An empirical study on the use of CSS preprocessors

Davood Mazinanian, Nikolaos Tsantalis
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
Email: {d_mazina, tsantalis}@cse.concordia.ca

***Abstract*—Cascading Style Sheets (CSS) is the standard language for styling structured documents, such as HTML. However, CSS lacks most of the traditional programming constructs, including variables and functions, which enable code reuse and structured programming. Alternatively, CSS Preprocessors (e.g., LESS, SASS) have been introduced as superset languages to extend CSS by supporting those missing constructs. While these languages are being widely used by developers, we do not have sufficient knowledge about how developers take advantage of the features they provide. Gaining this knowledge is crucial for providing better tool support to the developer community by devising techniques for the automatic migration of existing CSS code to take advantage of CSS Preprocessor language features, designing refactoring recommendation systems for existing Preprocessor code, and giving insights to the Preprocessor language designers for improving language usability. In this paper, we have empirically investigated the CSS Preprocessor codebase of 150 websites regarding four preprocessor features, namely variables, nested selectors, *mixins* and *extend* constructs, and report the discovered usage patterns for each feature. We also discuss how the gained knowledge can be put into practice towards improving the development and maintenance of CSS preprocessor code.**

I. INTRODUCTION

Cascading Style Sheets (henceforth, CSS) is the standard language used for defining the look and feel of structured documents, for instance HTML and XML documents [1]. Surveys show that over 90% of the web developers use CSS in their everyday development tasks [2], and more than 90% of the websites are using CSS in their technology portfolio [3]. More recently, CSS started being adopted in the design of desktop applications (e.g., using WinJS), as well as mobile applications (e.g., using PhoneGap), extending its use in a wide spectrum of application domains.

CSS code is applied on some *target documents*, most usually, HTML documents. As it can be observed in Figure 1, CSS has a very simple syntax. Every CSS file (i.e., Style Sheet) contains a list of CSS rules in the form of one or more CSS *selectors*. A selector specifies which elements of the target documents should be styled (e.g., selector `p` selects all paragraphs in an HTML document). Inside the body of a selector there are one or more *style declarations*, which apply some *style values* (e.g., `red`) to some *style properties* (e.g., `color`) of the selected elements.

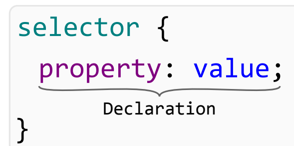

Fig. 1. CSS Syntax

The simplicity of the CSS syntax has historical roots. CSS was initially designed for web designers with limited programming experience [4]. Consequently, it lacks many of the fundamental programming constructs, such as variables, functions, loops and conditionals, which enable the reuse of code and structured programming. Therefore, maintaining CSS code can be a very difficult task.

A direct consequence of this lack of programming features is that CSS developers are prone to copying style declarations from one selector to another (i.e., code cloning). Although there is some limited built-in CSS support for minimizing duplication (such as *grouping* selectors sharing common declarations), there is still a considerable amount of duplicated code in the CSS code transferred to the end-users of websites. In a previous work, we examined the CSS code of 38 high traffic websites and we found that, on average, more than 60% of the CSS declarations were duplicated across at least two selectors [5].

CSS preprocessor languages were introduced by the industry as a response to the missing features of CSS. The code written in a CSS preprocessor can include variable and function declarations, which can be used inside CSS selectors. The preprocessor compiler essentially transforms (i.e., *transpiles*) the function calls and variable uses to pure CSS. Currently, there is a long list of CSS preprocessors offering very similar features with a different syntax (e.g., HSS [6], SASS [7], LESS [8], Google Closure StyleSheets [9]), and their use is becoming a fast growing trend in the industry. An online survey with more than 13,000 responses from web developers, conducted by a famous website focusing on CSS development, showed that around 54% of web developers use a CSS preprocessor in their development tasks [10]. United States Federal Government advises front-end web developers who design websites for government services to use SASS as their Style Sheet development language in order to get "resources such as frameworks, libraries, tutorials, and a comprehensive styleguide as support" [11].

While CSS preprocessors are popular among developers and they include several useful features, we do not have enough knowledge about how developers take advantage of these features in real web applications. Having such information can be useful for different reasons:

- A considerable number of web developers is still coding directly in pure CSS. Therefore, *migrating* existing CSS code to take advantage of preprocessor features (e.g., extracting duplicated declarations to a function in a CSS preprocessor) is greatly demanded in the industry. Knowing the practices applied by web developers when coding in preprocessors will certainly help in developing more useful and efficient migration strategies.
- CSS preprocessors might be sub-optimally used, because web developers miss opportunities to further eliminate existing duplicated code and other bad practices. Therefore, there is a need for *refactoring recommendation* systems to help developers in improving the quality of their CSS preprocessor code. Knowing developers' practices will help in prioritizing the refactoring opportunities leading to the most commonly used solutions/patterns.
- Finally, the knowledge of developers' practices can also guide the CSS preprocessor language designers to revisit the design of these languages, e.g., by adding support for new features (which are currently implemented by developers in an ad-hoc manner), or making existing features easier to use, or eliminating features that are not adopted by developers.

These reasons motivate us for conducting the *first empirical study on the use of CSS preprocessors*. We have analyzed the preprocessor code of 150 websites, having their CSS code written in LESS or SASS. We focused on these two preprocessors because, according to the results of an online survey [10], LESS and SASS are the most popular CSS preprocessors among web developers (92% of the developers who used a CSS preprocessor in their careers, preferred either LESS or SASS). Additionally, to achieve more generalizable results, we analyzed Style Sheets written using both of the two dialects that SASS provides: 1) The initial syntax of SASS, which is closer to Python (decreases development effort by removing braces and commas, and relying on the indentation to show code blocks and nesting); and 2) The so-called *SCSS* syntax, which is more similar to the syntax of pure CSS. We selected 50 websites for each of these two dialects (accumulating to 100 websites for SASS preprocessor), in addition to 50 websites for LESS.

In our analysis, we took into account the features of CSS preprocessors which are common in almost all preprocessors. These features include *variables*, *nesting*, *mixin* (i.e., function) calls and the *extend* construct.

Overall, this paper makes the following main contributions:

- We conduct the first empirical study on the use of CSS preprocessors and report our findings on 4 major preprocessor language features. We plan to use these insights to design refactoring/migration techniques for CSS and preprocessors.
- We make publicly available the dataset compiled from 150 websites to enable the validation and replication of our study, and facilitate future research on CSS preprocessors. We plan to use this dataset to evaluate the effectiveness and accuracy of our refactoring/migration techniques.

The rest of this paper is organized as follows: In Section II, we briefly introduce the reader to the features offered by CSS preprocessors. In Section III, we present the design of the empirical study that we conducted. In Section IV, we present the findings of our study and discuss the lessons learned.

II. CSS PREPROCESSOR FEATURES

In this section, we briefly demonstrate some of the common features of CSS preprocessors, which are widely used by developers. All code examples are given in LESS; the other CSS preprocessors use a similar syntax.

A. Variables

Supporting variables is one of the most basic features of traditional programming languages, which is missing in CSS. In preprocessors, variables can be defined to store one or more *style values*, for instance `@color: red` (i.e., a single-value variable), or `@margin: 1px 2px 4px 3px` (i.e., a multi-value variable). Variables can be used for various purposes, such as theming (i.e., one style sheet representing different themes/colors).

Preprocessor variables are type-less; a value representing a color value (e.g., `#FF00FF`) can be assigned to a variable which currently stores a dimension value (e.g., `2px`). Interestingly, some preprocessors also let developers to manipulate the value of variables by using arithmetic operators or by passing them to preprocessor built-in functions (e.g., making a color value darker using the `darken()` function in LESS). Preprocessors also support the notion of *variable scope*. A variable can be defined in the *global* scope (i.e., visible in the entire style sheet), or in some *local* scope (i.e., visible inside the body of a selector or *mixin*).

In Figure 2 (left), a piece of LESS code from the Semantic-UI[1] (version 1.6.2) is shown. The result of compiling this code is shown in Figure 2 (right).

LESS Code

```
...
@chAccordionMargin: 1em 0em 0em;
@chAccordionPadding: 0em;
...

...
.ui.accordion .accordion {
  margin: @chAccordionMargin;
  padding: @chAccordionPadding;
}
...
```

Generated CSS Code

```
.ui.accordion .accordion {
  margin: 1em 0em 0em;
  padding: 0em;
}
```

Fig. 2. Variables in LESS

B. Nested rules

Preprocessors support a feature called *nesting*, which generates selectors using the following constructs in pure CSS:

Combinators make an existing selector `B` more specific, with respect to another selector `A`. For instance, for selecting all elements selected by `B` which are descendants of the elements selected by `A`, we can use the *descendant combinator*, denoted as `A B` (with a space between the two selectors). Likewise, we can select all elements of `B` which are *direct*

[1] Semantic-UI is a CSS library used for building adaptive user interfaces for websites (https://github.com/Semantic-Org/Semantic-UI).

children of A using the *child combinator* (A>B), all elements of B which have an element of A as a sibling using the *general sibling combinator* (A~B), and all elements of B directly preceded by a sibling element of A using the *adjacent sibling combinator* (A+B).

Pseudo-Classes filter selectors. For instance, selector tr selects all table rows, and we can add the pseudo-class :hover to select all table rows when they are hovered by mouse (tr:hover).

Pseudo-Elements represent abstract elements with respect to real ones. For instance, p::first-line, selects the first line of all paragraph elements (p) in the target document.

As a real example of *nesting*, in Figure 3 (left), a code snippet from the Bootstrap CSS library[2] (version 3.3.1) is shown. The generated CSS code is shown in Figure 3 (right). As it can be observed, nesting avoids the repetition of .navbar-toggle selector and organizes relevant selectors in a hierarchical manner. The use of *nesting* leads to a more organized code by keeping relevant selectors in the same location. As we will see in Section IV, *nesting* is a very popular preprocessor feature.

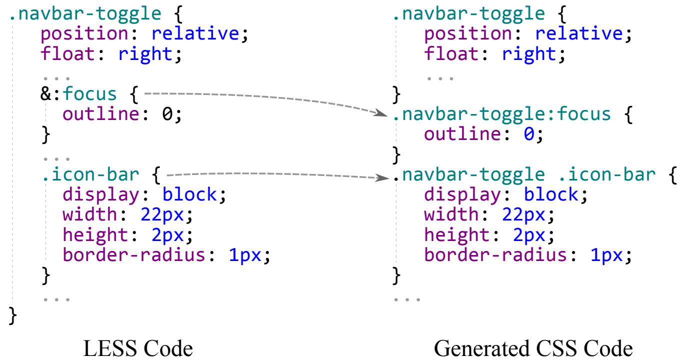

Fig. 3. Nesting selectors in LESS

C. Mixin calls

As it was mentioned earlier, pure CSS does not support the notion of functions. CSS preprocessors have introduced a specific construct, called *mixin*, to mimic the behavior of functions. A *mixin* can be defined as a set of declarations, and can be called inside other constructs (such as a selector or another *mixin*). The construct in which the *mixin* is called will include all the declarations of the called *mixin*. The declarations inside a *mixin* may have parameterizable values, therefore a *mixin* declaration can have parameters (just like a function in traditional languages). These parameters are preprocessor values, with the characteristics that were explained in Section II-A. Arguments can be omitted, if *default values* are provided in the parameter declarations of a *mixin*.

In Figure 4, a *mixin* is shown from the Bootstrap CSS library. As shown in Figure 4, after compiling this code, the declarations inside the *mixin* body appear in the selector .btn,

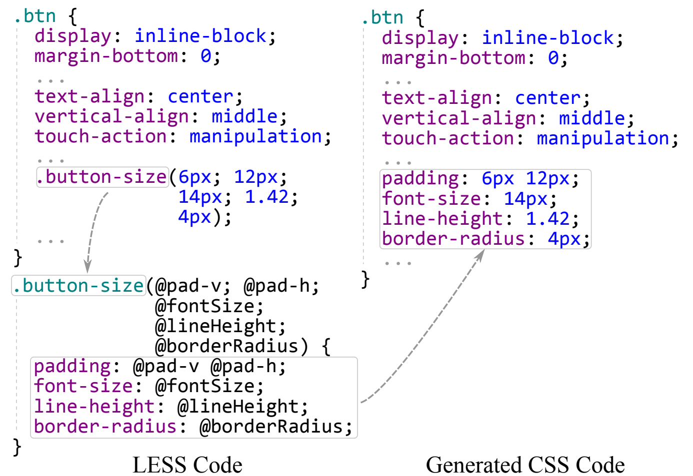

Fig. 4. Mixin in LESS

and the parameters are replaced with the arguments passed to the *mixin*.

D. Extend construct

As we mentioned, one of the major concerns in developing CSS is the inevitable duplication of declarations across different selectors. Pure CSS includes two main mechanisms to avoid this kind of duplication:

- ***Creating classes***. A set of declarations can be grouped in a *class selector*, associated with a class name. Such selector will select all the elements in the target document, which have the same class name in their class attribute. For instance, the selector .class1 can select the element <div class="class1"> in the target document.
- ***Grouping selectors***. A grouping selector consists of two or more basic selectors (separated by a comma), which share a set of declarations.

In CSS preprocessors, the *extend* construct is designed to play the same role towards avoiding duplication. The name of this construct was chosen to remind the extension feature of object-oriented programming languages like Java. In other words, the *extend* construct is used to "extend" the behavior of an existing selector by adding more style rules, while inheriting the existing style declarations from the extended selector. When using the *extend* construct, the common declarations are placed inside a grouping selector in the generated CSS code.

Figure 5 (left) demonstrates the use of the *extend* construct in a piece of code from the Flat-UI design framework[3]. The compiled CSS code is shown in Figure 5 (right). As it is observed, the selector which is extended (.dropdown-menu) and the extending selector (.select2-drop) are grouped to share some common declarations in the generated CSS code, while the extra declarations appear in a separate selector.

[2]The most famous CSS library which includes predefined classes for facilitating designing complex multi-column, responsive web pages, designed, used and maintained by Twitter (https://github.com/twbs/bootstrap)

[3]A design framework based on Bootstrap which includes a set of easy-to-use predefined UI elements (https://github.com/designmodo/Flat-UI)

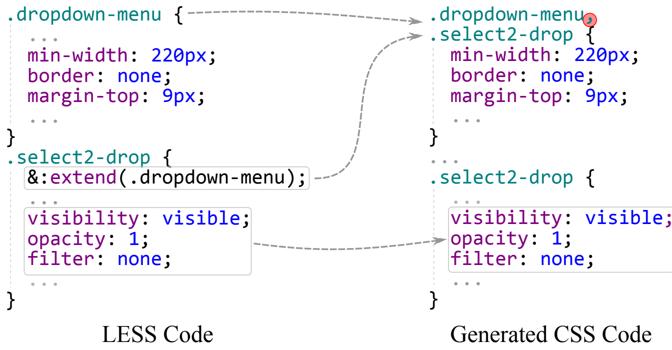

Fig. 5. Extending selectors in LESS

III. EXPERIMENT SETUP

In this section, we provide information about the subject systems used in the study, as well as the process we followed for collecting the experimental data.

A. Subject Systems

Within the context of this study, we focused on websites, which make use of preprocessor languages and have their preprocessor code publicly available. We have deliberately avoided the analysis of preprocessor libraries and frameworks (such as Bootstrap), because their code is meant to be used externally by other projects, in the same way that public APIs are used. Adding such libraries in our analysis would affect negatively the validity of this study, since a large number of *mixin* declarations developed to be used externally, would appear as not being used at all (i.e., unreachable or dead code). Therefore, we decided to focus on websites having their own internal preprocessor codebase, and study them in isolation from potential external dependencies.

While it is not necessary for websites to make their preprocessor codebase publicly available to the end users, some web developers intentionally upload the preprocessor code along with the generated CSS code on the web. This might be done to enable the compilation of the preprocessor code on demand (either on the server- or client-side). We used Google's advanced search feature to find these preprocessor files. Particularly, we searched the Internet for files with the extensions `*.less`, `*.scss` and `*.sass`. This search query allowed us to find websites satisfying our selection criteria:

1) the website should have its CSS code generated by LESS or SASS/SCSS, and
2) the website should publicly provide its preprocessor code along with the generated CSS code.

When we found a preprocessor file on a website, we manually attempted to extract the contents of the file's parent directory. If this directory was accessible, we collected all the preprocessor files inside it, and recursively all the preprocessor files inside its sub-folders. This was necessary to make sure that all the files which are imported using the `@import` directive are also collected. Then we manually found and marked the main Style Sheet files, i.e., the files that are passed to the preprocessor compiler to get the generated CSS files. This step is crucial, as we start our analysis from these main files and recursively parse and analyze the files which are imported from them.

More specifically, we collected 1266 preprocessor files, containing 255 LESS, 427 SASS, and 584 SCSS files. Due to space limitations, we do not include the full list of the website names and URLs in the paper, but we have made this list and the collected files available online[4]. In Figure 6, we have included box and bean plots of various size metrics for the collected files in logarithmic scale. Bean plots are useful for presenting the distribution of data. As it is observed, the examined LESS, SASS and SCSS files have similar size characteristics.

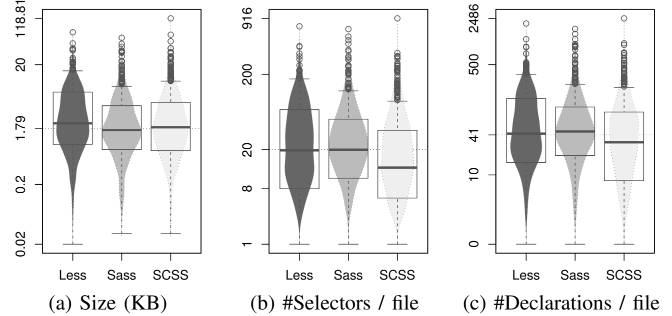

Fig. 6. Characteristics of the analyzed preproecssor files

B. Data Collection

After collecting the preprocessor files, we applied the following process. First, we parsed each preprocessor file to obtain its Abstract Syntax Tree (AST). For parsing, we used the corresponding compilers for LESS and SASS/SCSS. The LESS compiler is originally written in JAVASCRIPT, but we used a Java implementation of this compiler, called Less4j. For SASS/SCSS, we used the original compiler written in Ruby. In both cases, we developed additional code for querying the ASTs. The results of the queries were exported to CSV files for further statistical analysis. In Table I, we provide an overview of the collected data in the subject systems. We will refer to this table in Section IV and discuss the numbers in more detail.

For each examined preprocessor feature we create a separate CSV file. Every CSV file contains the website name, the preprocessor file name, and the line number in which a particular AST element (e.g., variable declaration, *mixin* declaration, *mixin* call) was found. According to the specific characteristics of the AST element type, we include the following additional information in the corresponding CSV file:

1) For variable declarations, we include
   a) the scope of the variable (global or local scope)
   b) the type of the value stored in the variable. This type can take one of these possible values: color, number, identifier, string, function call, and "other" for all other types of values, as discussed in Section IV-A
2) For *mixin* calls, we include
   a) the name of the called *mixin*
   b) the total number of arguments passed to the *mixin*

[4]http://bit.ly/1ZgarwZ

3) For *mixin* declarations, we include
   a) the name of the *mixin*
   b) the number of times the *mixin* is called
   c) the number of its parameters
   d) the number of declarations which directly or indirectly (i.e., using *nesting*) exist inside the body of the *mixin*
   e) the number of declarations in the body of the *mixin* which use at least one of the parameters of the *mixin*
   f) the number of declarations styling vendor-specific properties (e.g., `-webkit-column-gap` for Chrome and Safari, `-moz-column-gap` for Firefox)
   g) the number of distinct parameters which are used for two or more different property types (e.g., a parameter used for styling the `top` and `margin` properties)
   h) the number of declarations using only hard-coded (i.e., literal) values
   i) the number of vendor-specific property declarations which share at least one of the *mixin*'s parameters
4) For *nesting*, we include
   a) the name of the selector
   b) the number of *base* selectors it consists of (e.g., the *grouped* selector `H1, A > B` consists of two *base* selectors, namely `H1` and `A > B`)
   c) the number of *combinator* selectors in the list of its *base* selectors (the presence of a *combinator* selector indicates a missed *nesting* opportunity)
   d) the name of its parent selector
5) For each use of the *extend* construct, we include the target selector which is extended.

TABLE I
OVERVIEW OF THE COLLECTED DATA

	LESS	SASS	SCSS
# Websites	50	50	50
# Files	255	427	584
Avg. # selectors / file	57	52	40
Avg. # defined variables / file	16	14	16
Avg. # *nesting* usages / file[†]	43	44	35
Avg. # *mixin* calls / file	11	6	12
Avg. # *mixin* declarations / file	4.7	4.7	3.7
Avg. # *extend* construct usages / file	0	5.2	5
Avg. # calls to parameterless *mixins* / file	8	4	6

[†] Includes all selectors which were nested under another selector, or had at least one selector nested under them.

In order to count the number of times a *mixin* is called, we analyze the CSV file containing the *mixin* calls in order to extract the number of calls having the same name with that of the *mixin* declaration. If multiple *mixin* declarations have the same name (i.e., *mixins* with an identical name declared in different preprocessor files), then we count only the number of calls having the same name and belonging to the same file with that of the *mixin* declaration.

IV. EMPIRICAL STUDY

In this study, we investigate the use of the following preprocessor features: *variables*, *nesting*, *mixin calls* and *extend constructs*. Targeting the goals mentioned in Section I (developing better migration and refactoring recommendation systems and giving feedback to preprocessor language designers), we attempt to answer the following research questions:

RQ1 ***How do developers use variables in preprocessors?***
We aim at investigating whether developers have a particular preference to global or local scope variables, and the types of style values stored in the variables.

RQ2 ***Are developers using nesting whenever possible?***
We are going to investigate whether developers use *nesting* in every possible situation, or only when the benefits to maintainability are stronger (e.g., in deep hierarchies of elements).

RQ3 ***How and why do developers use mixins?***
For *mixins*, several dimensions will be investigated, namely:
a) Are *mixins* created to be reused in a style sheet?
b) Do *mixins* tend to have a large number of parameters?
c) Are *mixin* parameters reused in multiple style properties?
d) What is the nature of declarations inside the body of *mixins*? For instance, do developers use *mixins* for grouping a set of *related* declarations (e.g., declarations which style the same property for different web browsers)?

RQ4 ***Are developers using the extend construct whenever possible?***
Given the fact that an *extend* construct can be used in place of a parameterless *mixin* (because they are both used to remove duplication of declarations), we are going to investigate whether developers have a preference to use parameterless *mixins* over *extend* construct or vice versa.

In the following subsections, we answer the abovementioned research questions.

A. Variables

We investigate whether developers declare variables in the global scope (i.e., for the entire style sheet), or they mostly prefer local variables (e.g., inside a *mixin*). Gaining such knowledge can be beneficial in devising migration or refactoring techniques, because, as mentioned before, variables can be used to store one or more style values repeated across different selectors, and thus facilitate the maintainability of the code. Therefore, a migration (or refactoring) algorithm can detect such value-level duplications in pure CSS (or preprocessor code) and suggest the introduction of appropriate variables. Based on our empirical findings, we can align the refactoring recommendations with the practices which are more commonly applied by the developers, when there are multiple alternative Introduce-Variable refactoring opportunities in the local or global scope.

TABLE II
SCOPE OF VARIABLES

	Global (%)	Local (%)	Total
LESS	956 (95.79)	42 (4.21)	998
SASS	917 (84.67)	166 (15.33)	1,083
SCSS	1,387 (88.34)	183 (11.66)	1,570
Total	3,260 (89.29)	391 (10.71)	3,651

As shown in Table II, out of 3,651 total variable declarations in the dataset, there are 3,260 global variables (89.29% of the total variable declarations). On the other hand, only 10.71% of the variable declarations are in the local scope (note that we do not count *mixin* parameters as variable declarations). This clearly shows a preference of the developers to define variables in the global scope.

In addition, we are interested in understanding the types of the values stored in the variables. We categorized all possible value types that are allowed in preprocessors, as shown in Table III, and counted the instances of the variables belonging to each category.

TABLE III
CATEGORIZATION OF VALUE TYPES

Category	Value type	Example

Number	angle
	integer
	length
	number
	percentage
	resolution
	time
	number function
		`45deg`
		`-13`
		`18px`
		`4.01`
		`50%`
		`72dpi`
		`5ms`
		`floor()`

Color	named color
	hex color
	color function
		`red`
		`#FF00FF`
		`rgb(50, 0, 0)`

Identifier	user-defined CSS keyword	`nice-animation` `top`
String	Unicode string enclosed in " " or ' ' string function	`"Concordia"` `replace()`
Function call	excluding number, color, string functions	`svg-gradient()`
URL	resource path, using the `url()` function	`url()`
Expression	expression involving other variable(s)	`@opac1+0.2`
List	any of the above	`solid 1px red`

In Figure 7, we have demonstrated the percentage of variable instances in each value category, for each of the analyzed preprocessors. As it can be observed, most of the variable declarations are used for *color* values. This accounts for 45.98% of all variables defined in the three datasets. Values in this category consist of named and Hexadecimal colors, in addition to color functions, such as `rgb()` and `rgba()`. This observation shows that variables are mostly used for facilitating the modifications to the *theme* of web pages (i.e., same structural layout with different color themes).

As it can be observed in Figure 7, there is a considerable use of expressions for the initialization of preprocessor variables. These expressions are either direct references to previously defined variables, or mathematical expressions manipulating the values of existing variables (e.g., `@opac2: @opac1 + 0.2`). In this way, the preprocessor developers can easily modify existing themes and layouts.

It should be mentioned that, there was one file in the SASS dataset in which the developer used variables for *all* the declarations defined in the file. This practice is definitely uncommon in developing CSS preprocessor code, and it can negatively affect the results of this study by changing the number of times a certain value type is used. Thus, we excluded this single file from this specific analysis for counting variable types.

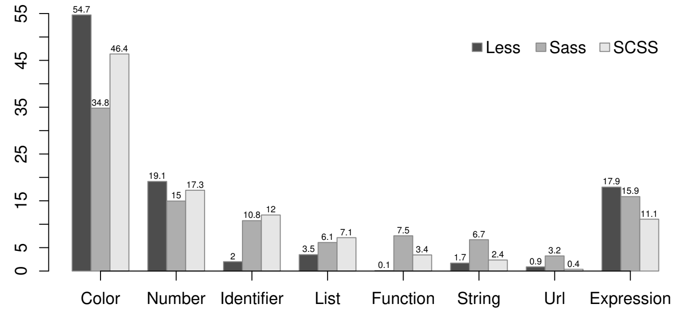

Fig. 7. Variable types distribution (numbers represent percentages)

RQ1 Conclusions: Developers mostly declare global variables (89.29% of the variable declarations have a global scope), and especially variables storing color values (45.98% of the variable declarations have a color value). Hence, any **migration/refactoring technique** should rank higher the suggestions that introduce variables for identical values across different selectors and *mixins*, leading to the introduction of global variables. Recommendations can also be prioritized based on the types of the involved values, giving higher priority to those involving color values.

B. Nesting

In this subsection, we examine how developers take advantage of *nesting* in preprocessors. Our investigation shows that *nesting* is a construct that is widely used by the developers. In Table IV, we present the collected data for *nesting* usage in the three subject preprocessors.

TABLE IV
USE OF NESTING

	LESS	SASS	SCSS	Total
# All Selectors	12,390	18,555	18,242	49,187
# Selectors involved in *nesting*	6,481	13,370	10,269	30,120
# Potential *nesting* opportunities	2,685	1,939	3,861	8,485
# All *nestable* Selectors	9,166	15,309	14,130	38,605

As it can be observed in this table, out of all 49,187 selectors, there were 38,605 selectors which were either already nested or could be potentially nested. Out of this number, there were 30,120 selectors (78.02%), which were actually involved in some nesting hierarchy, i.e., they had at least one selector nested under them, or were nested under another selector. On the other hand, in the whole dataset, there were 8,485 selectors which could be nested, but developers did not apply *nesting* for them. These selectors are basically *combinators*, *pseudo-classes*, and *pseudo-elements* (Section II-B), which can be refactored to take advantage of *nesting*.

To gain more knowledge about *nesting* practices, we also investigated the *nesting depth* in preprocessor files. We define the *nesting* depth of selector `s`, which is nested under selector `p`, as the depth of selector `p` plus one. The depth of a top-level selector (i.e., a selector which has no parent in the *nesting* hierarchy) is equal to zero.

Figure 8 demonstrates the box plots along with the violin plots (for exhibiting the distribution of values) for the *nesting* depth of selectors in the examined style sheets. As it can be observed, the median of the *nesting* depth is 2 in all three datasets (for the SCSS dataset, the third quartile is the same as the median, both equal to 2). This means that, in half of the cases, selectors are nested only one or two levels deep, which is a clear indicator that developers prefer to nest selectors even for very shallow *nesting* hierarchies.

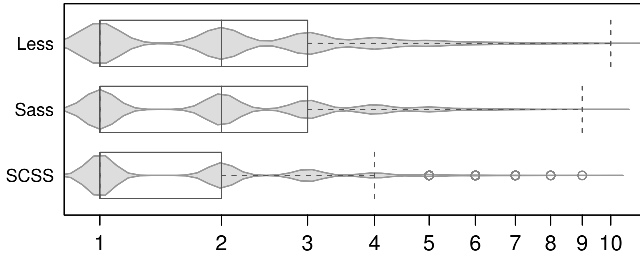

Fig. 8. Nesting depth

RQ2 Conclusions: *nesting* is a very popular preprocessor feature that is widely used by the developers (78.02% of the selectors are nested), even in very shallow *nesting* hierarchies consisting of one or two levels. Given this result, any **migration/refactoring technique** should support the recommendation of *nesting* refactoring opportunities, wherever it is possible.

C. Mixins

We examined the use of preprocessor *mixins*, taking into account four different dimensions.

1) Number of mixin calls: Our goal is to understand whether *mixins* are created to be reused (i.e., called by multiple selectors or other *mixins*), or whether they are created to decompose selectors by extracting a subset of relevant declarations from them (i.e., called by only one selector). In the former case, *mixins* are used to eliminate duplication of declarations in the CSS code.

For answering this question, first we counted the *mixin* calls for each *mixin* declaration. As shown in Figure 9, the median value for number of times each *mixin* is called is 2 for LESS and SASS, and 3 for SCSS. Overall, we found out that 63% of the *mixins* are called more than once.

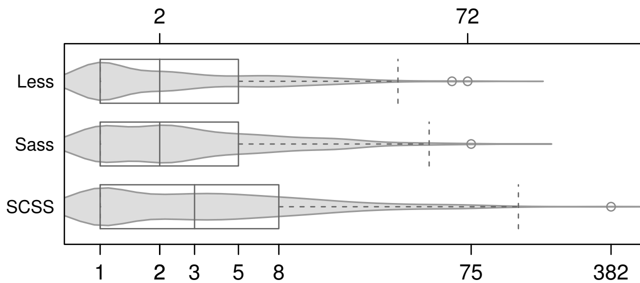

Fig. 9. Number of *mixin* calls

In addition, we applied the Wilcoxon signed-rank test on the paired samples of the numbers of *mixins* being called just once and of those being called more than once in each website with the following null hypothesis: "the number of *mixins* being called once is larger than the number of *mixins* being called more than once". The null hypothesis was rejected with significance at 95% confidence level (p-value = 0.00003), and thus we can conclude that the *mixins* being called more than once are more than the *mixins* being called only once.

There were some interesting cases that we found during the analysis of the results. In the SCSS dataset, there was a *mixin* which was called 382 times. Closer investigation revealed that this case was a *mixin* which was used for generating selectors having different *Media Queries* [12]. Media Queries provide the possibility of defining alternative styles for different media, e.g., a high-resolution monitor, or the display of a mobile device. It turned out that the developer called this *mixin* inside the majority of the selectors to avoid the effort needed to rewrite the complete Media Query declaration. On the other hand, in the SASS dataset, there was a website for which designers used the same animation for several elements in the web pages. Consequently, 75 *mixin* calls referred to a *mixin* which included style declarations for these animations. Finally, the maximum number of calls to a *mixin* in the LESS dataset was 72, which occurred for a *mixin* that was used for defining the size of fonts in the target documents. In other words, this *mixin* was called whenever a `font-size` was to be defined. These cases essentially show that *mixins* can be employed a wide range of purposes when developing style sheets.

2) Size of mixins: We counted the declarations which were placed directly or indirectly inside each *mixin*, as a measure for *mixins* size. By *indirectly*, we refer to the declarations which belong to selectors being nested under the examined *mixin*. Here, the goal is to investigate whether developers tend to keep *mixins* short, similar to what is suggested for their counterparts in traditional programming, i.e., functions. As shown in Figure 10, the median of the number of declarations is 3 in all three datasets. Further analysis shows that only 20% of the *mixins* include more than 5 declarations in the whole dataset, suggesting that developers mostly prefer to develop *mixins* having 5 declarations or less.

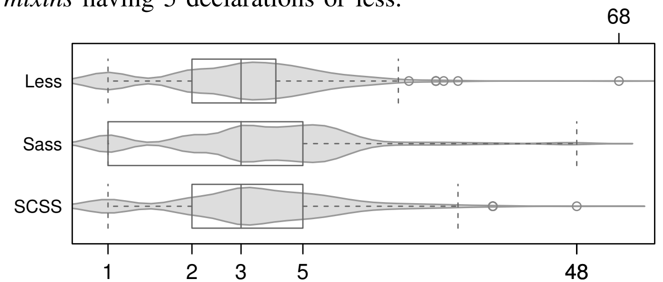

Fig. 10. Number of property declarations inside *mixins*

3) Number of parameters: We are also interested to investigate whether *mixins* tend to have a large number of parameters or not. As it is exhibited in Figure 11, the median value for the number of parameters in *mixin* declarations is equal to one in all datasets. We further found that 68% of the *mixins* have either one or no parameters. The difference in the number of declarations inside *mixins* and the number of *mixin* parameters possibly shows that, in most of the cases,

mixins either have hard-coded values for the majority of the properties defined inside their body, or their parameters are *reused* in multiple property declarations. We will investigate the reuse of parameters in the next subsection.

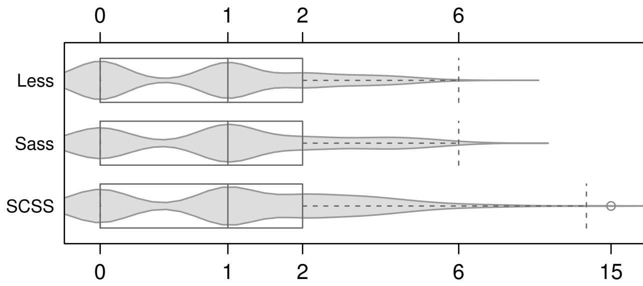

Fig. 11. Number of *mixin* parameters

4) Parameter reuse: We attempted to examine the hypothesis that parameters are reused in multiple declarations. We should first note that style properties in CSS are divided into two categories:

1) Properties which are common across different web browsers;
2) Properties which are specific to one web browser (i.e., vendor-specific properties).

As an example, to style font size in different browsers, one will only need to define the `font` property. On the other hand, when styling the `border-radius` property, the developer would need to define a different property for each web browser; for instance, `-webkit-border-radius` for Google Chrome or Safari and `-moz-border-radius` for Mozilla Firefox. Otherwise, the presentation of the target documents will differ across different web browsers. As a result, *mixins* can serve as a solution for grouping vendor-specific properties. In this situation, the same *mixin* parameter would be *reused* across different declarations corresponding to vendor-specific properties. An example of such a case is depicted in Figure 12.

```
.rounded(@radius: 2px) {
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
  border-radius: @radius;
}
```

Fig. 12. Parameter reuse across vendor-specific properties

Vendor-specific properties can be easily distinguished by examining whether the property name starts with one of the predefined prefixes by Word Wide Web Consortium (W3C) [13]. Our investigation showed that 42% of the *mixins* are used for grouping declarations associated with vendor-specific properties. When a *mixin* has at least one set of vendor-specific properties, on average only 6.6% of the declarations inside that *mixins* are not related to a vendor-specific property. As for parameter reuse, it turned out that 88.81% of the declarations associated with vendor-specific properties shared at least one of the *mixin*'s parameters. This indicates excessive amount of parameter reuse for vendor-specific properties. On the other hand, on average 19% of the *mixins* parameters were reused across properties which did not style the same property in the target documents (for instance, the variable `@w` is used both for `margin` and `padding` properties). This demonstrates that parameter reuse is also taking place for non-vendor-specific properties, although to a much smaller extent.

RQ3 Conclusions: Two thirds of the *mixins* are reused two or more times. Given that, any **migration/refactoring technique** should suggest extracting *mixins* even when there is a small number of selectors sharing the same set of declarations (i.e., to avoid declaration-level duplication). In addition, such a technique should rank higher the suggestions which have small number of parameters (i.e., small number of differences in property values), and include declarations for vendor-specific properties. Moreover, the **preprocessor language designers** should consider creating built-in *mixins* for vendor-specific properties, because a considerable amount of *mixins* (42%) are used for styling this kind of properties.

D. Extend Construct

Finally, we examine the usage of the *extend* construct. As mentioned in Section II, the *extend* construct is used to eliminate declaration-level duplication, similar to *mixins*. While *mixins* can have parameterized declarations in their body (in contrast to the *extend* construct), a parameterless *mixin* may be thought to have the same use as the *extend* construct. However, one should note that these constructs will result to different CSS code. A use of the *extend* construct will compile to a grouping selector (as shown in Figure 5), while the code inside a *mixin* will be *duplicated* in the generated CSS code in all the places where the *mixin* is called. In other words, the use of *mixins* introduces duplication in the generated CSS code; consequently, the developer may be tempted to use the *extend* construct over parameterless *mixins*.

On the other hand, when using the *extend* construct, the preprocessor compiler places the resulting grouping selector in the position of the selector being extended in the generated CSS code (Figure 5). This changes the relative order of the selectors in the style sheet, which may result in changing the presentation semantics of target documents. This happens because there is a dependency between two selectors which select the same element in the target document and style the same property. For instance, if two selectors style the `border-color` of an `img` element in the target document, this element will get the border color from the selector which appears *later* in the style sheet file. As a result, developers should take extra caution when using the *extend* construct, and make sure that these kind of dependencies (i.e, the *order dependencies* [5]) will not break. This might be a factor that makes developers reluctant to use the *extend* construct.

As shown in Table I, on average there were around 5 usages of the *extend* construct per file, in the SASS and SCSS datasets (in total 204 and 676 usages, respectively). At the same time, we did not find any use of the *extend* construct in the LESS dataset. This could be justified by the fact that the *extend* construct was more recently introduced in the LESS

preprocessor (version 1.4 released in June 2013), so developers might have not started yet using this feature in a systematic way.

On the other hand, we observed that the average number of calls to parameterless *mixins* in each file is 8, 4 and 6, respectively for LESS, SASS and SCSS datasets (Table I). The higher value for the LESS dataset might be explained from the fact that developers did not use the *extend* construct as an alternative solution, because it was not supported by the LESS preprocessor until recently.

For the SASS and SCSS datasets where developers used the *extend* construct, we conducted further analysis to understand if there is any preference for using *extend* construct over the parameterless *mixin* or vice versa. Figure 13 displays the Venn diagrams showing the percentage of the websites (out of the total number of websites in the corresponding dataset) which only used one of the constructs or both of them (the overlapping area). As it can be observed, both in SASS (Figure 13a) and SCSS (Figure 13b) datasets, the websites that used only parameterless *mixins* outnumber the ones which used only the *extend* construct.

Figure 13c shows the Venn diagram for both SASS and SCSS datasets combined together including 100 websites in total. We can clearly see that developers have a preference to parameterless *mixins* over *extend*, since in 28% of all websites they exclusively used parameterless *mixins*, while in only 9% of all websites they exclusively used the *extend* construct. Therefore, we may conclude that developers mostly tried to avoid the caveats associated with the *extend* construct, while accepting the duplication in the generated CSS code resulting from the use of parameterless *mixins*. Nevertheless, previous research showed that declaration-level duplication in the generated CSS code can be safely refactored and eliminated in some cases [5], and thus parameterless *mixins* could be replaced with the *extend* construct.

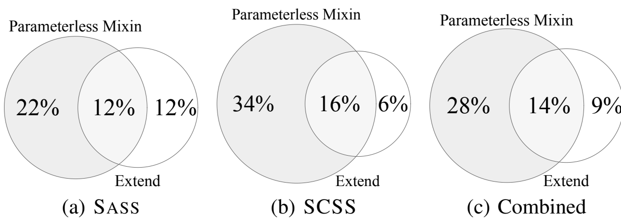

Fig. 13. Percentage of websites using *extend* or parameterless *mixins*

RQ4 Conclusions: Developers tend to prefer using parameterless *mixins* over the *extend* construct, possibly because *mixins* do not affect the presentation semantics of the target documents. As a result, any **migration/refactoring technique** should give higher priority to opportunities introducing parameterless *mixins*, especially when the alternative solution using the *extend* construct cannot guarantee that the presentation of the target documents will be preserved. The **preprocessor compilers** can be enhanced to warn developers about potential styling bugs caused by the incautious use of the *extend* construct.

V. THREATS TO VALIDITY

For minimizing the threats to the *external* validity of this study, we selected two CSS preprocessors which are known to be the most widely used by web developers [10], namely LESS and SASS. Additionally, we used the two dialects that SASS preprocessor supports (SASS and SCSS). Moreover, to make the results of the study as generalizable as possible, we examined 150 websites from a wide range of application domains.

To avoid *selection bias*, we included in the list of subjects the top-50 websites for each preprocessor language/dialect, as returned by the Google search engine. As a result, the authors of the paper were not involved in any kind of selection process.

To support the reliability of the study, we have made available the artifacts, which are necessary for replicating the experiment. These include the preprocessor files that we collected, the code we implemented for parsing LESS and SASS/SCSS files and querying their ASTs, the CSV files resulting from querying the ASTs, and the R scripts that we developed for the statistical analysis.

VI. RELATED WORK

To the best of our knowledge, this is the first empirical study on the use of CSS preprocessors. As mentioned before, we conducted this study to gain more knowledge about how developers utilize CSS preprocessors with the ultimate goal of designing a migration/refactoring recommendation system that migrates pure CSS code to preprocessors, as a means to improve the maintainability of existing CSS code.

There are a few works in the literature focusing on the quality, and improving the maintainability of CSS code. Keller and Nussbaumer [14] compared human-written to machine-generated CSS code and conclude that the former has a higher abstractness (i.e., higher reusability) compared to generated code. Serrano [6] proposed HSS, a preprocessor for CSS supporting all features discussed in this paper with the exception of *nesting*. Since HSS has not been adopted by the industry, we did not use any websites using HSS as subjects in our empirical study.

Mesbah and Mirshokrae [15] developed an automated technique for detecting dead code (i.e., unused selectors) in CSS, which analyzes the runtime relationship between the CSS rules and DOM elements of dynamic web applications, and detects unmatched and ineffective selectors, overridden declaration properties, and undefined class values. Genevès et al. [16] pursued the same goal using static analysis and tree logics to detect unused CSS code.

In our previous work, we proposed a technique for safely refactoring CSS by detecting different kinds of declaration-level duplications, and eventually reducing the size of the CSS files [5]. As an alternative approach for eliminating the duplication of declarations, we can extract *mixins* or apply the *extend* construct in the preprocessor code, as we discussed in this paper. Bosch et al. [17] introduced an approach for reducing the size of CSS files by removing redundant style

declarations and rules based on static analysis. A migration/refactoring technique can also take advantage of the work by Bosch et al. to recommend opportunities for removing unnecessary code in CSS and/or preprocessor code.

In the literature, there are several empirical studies on the use of language features in different languages and technologies with similar goals to this paper, e.g., understanding how developers have adopted these language features. For instance, Ernst et al. [18] investigated how C preprocessors are used in practice, by conducting an empirical study on 26 publicly available C programs, using a tool which includes approximate, Cpp-aware parsers for expressions, statements, and declarations. Tempero et al. [19] studied the use of inheritance in Java programs. They used different metrics, such as Depth of Inheritance, extracted from the bytecode of the subject systems for their analysis. Grechanik et al. [20] conducted a large-scale study on the use of object-oriented features including classes, methods, fields and conditional statements on 2000 open-source Java projects. They represented the information about the source code in a relational database and used SQL to extract the required metrics about different features. Gil and Lenz [21] conducted an empirical study on how Java developers take advantage of method overloading in 99 open source Java programs. Similar to [19], they also used bytecode for data collection. Xiaoyan et al. [22] investigated the frequency of different statement types (e.g., `if`, `return`, function declarations) in 311 projects written in C, C++ and Java. They extracted this information from an XML representation (i.e., srcML) of the source code of subject systems. Dyer et al. [23] conducted a very large-scale study on 31K open-source Java projects to find usages of new Java language features over time. This is done on the Abstract Syntax Tree (i.e., AST) of the source code of the subject systems. Richards et al. [24] studied the use of dynamic language features in JavaScript applications, using an instrumented web browser. Callaú et al. [25] conducted an empirical study on the use of the reflection feature in 1000 Smalltalk projects by statically tracing the features being used from the AST of the source code. Martin et al. [26] examined the use of GNU Make's language features (such as functions, macros, lazy variable assignments and the Guile embedded scripting language) in around 12k make files of 250 open source projects. They used TXL to define a custom grammar for Makefiles to extract and count instances of features. In our analysis, we used the AST of the parsed preprocessor files to extract the required information, similar to other works including [25], [23].

VII. Conclusions and Future Work

In this paper, we examined the preprocessor codebase of 150 websites to investigate the usage patterns of four language features, namely variables, *nesting*, *mixins* and *extend* constructs. We found out that developers frequently use all these features whenever possible, and gained some valuable knowledge which certainly can help us in devising migration/refactoring techniques and providing feedback to the preprocessor language designers. In summary the take-home messages of the study are:

1) Developers have a clear preference to global variables (89.28% of the variable declarations have a global scope), and especially variables storing color values (45.98% of the variable declarations have a color value).
2) Developers widely use the *nesting* feature (78% of the selectors are nested), even in very shallow *nesting* hierarchies consisting of one or two levels.
3) Developers tend to reuse *mixins* (63% of the *mixins* are called two or more times). They also tend to create *mixins* with a small number of parameters (68% of the *mixins* have either one or no parameters), and a relatively small size (80% of the *mixins* include 5 or less declarations). Finally, 42% of the *mixins* are used for styling vendor-specific properties.
4) While both parameterless *mixins* and the *extend* construct can be used to eliminate declaration-level duplication in the preprocessor code, developers tend to prefer using parameterless *mixins* to avoid the caveats associated with the *extend* construct.

While the gained knowledge in this paper is valuable for future research, we acknowledge the need for a qualitative user study with real-world developers, for triangulating the results of our quantitative study. Unfortunately, as the websites which have been investigated in this paper were collected using a web search engine (Google), we did not have access to the developers of the analyzed preprocessor files. Moreover, we anticipate that such qualitative study would require even more analysis and in-depth discussion, for each of the analyzed preprocessor features, which would be certainly beyond the space limitations of this paper. Ideally, we would replicate the same study on the preprocessor code collected from projects hosted on repository hosting websites (e.g., Github), and augment the results with qualitative data collected from their developers. We leave such a study for future work.

Another interesting possible future research direction is to study the evolution trends in preprocessor codebases. To this end, we aim at studying the history of the open-source preprocessor libraries and frameworks (such as Bootstrap) to investigate developer practices in refactoring preprocessor code. This can help us in designing refactoring techniques which are aligned with the developer needs. Moreover, as mentioned before, the lessons learned in this study provide us insights for devising techniques to automatically migrate existing CSS code to preprocessor code. As the next step, we are planning to develop such a technique that will be available in the form of a plug-in for a state-of-the-art IDE.

Acknowledgment

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds de Recherche du Québec – Nature et Technologies (FRQNT), and the Faculty of Engineering and Computer Science at Concordia University.

REFERENCES

[1] World Wide Web Consortium. CSS specifications. http://www.w3.org/Style/CSS/current-work.

[2] Mozilla Developer Network, "Web developer survey research," https://hacks.mozilla.org/2010/11/its-all-about-web-developers/, Mozilla, Tech. Rep., 2010.

[3] Web Technology Surveys. Usage of CSS for websites. http://w3techs.com/technologies/details/ce-css/all/all.

[4] H. W. Lie and B. Bos, *Cascading Style Sheets: Designing for the Web*, 3rd ed. Boston, MA, USA: Addison-Wesley Professional, 2005.

[5] D. Mazinanian, N. Tsantalis, and A. Mesbah, "Discovering Refactoring Opportunities in Cascading Style Sheets," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 496–506.

[6] M. Serrano, "HSS: A Compiler for Cascading Style Sheets," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2010, pp. 109–118.

[7] H. Catlin. SASS: Syntactically Awesome Style Sheets. http://sass-lang.com/.

[8] A. Sellier. LESS - The dynamic stylesheet language. http://lesscss.org/.

[9] Google Inc., "Google Closure Tools," https://developers.google.com/closure.

[10] C. Coyier. Popularity of CSS Preprocessors. http://css-tricks.com/poll-results-popularity-of-css-preprocessors/.

[11] U.S. General Services Administration, "CSS coding styleguide," https://pages.18f.gov/frontend/css-coding-styleguide/preprocessor/.

[12] "Media Queries," http://www.w3.org/TR/css3-mediaqueries/, World Wide Web Consortium, Tech. Rep., June 2012.

[13] "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, Assigning property values, Cascading, and Inheritance," http://www.w3.org/TR/CSS21/syndata.html, World Wide Web Consortium, Tech. Rep., June 2011.

[14] M. Keller and M. Nussbaumer, "CSS code quality: a metric for abstractness; or why humans beat machines in CSS coding," in *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2010, pp. 116–121.

[15] A. Mesbah and S. Mirshokraie, "Automated analysis of CSS rules to support style maintenance," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012, pp. 408–418.

[16] P. Genevès, N. Layaïda, and V. Quint, "On the analysis of cascading style sheets," in *Proceedings of the 21st International Conference on World Wide Web (WWW)*, 2012, pp. 809–818.

[17] M. Bosch, P. Genevès, and N. Layaïda, "Automated refactoring for size reduction of css style sheets," in *Proceedings of the 2014 ACM Symposium on Document Engineering (DocEng)*, 2014, pp. 13–16.

[18] M. D. Ernst, G. J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions On Software Engineering*, vol. 28, no. 12, 2002.

[19] E. Tempero, J. Noble, and H. Melton, "How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software," in *Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, 2008, vol. 5142, pp. 667–691.

[20] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An Empirical Investigation into a Large-scale Java Open Source Code Repository," in *Proceedings of the 2010 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010, pp. 1–10.

[21] J. Gil and K. Lenz, "The use of overloading in java programs," in *Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, 2010, vol. 6183, pp. 529–551.

[22] X. Zhu, E. J. Whitehead, C. Sadowski, and Q. Song, "An analysis of programming language statement frequency in c, c++, and java source code," *Software: Practice and Experience*, vol. 45, pp. 1479–1495, 2014.

[23] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 779–790.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the 31th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 1–12.

[25] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger, "How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, 2011, pp. 23–32.

[26] D. H. Martin, J. R. Cordy, B. Adams, and G. Antoniol, "Make It Simple - An Empirical Analysis of GNU Make Feature Use in Open Source Projects," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC)*, 2015.