aspect of an element's presentation. Typical examples include the manipulation of backgrounds, fonts, decorations, visual layout, and animations.

A web browser's rendering engine applies styles to a document according to the *specificity* of selectors. A selector's specificity is a 3-tuple of nonnegative integer values, computed from its constituent simple selectors—every ID selector adds (1, 0, 0) to the specificity, every class selector (0, 1, 0), and every type selector (0, 0, 1). If the same property is assigned conflicting values by rules that target the same element with selectors of differing specificity, the declaration with higher specificity overwrites the others. In the case where the specificities are equal, the last declaration in the source code is used.

TSelT 45, July 3, 2026, Enschede, The Netherlands

© 2026 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1.1 Industry context & Motivation

The straightforward declarative paradigm of CSS has historically been seen by many developers as restrictive and harmful to maintainability. Consequently, the desire to declare styles in a procedural manner has led to the creation of several preprocessor tools which augment the base language syntax in various ways.

The long-standing preprocessors Sass [6] and Less [28] each define a superset language of CSS with added constructs such as mixins and inheritance, which is transpiled to pure CSS by a dedicated compiler. PostCSS [29], a more recent project in the space, offers an extensible plugin-driven architecture that allows style sheet authors to select exactly which transformations should be applied to the code. Since the initial release of Sass in 2006, preprocessors have become an inextricable part of an overwhelming majority of developer workflows [13].

However, with web browsers delivering implementations of new CSS features at an unprecedented pace, a large number of web developers have begun critically weighing the benefits of preprocessors against their added complexity. Variables, nesting, and custom functions, among others, have gone from exclusive features of preprocessors to standardization in the CSS specification. Between 2022 and 2025, the adoption rates of Sass [6], Postcss [29], and Less [28] have been on a consistent downtrend, having diminished from 87.5%, 54.9%, and 23.8% to 61%, 38%, and 9%, respectively [10–13].

Complementarily, the practice of writing CSS without a preprocessor has seen an upward trend from 15% in 2023 to 21% in 2025 [11–13]. This is observed in the context of new, maintainability-oriented features in CSS that enable developers to better structure their code using native syntax that does not require transpilation.

1.2 Scope & Research questions

In light of the noticeable industry shift toward plain CSS, this paper aims to explore maintainability improvements in manually written CSS, i.e. CSS code not produced by a preprocessor. The specific focus is on the refactoring opportunities introduced by the `:is()` pseudo-class [14], the nesting module [1], and the `if()` function [2], as well as their effect on code maintainability. The goals of the study are formalized by the following research questions.

- RQ1. What are the refactoring opportunities offered by the `:is()` pseudo-class, the nesting syntax, and the `if()` function, and what is their effect on maintainability?
- RQ2. How can potential improvements be detected and applied algorithmically to an arbitrary style sheet?
- RQ3. What is the impact of the refactorings under study on real-world plain CSS files?

2 RELATED WORK

The practical utility of an automated CSS refactoring tool hinges on its ability to preserve the presentation of an arbitrary style sheet.

Existing literature on the topic can be organized into two broad categories by the approach taken to prevent regression.

On the one hand, CSS can be analyzed against one or more HTML documents, which provide the context necessary to mark unreachable selectors for removal. This endeavor is encumbered by the dynamic nature of many web pages, which calls for a more intricate solution than simply processing the raw HTML file. Mazinianian et al. [25] sampled several snapshots of the DOM while the page is in use, under the assumption that it achieves a satisfactorily high coverage of possible execution flows. Another study by Hague et al. [21] involved encoding the JavaScript code of web pages to a mathematical tree rewriting model to identify CSS selectors which are redundant in any reachable state of the DOM.

On the other hand, algorithms can be developed to identify refactoring opportunities in style sheets in the absence of a target document. A narrower scope of possible refactorings is counterbalanced by the preservation of style across any possible HTML document. Part of the work by Genevès et al. [18] is dedicated to this approach, and several minifier tools [20] exist with the aim of reducing the file size of CSS code while conserving semantics. As the three chosen CSS features can be applied effectively without a reference HTML document, this paper is only concerned with target document-agnostic transformations.

Finally, the body of literature on suitable metrics for the maintainability of CSS code is scarce. Keller and Nussbaumer [24] proposed the metric of abstractness for expressing the quality of CSS code, in response to a rise in machine-generated style sheets. However, the metric was devised based on contemporary CSS syntax and has not been adapted to current specifications.

In a broader software engineering context, Boehm et al. [3] characterized *understandability* and *modifiability* as constituent properties of maintainability, where understandability further necessitates *conciseness* and *legibility* among other factors.

3 ARCHITECTURE

The proposed architecture for harnessing the refactoring potential of the selected CSS features is described in this section. The section opens with an account of the mathematical model and notation used in the proposed pseudocode algorithms, followed by a subsection dedicated to each selected feature.

3.1 Mathematical model

In pseudocode fragments in this paper, a CSS document is represented as an abstract syntax tree (AST). Each proposed algorithm manipulates an AST subtree to apply the corresponding refactoring.

3.1.1 Basic selectors. A variety of selector nodes can be found in a typical CSS AST. The set of *basic selectors*, denoted by S_{basic} , can be expressed as the union of four disjoint sets $S_{universal} \cup S_{type} \cup S_{class} \cup S_{id}$. The set $S_{universal} = \{*\}$ contains only the *universal selector*. The sets S_{type} , S_{class} , and S_{id} contain all *type selectors*, *class selectors*, and *ID selectors*, respectively; for example, `article` $\in S_{type}$, `.alert` $\in S_{class}$, and `#account` $\in S_{id}$.

3.1.2 Simple selectors. The set of *simple selectors* is further defined as $S_{simple} = S_{basic} \cup S_{pseudo-element} \cup S_{pseudo-class} \cup S_{attribute}$.

The set $S_{pseudo-element}$ comprises all *pseudo-element selectors* (e.g. `:before`), while $S_{pseudo-class}$ is the set of all *pseudo-class selectors* (e.g. `:focus`). A pseudo-selector which admits child nodes is represented as an ordered pair of its identifier and its arguments. $S_{attribute}$ contains all *attribute selectors*.

3.1.3 Selectors and selector lists. The set S of all *selectors* can be defined as the language of valid CSS selectors over the alphabet $S_{simple} \cup C$. Here, $C = \{_, >, \sim, +, | \}$ is the set of *combinators*. The symbol L denotes the set of all *selector lists*, which are defined as all finite, nonempty sequences over S .

3.1.4 Declarations. CSS *declarations* are elements of the set $D \subset P \times V$, containing all valid property–value pairs. The sets P and V are assumed to contain all valid properties and values, respectively.

3.1.5 Rules, at-rules, declaration blocks. A simplified model of style sheets is assumed, in which a *document* node—the root of every style sheet AST—is a nonempty, finite sequence of *statements*.

The set of statements Σ is partitioned into the set of *rules* $\Sigma_{rule} = L \times \Delta$ and the set of *at-rules* $\Sigma_{at-rule} \subset A \times \Delta$. Here, A is the set of at-rule queries, partitioned into the sets A_{media} , $A_{supports}$, A_{layer} , A_{scope} , $A_{container}$, $A_{starting-style}$ representing the different CSS at-rules¹. Δ is the set of declaration blocks expressed as nonempty, finite sequences over $D \cup \Sigma$. Notably, declaration blocks of at-rules without a rule ancestor are not permitted to contain declarations.

3.1.6 Shorthands. Throughout the paper, several conventions and shorthands are employed as a means to simplify the notation of sequences, namely the elements of the sets S , L , and Δ .

Firstly, a sequence is denoted by a lowercase letter marked with an overline, while individual elements are referred to using subscript notation. Sequences are 0-indexed, and vertical bars indicate the length of a sequence. For instance, the sequence $\bar{\lambda} \in L$ comprises the elements $\lambda_0, \lambda_1, \dots, \lambda_{|\bar{\lambda}|-1}$.

Secondly, shorthands are employed as a means to simplify the notation of sequences. Elements of S are expressed by juxtaposing their members, while elements of L are denoted by comma-separated lists. A pair of empty parentheses $()$ stands for an empty sequence.

3.2 Refactoring using `:is()`

The `:is()` pseudo-class is used to target elements that match any selector out of a given selector list [14].

3.2.1 Opportunities for refactoring. The principal utility of the `:is()` pseudo-class is writing long, combinatorial lists of selectors in a more compact form.

Figure 1 showcases a small-scale example of a rule whose selector list can be refactored effectively using the `:is()` pseudo-class. The pre-refactoring code in Figure 1a targets nested lists located inside a navigation section, which requires four selectors to account for all different combinations of descendants. The practice is harmful from a maintainability standpoint, as any changes to be made in the common part of the selectors need to be manually applied to each selector in the list, creating an opportunity for error. The quality characteristic of modifiability [3] is thus impaired. Furthermore,

¹The existence of other at-rules is ignored for the purpose of the simplified model.

allowing the selection of, for instance, menu elements on both the inner and outer list levels would scale the list up to nine selectors.

<pre>nav ol ol, nav ol ul, nav ul ol, nav ul ul { margin: 0; }</pre>	<pre>nav :is(ol, ul) :is(ol, ul) { margin: 0; }</pre>
(a) CSS without :is()	(b) CSS with :is()

Fig. 1. Refactoring opportunity using :is()

An opportunity to refactor the long selector list arises from an observation about the selectors’ congruent structure—each can be loosely described as “nav, ol or ul, ol or ul”. It is precisely this highly concise [3] pattern that the :is() pseudo-class allows to be elegantly represented in a single selector, as shown in Figure 1b.

An important limitation of the pseudo-class stems from its specificity rules—an :is() selector takes on the highest specificity of a selector it contains. As a consequence, selector lists with differing specificities cannot be safely refactored. For example, the selector list #main section, main section being refactored to :is(#main, main) section would artificially increase the second selector’s specificity from (0, 0, 2) to (1, 0, 1), preventing a selector such as main article section from overriding it.

Additionally, the specification prohibits pseudo-elements from being used anywhere inside the selector list of :is() [14], which further restricts the set of allowed refactorings.

3.2.2 Proposed algorithm. The automated detection and application of the transformation previously discussed is made possible through Algorithm 1. The algorithm can be applied to each rule node of a style sheet AST and, in the case of a positive outcome, the selector list can be replaced by the resulting merged selector.

Algorithm 1 takes as input a selector list and repeatedly attempts to merge pairs of selectors; the process stops when either all selectors are merged into one, or no more improvements are possible. In the latter case, a null value is returned to signal the preservation of the selector list in its original form.

The fusion of two selectors is factored out into Algorithm 2 for increased clarity and to facilitate recursion. The function MERGESELECTORS, which receives as input two selector nodes, returns the combined selector if possible, or a null value otherwise.

The algorithm first rules out exceptional cases before computing p and s as the lengths of the longest common prefix and the longest common suffix, respectively. If either is nonzero, the merged selector is created by adjoining the common prefix, an :is() pseudo-class containing the two diverging segments as its list of selectors, and the common suffix of the two input selectors.

One particularly nuanced aspect of Algorithm 2 is the handling of selector pairs which first—or last—diverge at a combinator rather than a simple selector. For the selector list of the :is() pseudo-class to contain only valid selectors, the preceding—or succeeding—simple selector must be included by decrementing p or s , respectively.

Algorithm 1 Merge a selector list

Require: $\bar{\lambda} \in L$ is a selector list

```

1: function MERGELIST( $\bar{\lambda}$ )
2:    $changed \leftarrow True$ 
3:   while  $changed$  do
4:      $changed \leftarrow False$ 
5:      $i \leftarrow 0$ 
6:     while  $i < |\bar{\lambda}|$  do
7:        $j \leftarrow i + 1$ 
8:       while  $j < |\bar{\lambda}|$  do
9:          $s \leftarrow MERGESELECTORS(\lambda_i, \lambda_j)$ 
10:        if  $s \neq NULL$  then
11:           $\bar{\lambda} \leftarrow \lambda_0 \cdots \lambda_{i-1} s \lambda_{i+1} \cdots \lambda_{j-1} \lambda_{j+1} \cdots \lambda_{|\bar{\lambda}|-1}$ 
12:           $changed \leftarrow True$ 
13:        else
14:           $j \leftarrow j + 1$ 
15:        end if
16:      end while
17:       $i \leftarrow i + 1$ 
18:    end while
19:  end while
20:  if  $|\bar{\lambda}| = 1$  then
21:    return  $\lambda_0$ 
22:  else
23:    return  $NULL$ 
24:  end if
25: end function
```

Two modifications are further applied to ensure the highest effectiveness of the transformation. Firstly, the function MERGELIST is recursively called on the contents of :is(). This enables a selector list of the structure $x a y m z, x a y n z, x b y m z, x b y n z$ to be brought to the most compact form $x :is(a, b) y :is(m, n) z$ rather than $x :is(a y m, a y n, b y m, b y n) z$.

Secondly, the created :is() pseudo-class is processed to ensure the selector list `nav ol, nav ul, nav menu` is brought from `nav :is(:is(ol, ul), menu)` to the flat structure `nav :is(ol, ul, menu)`. The process of appending the contents of each nested :is() to the outer pseudo-class is laid out in Algorithm 3.

3.3 Refactoring using nesting

CSS nesting allows rules to be placed inside the declaration blocks of other rules [1]. The inner rule is defined with a partial selector which is interpreted as if it were prefixed by the outer selector. Widely available across modern browsers, nesting is frequently cited by developers as a compelling reason to abandon preprocessors [13].

3.3.1 Opportunities for refactoring. Figure 2 depicts two equivalent snippets of CSS that showcase the difference between “traditional” and nested syntax.

The simplest type of refactoring made available by the nesting syntax is easily discerned from the aforementioned example: for two rules which are declared consecutively, if the selector of the first rule is a strict prefix of the second rule’s selector, the second rule can be nested inside the first. This is achieved by replacing

Algorithm 2 Merge two selectors**Require:** $\bar{a} \in S$ is a selector**Require:** $\bar{b} \in S$ is a selector

```

1: function MERGESELECTORS( $\bar{a}, \bar{b}$ )
2:   if  $\bar{a}$  and  $\bar{b}$  have different specificities then
3:     return NULL
4:   end if
5:   if  $\bar{a}$  or  $\bar{b}$  contains a pseudo-element selector then
6:     return NULL
7:   end if
8:   if  $\bar{a} = \bar{b}$  then
9:     return  $\bar{a}$ 
10:  end if
11:   $p \leftarrow \min \{i \in \mathbb{Z}_{\geq 0} \mid a_i \neq b_i\}$ 
12:  if  $a_p$  or  $b_p$  is a combinator then
13:     $p \leftarrow p - 1$ 
14:  end if
15:   $s \leftarrow \min \{i \in \mathbb{Z}_{\geq 0} \mid a_{|\bar{a}|-1-i} \neq b_{|\bar{b}|-1-i}\}$ 
16:  if  $a_{|\bar{a}|-1-s}$  or  $b_{|\bar{b}|-1-s}$  is a combinator then
17:     $s \leftarrow s - 1$ 
18:  end if
19:  if  $p = 0$  and  $s = 0$  then
20:    return NULL
21:  end if
22:   $\bar{\lambda} \leftarrow a_p \cdots a_{|\bar{a}|-1-s} \cdot b_p \cdots b_{|\bar{b}|-1-s}$ 
23:   $\bar{s} \leftarrow \text{MERGELIST}(\bar{\lambda})$ 
24:  if  $\bar{s} = \text{NULL}$  then
25:     $\bar{s}' \leftarrow :is(\bar{\lambda})$ 
26:     $\bar{s} \leftarrow \text{FLATTEN}(\bar{s}')$ 
27:  end if
28:  return  $a_0 \cdots a_{p-1} \bar{s} a_{|\bar{a}|-s} \cdots a_{|\bar{a}|-1}$ 
29: end function

```

Algorithm 3 Flatten an $:is()$ pseudo-class selector**Require:** $s = :is(\bar{\lambda}) \in S_{pseudo-class}$ is an $:is()$ selector

```

1: function FLATTEN( $S$ )
2:    $\bar{\lambda}^* \leftarrow ()$ 
3:   for  $\bar{s} \in \bar{\lambda}$  do
4:     if  $|\bar{s}| = 1$  and  $s_0 = :is(\bar{\lambda}')$   $\in S_{pseudo-class}$  then
5:        $\bar{\lambda}^* \leftarrow \lambda_0^* \cdots \lambda_{|\bar{\lambda}^*|-1}^* \lambda_0' \cdots \lambda_{|\bar{\lambda}'|-1}'$ 
6:     else
7:        $\bar{\lambda}^* \leftarrow \lambda_0^* \cdots \lambda_{|\bar{\lambda}^*|-1}^* \bar{s}$ 
8:     end if
9:   end for
10:  return  $:is(\bar{\lambda}^*)$ 
11: end function

```

the occurrence of the first selector in the second selector with the nesting combinator $\&$, and assigning the second rule as the last child of the first. An analogous procedure can be applied in the converse case, when the second selector is a prefix of the first.

<pre> a { color: red; } a:hover { color: blue; } </pre> (a) CSS without nesting	<pre> a { color: red; &:hover { color: blue; } } </pre> (b) CSS with nesting
---	---

Fig. 2. Refactoring opportunity using nesting

Additionally, two consecutive rules with identical selectors can simply be collapsed into one, concatenating their declaration blocks.

Furthermore, the CSS nesting syntax is also defined for several at-rules such as `@media`. Figure 3 provides a representative example of a transformation made possible by the nesting of at-rules inside rules. The at-rule must be “shifted” inside the rule to allow the two selectors to be collapsed, which is only performed cleanly if the rule node is the single child of the at-rule. Like regular rules, at-rules with identical queries can be collapsed if they are consecutive.

<pre> body { background: #eee; } @media print { body { background: #fff; } } </pre> (a) CSS without nesting	<pre> body { background: #eee; } @media print { @media print { background: #fff; } } </pre> (b) CSS with nesting
---	--

Fig. 3. Refactoring opportunity using at-rule nesting

3.3.2 Proposed algorithm. The process of refactoring CSS by introducing nesting can be summarized in three algorithms.

Algorithm 4 introduces the `NESTRULES` function, which receives two rule nodes as an argument. If the rules have identical selectors, the function concatenates their declaration blocks. If one rule can be nested inside the other, it returns the resulting rule. In other cases, `NESTRULES` returns a null value.

The function `COLLAPSEATRULES` introduced in Algorithm 5 returns an at-rule whose declaration block is the concatenation of those of the at-rules it receives as arguments, if the two have identical queries; otherwise, it produces a null value. Algorithm 6 defines the function `SHIFTDOWN`, which interchanges an at-rule node and its only child if that child is a rule node.

The overarching algorithm for performing nesting-related refactorings consists of walking the statements in a style sheet through a breadth-first search. For each visited statement node σ , the following actions are attempted in sequence:

- If $\sigma \in \Sigma_{at-rule}$:
 - Call `COLLAPSEATRULES`(σ, σ^+) if the next sibling σ^+ of σ exists and $\sigma^+ \in \Sigma_{at-rule}$.

- Call $\text{COLLAPSEATRULES}(\sigma^-, \sigma)$ if the previous sibling σ^- of σ exists and $\sigma^- \in \Sigma_{at-rule}$.
- Call $\text{SHIFTDOWN}(\sigma)$.
- Otherwise, if $\sigma \in \Sigma_{rule}$:
 - Call $\text{NESTRULES}(\sigma, \sigma^+)$ if the next sibling σ^+ of σ exists and $\sigma^+ \in \Sigma_{rule}$.
 - Call $\text{NESTRULES}(\sigma^-, \sigma)$ if the previous sibling σ^- of σ exists and $\sigma^- \in \Sigma_{rule}$.

Algorithm 4 Nest two rules

Require: $a = (\bar{\lambda}^a, \bar{\delta}^a) \in \Sigma_{rule}$ is a rule

Require: $b = (\bar{\lambda}^b, \bar{\delta}^b) \in \Sigma_{rule}$ is a rule

```

1: function  $\text{NESTRULES}(a, b)$ 
2:   if  $\bar{\lambda}^a = \bar{\lambda}^b$  then
3:     return  $\left( \bar{\lambda}^a, \delta_0^a \dots \delta_{|\bar{\lambda}^a|-1}^a \delta_0^b \dots \delta_{|\bar{\lambda}^b|-1}^b \right)$ 
4:   end if
5:   if  $|\bar{\lambda}^a| \neq 1$  or  $|\bar{\lambda}^b| \neq 1$  then
6:     return NULL
7:   end if
8:    $\bar{s}^a \leftarrow \lambda_0^a$ 
9:    $\bar{s}^b \leftarrow \lambda_0^b$ 
10:   $p \leftarrow \min(|\bar{s}^a|, |\bar{s}^b|)$ 
11:  if  $\exists i \in \{0, \dots, p-1\} : s_i^a \neq s_i^b$  then
12:    return NULL
13:  end if
14:  if  $|\bar{s}^b| > p$  then
15:     $\sigma \leftarrow \left( \& \lambda_p^b \dots \lambda_{|\bar{\lambda}^b|-1}^b, \bar{\delta}^b \right)$ 
16:    return  $\left( \bar{\lambda}^a, \delta_0^a \dots \delta_{|\bar{\lambda}^a|-1}^a \sigma \right)$ 
17:  else
18:     $\sigma \leftarrow \left( \& \lambda_p^a \dots \lambda_{|\bar{\lambda}^a|-1}^a, \bar{\delta}^a \right)$ 
19:    return  $\left( \bar{\lambda}^b, \sigma \delta_0^b \dots \delta_{|\bar{\lambda}^b|-1}^b \right)$ 
20:  end if
21: end function

```

If a function call succeeds, σ is replaced in the tree by the returned node, the newly added node is added to the BFS queue to be checked for nesting again, and the rest of the actions in the sequence are canceled. If applicable, the neighbor node σ^+ or σ^- is removed.

Since rules with multiple selectors are explicitly not handled by the algorithm, its results are significantly improved by first subjecting the input document to the `:is()` refactoring procedure.

3.4 Refactoring using `if()`

The `if()` function is a recent, experimental feature of CSS allowing multiple values to be assigned to a property within a single declaration based on a list of conditions [2]. Although it appears promising,

Algorithm 5 Collapse two at-rules

Require: $\sigma^a = (a^a, \bar{\delta}^a) \in \Sigma_{at-rule}$ is an at-rule

Require: $\sigma^b = (a^b, \bar{\delta}^b) \in \Sigma_{at-rule}$ is an at-rule

```

1: function  $\text{COLLAPSEATRULES}(\sigma^a, \sigma^b)$ 
2:   if  $a^a = a^b$  then
3:     return  $\left( a^a, \delta_0^a \dots \delta_{|\bar{\delta}^a|-1}^a \delta_0^b \dots \delta_{|\bar{\delta}^b|-1}^b \right)$ 
4:   else
5:     return NULL
6:   end if
7: end function

```

Algorithm 6 Shift an at-rule down

Require: $\sigma = (a, \bar{\delta}) \in \Sigma_{at-rule}$ is an at-rule

```

1: function  $\text{SHIFTDOWN}(\sigma)$ 
2:   if  $|\bar{\delta}| \neq 1$  or  $\delta_0 \notin \Sigma_{rule}$  then
3:     return NULL
4:   end if
5:    $(\bar{\lambda}, \bar{\delta}') \leftarrow \delta_0$ 
6:   return  $(\bar{\lambda}, (a, \bar{\delta}'))$ 
7: end function

```

it is not widely available in modern web browsers. As a result, the rate of adoption of the feature by developers is exceedingly low, barely exceeding 2% as of 2025 [13].

3.4.1 Refactoring opportunities. Conditional logic in CSS is customarily applied by defining declarations for the same property in different rules. A common case in which this is necessary is when the presentation must adapt to the characteristics of the display medium, as in the code in Figure 3. Figure 4 depicts another valid rewriting of the same presentation logic, using the `if()` function.

```

body {
  background: if(
    media(print): #fff;
    else: #eee;
  );
}

```

Fig. 4. Refactoring opportunity using `if()`

The `if()` function allows the developer to specify one or more media or supports conditions (which allow queries identical to the corresponding at-rules) and a value for each, followed by a value for the default case using the `else` keyword. Similarly to `if()` constructs in most programming languages, conditions are evaluated in the order of declaration, and the value matching the first condition which evaluates to true is used.

The principal benefit of the `if()` function is the ability to consolidate all values defined for a property into a single declaration, which

provides legibility [3] by aggregating related code. An additional improvement stems from the order of the conditions: when multiple declarations within declaration blocks with identical specificity target the same property, the last definition overrides the previous ones; the `if()` function addresses this deviation from the general convention in programming languages of evaluating `if`-statement conditions from top to bottom.

However, excessive use of the `if()` function can have a negative impact on conciseness and modifiability [3]. Figure 5 illustrates an `if()`-transformation being applied to a rule concerning two properties. The transformed code in Figure 5b suffers from considerable duplication in the media queries, which scales in severity with the number of declarations transformed—should the width threshold be changed in the future, the modification would need to be manually applied to each instance of the query.

```
h1 {
  font-family: sans-serif;
  font-size: 20px;

  @media print and (width > 900px) {
    font-family: serif;
    font-size: 24px;
  }
}

(a) CSS without if()
```

```
h1 {
  font-family: if(
    media(print and (width > 900px): serif;
  );
  font-size: if(
    media(print and (width > 900px): 24px;
  );
  );
}

(b) CSS with if()
```

Fig. 5. Decreased maintainability after using `if()`

3.4.2 Proposed algorithm. Algorithm 7 formalizes the process of refactoring declarations to use the `if()` function. The function `ADDF` receives a declaration block $\bar{\delta}$ and a threshold n_{max} , returning the modified declaration block. The associative maps M_{index} and M_{conds} (accessed and modified through the functions `SET`, `GET`, and `HAS`) store the index in the declaration block where a given property was set, and an accumulated sequence of condition-value pairs, respectively. After all declarations and at-rules inside the declaration block are processed in order—with at-rules being excluded if they contain more than n_{max} declarations, or any non-declaration children—declarations whose property is overwritten by at least one at-rule have their value replaced by an instance of the `if()` function.

Algorithm 7 Add the `if()` function

Require: $\bar{\delta} \in \Delta$ is a declaration block

Require: Each declaration child of $\bar{\delta}$ targets a different property

Require: $n_{max} \in \mathbb{Z}_{>0}$

```
1: function ADDIF( $\bar{\delta}$ ,  $n_{max}$ )
2:    $M_{index} \leftarrow \emptyset$ 
3:    $M_{conds} \leftarrow \emptyset$ 
4:    $i \leftarrow 0$ 
5:   while  $i < |\bar{\delta}|$  do
6:     if  $\delta_i = (a, \bar{\delta}') \in \Sigma_{at-rule}$  and  $a \in A_{media} \cup A_{supports}$  then
7:       if  $|\bar{\delta}'| > n_{max}$  or  $\exists \delta' \in \bar{\delta}' : \delta' \notin \Delta$  then
8:         continue
9:       end if
10:      if  $\exists (p, v) \in \bar{\delta}' : \neg HAS(M_{conds}, p)$  then
11:        continue
12:      end if
13:      for  $(p, v) \in \bar{\delta}'$  do
14:         $\bar{c} \leftarrow GET(M, p)$ 
15:        SET( $M_{conds}, p, ((a, v), c_0, \dots, c_{|\bar{c}|-1})$ )
16:      end for
17:       $\bar{\delta} \leftarrow \delta_0 \dots \delta_{i-1} \delta_{i+1} \dots \delta_{|\bar{\delta}|-1}$ 
18:    else
19:      if  $\delta_i \in D$  then
20:         $(p, v) \leftarrow \delta_i$ 
21:        SET( $M_{index}, p, i$ )
22:        SET( $M_{conds}, p, ((else, v))$ )
23:      end if
24:       $i \leftarrow i + 1$ 
25:    end if
26:  end while
27:  for  $(p, \bar{c}) \in M_{conds}$  do
28:    if  $|\bar{c}| > 1$  then
29:       $i \leftarrow GET(M_{index}, p)$ 
30:       $\bar{\delta} \leftarrow \delta_0 \dots \delta_{i-1} (p, if(\bar{c})) \delta_{i+1} \dots \delta_{|\bar{\delta}|-1}$ 
31:    end if
32:  end for
33:  return  $\bar{\delta}$ 
34: end function
```

Since Algorithm 7 detects refactoring opportunities only for at-rules nested inside rules, its performance relies on being executed in sequence after the nesting algorithm described in Section 3.3.

4 PROTOTYPE & EVALUATION

In order to prove the feasibility of the proposed transformation algorithms, a TypeScript implementation was developed and evaluated in a small-scale experiment on pure CSS style sheets.

4.1 Parsing framework

Creating a prototype of the proposed CSS transformation architecture required basing the work on a well-established parsing framework. With the conversion of a CSS document to and from a parse

tree abstracted away, more time could be allocated to developing and evaluating a proof of concept for the novel transformation architecture. A number of aspects related to several prominent frameworks were considered before selecting PostCSS [29] as the environment to create the prototype in.

4.1.1 *rework*. A historically significant CSS parser framework is *rework* [22], which offered a plugin architecture for style sheet manipulation. Built atop the *css* package for Node.js [23], *rework* was a lightweight JavaScript framework that brought about a rich assortment of plugins.

After having been actively developed up until 2014, the project was officially deprecated in May 2026. This indicated a high likelihood that the library would fail to parse more recent CSS syntax, making it unsuitable for the purposes of the study.

4.1.2 *LightningCSS*. *LightningCSS* is an actively maintained “CSS parser, transformer, bundler, and minifier” [20] written in the Rust language. It is based on the *rust-cssparser* [27] and *selectors* [8] libraries developed as part of the Servo project [15].

Despite its high performance as a minifier, the architecture of *LightningCSS* is not expressly geared towards plugin development. This makes implementing the prototype using it a needlessly challenging task, while using the underlying libraries is unwieldy. This factor resulted in *LightningCSS* being ruled out as a fitting CSS parser.

4.1.3 *PostCSS*. Used by 38% of developers as of 2025 [13], *PostCSS* is a JavaScript tool for transforming CSS code [29]. *PostCSS* encapsulates conversion from and to source code, exposing a tree manipulation API for plugin development. While *PostCSS* itself parses the broader structure of a style sheet, more fine-grained parsing for particular constructs of the language is made available by other modules such as *postcss-selector-parser* [5] and *postcss-value-parser* [7].

PostCSS has been chosen for the development of a functional prototype as part of the study. Its active maintenance and comprehensive API make it fit for rapid iteration while interfacing with modern CSS code.

4.2 Implementation

The choice of *PostCSS* as a parser framework imposed the condition of using JavaScript as the language of implementation for the functional prototype. The TypeScript compiler and related elaboration tools were employed in an effort to ensure the type safety of the code at compile-time.

Functional prototypes of the `:is()`², `nesting`³, and `if()`⁴ algorithms previously discussed are implemented as three separate *PostCSS* plugins and made publicly available under the MIT license on GitHub. The isolation between the three programs allows developers and researchers to import only the portions of the AST transformation pipeline that are relevant to their work, and makes it possible to intersperse other plugins between them.

²<https://github.com/VladNegara/postcss-add-is>

³<https://github.com/VladNegara/postcss-add-nesting>

⁴<https://github.com/VladNegara/postcss-add-if>

Each of the three repositories is provisioned with an extensive test suite devised to cover a large number of expected edge cases.

4.3 Experiment

The evaluation of the proposed algorithms consists of a small-scale experiment through which their applicability to real-life style sheets was assessed.

4.3.1 *Dataset*. Given the ubiquity of style sheets generated by pre-processors in projects of all scales, a fair appraisal of the proposed solution within the previously delineated research scope necessitated careful curation of an authentic, pure CSS dataset.

The dataset was formed through a manual selection process guided by strict exclusion criteria. GitHub repositories containing a minimum of one `.css` file were inspected for compliance with the criteria in Table 1, in descending order of star count. Additionally, pertinent CSS frameworks [30] were included in the dataset.

The repository shows clear indications of preprocessor usage, e.g. due to containing `.sass`, `.scss`, or `.less` files.

The repository contains proof-of-concept CSS illustrations rather than style sheets with practical applicability.

Table 1. Repository exclusion criteria

The CSS files in the selected repositories were further filtered by the exclusion criteria in Table 2. The selection process was carried out until the resulting collection reached satisfactory dimensions.

The file contains syntax disallowed by the CSS specification.

The file serves as a constituent part of another style sheet in the repository.⁵

The file comprises too few CSS statements for meaningful transformations to be applied.

Table 2. File exclusion criteria

4.3.2 *Experimental methodology*. The experiment involved applying the three transformation algorithms on each document in the data set, in the sequence `:is()`, `nesting`, `if()`. For the `if()` plugin, the threshold n_{max} was configured to the value 1. A manual comparison between the input and output files was performed to quantify the identified refactorings.

4.4 Results

Table 3 presents the results of the experiment. For each style sheet in the dataset, the table records the number of transformations each of the three proposed algorithms identified and performed in the file. Entries are sorted in descending order of the number of lines in the original file.

Almost every analyzed file was found to contain a multitude of refactoring opportunities related to the three features being studied.

Firstly, most of the style sheets had a moderate amount of selectors that could be written in a more compact form using the

⁵In cases where the main file used `@import`, the imported contents were inlined.

Style sheet	Lines	:is()	nesting	if()
github-dark.user.css [17]	12181	202	582	1
beer.css [31]	5969	14	173	1
chota.css [19]	1000	2	22	0
mvp.css [4]	609	12	34	2
general.css [26]	528	3	16	0
normalize.css [16]	349	0	1	0
style.css [9]	222	1	11	0

Table 3. Refactoring counts per algorithm

:is() pseudo-class. This was even true of style sheets that already made use of :is(), namely github-dark.user.css [17] and beer.css [31], indicating that developers do not strive to maximize brevity in their selector lists.

Secondly, the experimental results definitively place CSS nesting as the feature with the most refactoring potential out of the three involved in the study. Styling elements targeted by a specific selector alongside their descendants, siblings, or pseudo-elements is a very common pattern in the selected style sheets.

Thirdly, opportunities for the introduction of the if() value function were detected very infrequently, indicating its role as a new paradigm of CSS rather than a drop-in replacement for a frequent syntax pattern.

The combined potential of the algorithms was revealed by a few instances where one plugin built on top of the previous, being able to apply a transformation that would not have been applicable otherwise. Figure 6 presents an excerpt of the beer.css style sheet [31] before and after application of the :is() and nesting transformations in sequence. As a result of compressing the list of two selectors into a single one, the nesting transformation was invoked on the rule to place it inside its predecessor.

5 CONCLUSION

The study has pursued the proposed research questions about the nature and efficacy of automated CSS refactorings, and produced thorough answers to each. The problem space of RQ1 was comprehensively explored in Sections 3.2.1, 3.3.1, and 3.4.1. The refactoring opportunities were subsequently capitalized upon by the algorithms detailed in Sections 3.2.2, 3.3.2, and 3.4.2 as an answer to RQ2. Finally, the experimental results in Section 4.4 show that many opportunities for maintainability-oriented refactorings can be identified in popular, open source, pure CSS style sheets. In response to RQ3, the observed results prove the feasibility of automated refactorings involving the :is() pseudo-class, nesting, and the if() value function for tangibly improving conciseness, legibility, and modifiability.

The experimental results summarized in this thesis additionally act as highly applicable data that can inform the World Wide Web Consortium CSS Working Group of trends in CSS feature usage, and guide decision-making with regard to experimental features.

Ultimately, the present thesis constitutes an instrumental step in bridging the gap between the preprocessors of the past and the CSS of the future. However, it is one of many advances needed to bring the language to its fullest imaginable potential.

```
progress.circle {
  /* Properties omitted for brevity */
}

/* 5 rules omitted for brevity */

progress.circle:not([value]),
progress.circle.indeterminate {
  /* Properties omitted for brevity */
}
```

(a) CSS before refactoring

```
progress.circle {
  /* Properties omitted for brevity */

  /* 5 nested rules omitted for brevity */

  &:is(:not([value]), .indeterminate) {
    /* Properties omitted for brevity */
  }
}
```

(b) CSS after refactoring

Fig. 6. Refactoring identified in beer.css [31]

5.1 Future work

Several conceivable improvements to the nesting algorithm described in Section 3.3.2 have been deemed to exceed the scope of the project. Firstly, the algorithm only collapses rules with identical selectors (up to whitespace), rather than reason about the equivalence of selectors which are equal up to the reordering of (pseudo-)class selectors within them. Secondly, rules that are collapsed could be further compressed by removing redundant declarations for the same property in the resulting declaration block. Thirdly, it is possible for an (at-)rule that comes between two others to be mutually exclusive with them, meaning they can be safely reordered without changing the presentation semantics; this is not accounted for by the algorithm. Finally, it is also possible to nest at-rules if one at-rule's query is a conjunction that contains the query of the other; this pattern of refactoring is not implemented due to the inadequacy of available parsing libraries in processing at-rule queries.

Each of the aforementioned points could constitute a starting-off point for future development and research, with a reasonable expectation of ever-improving maintainability.

Another direction that could be explored further is developing a framework of metrics to assess the maintainability of CSS documents, in order to compare measurements taken before and after the application of refactoring algorithms. Moreover, automated refactoring tools could be developed for additional CSS features that promote deduplication, such as custom functions.

REFERENCES

- [1] Tab Atkins-Bittner et al. 2026. *CSS Nesting Module Level 1*. <https://www.w3.org/TR/css-nesting-1/>

- [2] Tab Atkins-Bittner, Erika J. Etemad, Miriam E. Suzanne, et al. 2026. *CSS Values and Units Module Level 5*. <https://drafts.csswg.org/css-values-5>
- [3] B. W. Boehm, J. R. Brown, and M. Lipow. 1976. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) (ICSE '76). IEEE Computer Society Press, Washington, DC, USA, 592–605.
- [4] Andy Brewer et al. 2020. *MVP.css*. <https://github.com/andybrewer/mvp>
- [5] Ben Briggs, Chris Eppstein, Alexander Akait, et al. 2015. *postcss-selector-parser*. <https://github.com/postcss/postcss-selector-parser>
- [6] Hampton Catlin, Natalie Weizenbaum, et al. 2006. *Sass*. <https://sass-lang.com>
- [7] Bogdan Chadkin et al. 2015. *postcss-value-parser*. <https://github.com/postcss/postcss-value-parser>
- [8] Emilio Cobos Álvarez et al. 2012. *selectors*. Servo. <https://github.com/servo/style/tree/main/selectors>
- [9] cocktailpeanut et al. 2024. *Dalai*. <https://github.com/cocktailpeanut/dalai>
- [10] Devographics. 2022. State of CSS 2022. <https://2022.stateofcss.com/>
- [11] Devographics. 2023. State of CSS 2023. <https://2023.stateofcss.com/>
- [12] Devographics. 2024. State of CSS 2024. <https://2024.stateofcss.com/>
- [13] Devographics. 2025. State of CSS 2025. <https://2025.stateofcss.com/>
- [14] Erika J. Etemad, Tab Atkins-Bittner, et al. 2026. *Selectors Level 4*. <https://drafts.csswg.org/selectors/>
- [15] The Linux Foundation Europe. 2012. *Servo*. <https://servo.org>
- [16] Nicolas Gallagher et al. 2011. *normalize.css*. <https://github.com/necolas/normalize.css>
- [17] Rob Garrison et al. 2011. *GitHub-Dark*. <https://github.com/StylishThemes/Github-Dark>
- [18] Pierre Genevès, Nabil Layaida, and Vincent Quint. 2012. On the analysis of cascading style sheets. In *Proceedings of the 21st International Conference on World Wide Web* (Lyon, France) (WWW '12). Association for Computing Machinery, New York, NY, USA, 809–818. <https://doi.org/10.1145/2187836.2187946>
- [19] Jenil Gogari et al. 2017. *chota*. <https://github.com/jenil/chota>
- [20] Devon Govett et al. 2021. *LightningCSS*. <https://github.com/parcel-bundler/lightningcss>
- [21] Matthew Hague, Anthony W. Lin, and C.-H. Luke Ong. 2015. Detecting redundant CSS rules in HTML5 applications: a tree rewriting approach. *SIGPLAN Not.* 50, 10 (Oct. 2015), 1–19. <https://doi.org/10.1145/2858965.2814288>
- [22] TJ Holowaychuk et al. 2012. *rework*. <https://github.com/reworkcss/rework>
- [23] TJ Holowaychuk, Conrad Zimmerman, Simon Lydell, et al. 2021. *css*. <https://github.com/reworkcss/css>
- [24] Matthias Keller and Martin Nussbaumer. 2010. CSS Code Quality: A Metric for Abstractness; Or Why Humans Beat Machines in CSS Coding. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*. 116–121. <https://doi.org/10.1109/QUATIC.2010.25>
- [25] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. 2014. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) (FSE 2014). Association for Computing Machinery, New York, NY, USA, 496–506. <https://doi.org/10.1145/2635868.2635879>
- [26] Carlos Polop et al. 2024. *HackTricks*. <https://github.com/HackTricks-wiki/hacktricks>
- [27] Simon Sapin et al. 2013. *rust-cssparser*. Servo. <https://github.com/servo/rust-cssparser>
- [28] Alexis Sellier, Luke Page, Matthew Dean, et al. 2010. *Less*. <https://lesscss.org>
- [29] Andrey Sitnik et al. 2013. *PostCSS*. <https://postcss.org>
- [30] troxler et al. 2017. *Awesome CSS Frameworks*. <https://github.com/troxler/awesome-css-frameworks>
- [31] Leonardo Rafael Wehrmeister et al. 2020. *Beer CSS*. <https://github.com/beerccss/beerccss>