A Survey on CSS Preprocessors

Ricardo Queirós

ESMAD, Department of Informatics, Polytechnic of Porto, Porto, Portugal
`ricardoqueiros@esmad.ipp.pt`

Abstract

In the Web realm, the adoption of Cascading Style Sheets (CSS) is unanimous, being widely used for styling web documents. Despite their intensive use, this W3C specification was written for web designers with limit programming background. Thus, it lack several programming constructs, such as variables, conditional and repetitive blocks, and functions. This absence affects negatively code reuse, and consequently, the maintenance of the styling code. In the last decade, several languages (e.g. Sass, Less) appeared to extend CSS, defined as CSS preprocessors, with the ultimate goal to bring those missing constructs and to foster stylesheets structured programming. The paper provides an introductory survey on CSS Preprocessors. It gathers information on a specific set of preprocessors, categorizes them and compares their features regarding a set of predefined criteria such as: maturity, coverage and performance.

1998 ACM Subject Classification D.3.4 Processors / Preprocessors

Keywords and phrases CSS, Preprocessors, Web formatting

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.8

1 Introduction

Nowadays, when it comes to Web frontend development, we need to understand three languages with different purposes: HyperText Markup Language (HTML) for structuring content, Cascading Style Sheets (CSS) to style it and JavaScript to attach some behaviour. Regarding CSS, several surveys show that this W3C specification is used in more than 95% of the websites [2], revealing its enormous importance on the process of constructing a website. A CSS file organizes a set of styling rules in selectors which has the responsibility to select the elements of the target documents that should be styled. Although its power, it lacks many programming constructs such as variables, conditional blocks and functions. This absence leads CSS developers to copying style declarations from one selector to another (e.g. code cloning) fostering code duplication and harming code maintenance. In a previous work [3], the CSS code of 38 high traffic websites were analysed and it was found that, on average, more than 60% of the CSS declarations were duplicated across at least two selectors.

In order to address these kind of issues, CSS preprocessor languages were introduced in the last decade. The code written in a CSS preprocessor can include programming constructs and, thereafter, be compiled to pure standardise CSS. Currently, there are several CSS preprocessors (e.g. Sass[1], Less [2], Stylus [3], PostCSS [4]), and their use is becoming a fast growing trend in the industry. In 2012, an online survey [1] with more than 13,000 responses from Web developers, conducted by a famous website focusing on CSS development, showed

[1] Available at `http://sass-lang`.
[2] Available at `http://lesscss.org/`.
[3] Available at `http://stylus-lang.com/`.
[4] Available at `http://postcss.org/`.

that 54% of the respondents use a CSS preprocessor in their development tasks. Despite these promising numbers, there are many developers that understands the power of CSS preprocessors but do not know which preprocessor should use.

In this paper we make a simple survey on CSS preprocessors. For this study, we start by selecting active preprocessors with a reasonable amount of popularity and documentation. Then, we describe the selected preprocessors and we compare them regarding a set of predefined criteria such as: maturity, coverage and performance. For the first criteria we based our study on the preprocessors activity in the Git repository hosting service, called GitHub, where the number of commits, releases, contributors and forks are enumerated and analysed. For the coverage criteria, we rely on a set of features (e.g variables, mixins, conditional, loops) that developers expect to have in such type of tools and check if these features are supported in the selected preprocessors. Finally, in the performance criteria, we conducted a simple experiment to test the performance of the selected preprocessors measuring their compilation time for a specific set of styling rules.

The remainder of this paper is organized as follows: Section 2 enumerates the typical programming constructs in a preprocessor. In the following section we initiate the survey based on three criteria: maturity, coverage and performance. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 Preprocessors

As previously stated, building a function, or inheritance are hard to achieve using solely CSS. When a webpage becomes more complex, we often see CSS files with a lot of rules and a reasonable level of redundancy. One way to save time, and to keep all these rules in a more flexible way, is through the use of CSS preprocessors. These tools make it easy to maintain large, complex style sheets, thanks to the use of features hitherto unavailable in the context of creating style sheets, such as variables, functions, and mixins. Using these constructs, code becomes more organized, allowing developers to work faster and make fewer mistakes. In this section we analysed several features typically included in CSS preprocessors. We demonstrate each feature through code examples with both preprocessor and CSS generation code. For the preprocessor language we use Sass syntax.

2.1 Variables

One of the main features of CSS preprocessors is the support for variables. This is a huge feature, since CSS developers lack the chance, for instance, to define a base color as a variable and reuse it all over on the CSS file. Instead they need hard-coded the hex or named color (or other property such as width, font-size, and others) as a property value for each rule. Using CSS preprocessors, in the case you need to change the color, you only have to do it in one place, in the variable inicialization. Using the traditional approach, you must change manually all occurrences of the color. In big and complex web apps this could be cumbersome and error-prone.

Variables work in a similar fashion to the those in any programming language, including the concepts for data types and scope. Also like every programming language, preprocessors have different syntax to represent variables' declaration and initialization. Despite their differences, their syntax are like classic CSS, so the learning curve for CSS developers is small. For instance, variables in Sass start with $ sign, in LESS with the @ sign and no prefix in Stylus. Both in Sass and Less, values are assigned with colon (:), and with equals sign (=) in Stylus. In the first example, we declare and initialise a variable and use it as a property value for two rules. The generated CSS will override all the variable occurrences with its value.

Listing 1 Sass code.

```
$primaryColor: #eeccff;

body {
  background: $primaryColor;
}
p {
  color: $primaryColor;
}
```

Listing 2 Generated CSS code.

```


body {
  background: #eeccff;
}
p {
  color: #eeccff;
}
```

Beyond using variables in property values, it is also possible to use variables in selectors or property names through interpolation. The next example shows how to use variables interpolation to define a name for a selector and a property:

Listing 3 Sass code.

```
$name: foo;
$attr: border;
p.#{$name} {
  #{$attr}-color: blue;
}
```

Listing 4 Generated CSS code.

```


p.foo {
  border-color: blue;
}
```

2.2 Nesting

As opposed to HTML, CSS nesting structure it's hard to write and to understand. Using preprocessors, we can combine various CSS rules by creating composite selectors. Basic nesting refers to the ability to have a declaration within another declaration. With the preprocessors nesting syntax, developers can organise stylesheets in a way that resembles HTML more closely, thus reducing the chance of CSS conflicts.

A classic example is when we have to style an HTML list composed by a set of links. In this case we have a structure with several depth levels (<ul>, <li>, <a>) and that requires some care when styling it through CSS rules. Using preprocessors, we have a more intuitive and compact way of styling these types of scenarios through nested declarations. The following example illustrates this situation:

Listing 5 Sass code.

```
ul {
  list-style: none;
  li {
    padding: 15px;
    display: inline-block;
    a {
      text-decoration: none;
      font-size: 16px;
      color: #444;
    }
  }
}
```

Listing 6 Generated CSS code.

```
ul {
  list-style: none;
}
ul li {
  padding: 15px;
  display: inline-block;
}
ul li a {
  text-decoration: none;
  font-size: 16px;
  color: #444;
}
```

In the case we want to reference the parent element in nested declarations, the & symbol should be used. The next example shows how to add pseudo-selectors using this technique:

Listing 7 Sass code.

```
a.myAnchor {
 color: blue;
 &:hover {
  text-decoration: underline;
 }
 &:visited {
  color: purple;
 }
}
```

Listing 8 Generated CSS code.

```
a.myAnchor {
 color: blue;
}
a.myAnchor:hover {
 text-decoration: underline;
}
a.myAnchor:visited {
 color: purple;
}
```

2.3 Extend

Inheritance is one of the key concepts in the object oriented programming paradigm. In OOPs, the concept of inheritance provides the idea of reusability. This means that we can add new features to an existing class without modifying it. This is possible by deriving a new class from the existing one. In the CSS realm, inheritance should be used when we need similar styled elements, but requiring slight changes between them. In Sass, we use the keyword @*extend* following by the base rule name we want to inherit. A classic example is the definition of two buttons: one for confirmation and one for cancellation.

Listing 9 Sass code.

```
.dialog-btn {
  box-sizing: border-box;
  color: #ffffff;
  box-shadow: 0 1px 1px 0
   rgba(0, 0, 0, 0.12);
}
.confirm {
  @extend .dialog-btn;
  background-color: #87bae1;
  float: left;
}
.cancel {
  @extend .dialog-btn;
  background-color: #e4749e;
  float: right;
}
```

Listing 10 Generated CSS code.

```
.dialog-btn, .confirm, .cancel {
  box-sizing: border-box;
  color: #ffffff;
  box-shadow: 0 1px 1px 0
   rgba(0, 0, 0, 0.12);
}
.confirm {
  background-color: #87bae1;
  float: left;
}
.cancel {
  background-color: #e4749e;
  float: right;
}
```

In the previous example, we note the CSS generated by Sass combined the selectors instead of repeating the same statements systematically, saving us precious memory.

In certain situations, you need to define styles that you just want to be extended and never used directly. A good example is when writing a library (such as Sass), where you just want to provide styles for users to extend in case they need to or ignore otherwise. For this type of scenario, Sass supports placeholder selectors (or placeholders). This type of selectors resemble the class and id selectors (# or.), But in this case the % symbol is used. The selectors can be used anywhere and will prevent rule sets from being rendered to CSS, except by extension.

Listing 11 Sass code.

```
%input style {
  font-size: 14px;
}
input {
  @extend %input style
  color: black;
}
```

Listing 12 Generated CSS code.

```
input {
  font-size: 14px;
}
input {
  color: black;
}
```

In this example, the %input-style rule will not be generated for CSS. However if we extend this rule, we will see that it is already possible to obtain the rule in the generated CSS.

2.4 Mixins

Mixins can be seen as a simplified version of constructors in object-oriented programming languages. In fact, it may include styles in the same way that @extend does but as the ability to provide arguments, thus making it a valuable tool for dynamic code reuse and redundancy reduction. The @*mixin* directive is used to create mixins and the @*include* directive is used to invoke them. In the next example, a mixin is defined to render squares with colors and sizes passed as arguments to the mixin:

Listing 13 Sass code.

```
@mixin square($size, $color) {
  width: $size;
  height: $size;
  background-color: $color;
}
.small-blue-square {
  @include square(20px,
   rgb(0,0,255));
}
.big-red-square {
  @include square(300px,
   rgb(255,0,0));
}
```

Listing 14 Generated CSS code.

```
.small-blue-square {
  width: 20px;
  height: 20px;
  background-color: blue;
}
.big-red-square {
  width: 300px;
  height: 300px;
  background-color: red;
}
```

Another typical example is when a property requires prefixes to work in all browsers, such as the transform property:

Listing 15 Sass code.

```
@mixin transform-tilt($tilt) {
  -webkit-transform: $tilt;
  ms-transform: $tilt;
  transform: $tilt;
}
table:hover {
  $tilt: rotate(15deg);
  @include transform-tilt($tilt);
}
```

Listing 16 Generated CSS code.

```
table:hover {
  -webkit-transform:
    rotate(15deg);
  -ms-transform: rotate(15deg);
  transform: rotate(15deg);
}
```

2.5 Functions

Typically, preprocessors support a long list of predefined functions. They serve all sorts of purposes, including string manipulation, color-related operations, and some useful math methods such as random and round. The next example uses the darken function that dims a particular color by a certain percentage:

Listing 17 Sass code.

```
$my-blue: #2196F3;

a {
  padding: 10px 15px;
  background-color: $my-blue;
}
a:hover {
  background-color:
    darken($my-blue,10%);
}
```

Listing 18 Generated CSS code.

```
a {
  padding: 10px 15px;
  background-color: #2196F3;
}
a:hover {
  background-color: #0c7cd5;
}
```

You can also define your own functions and invoke them anywhere through role directives. In Sass, the role directives are similar to the mixins, but instead of returning a set of properties, they return values through the *@return* directive. The definition of a function is done with the *@function* directive. The next example shows how to define a function for calculating the width of a column based on the width of its parent, the number of columns, and the margins size:

Listing 19 Sass code.

```
$ctn-width: 100%;
$col-count: 4;
$mgin: 1%;
@function getColW($w, $cols, $mgin){
  @return ($w / $cols) - ($mgin * 2);
}
.container {
  width: $ctn-width;
}
.column {
  width: getColW($ctn-width,$col-count,$mgin);
  height: 200px;
}
```

Listing 20 CSS code.

```
.container {
  width: 100%;
}
.column {
  width: 23%;
  height: 200px;
}
```

2.6 Conditional and cyclic structures

Some preprocessors support decision and cyclic structures. You can actually use directives like *@if*, *@for*, *@each*, and *@while*. The next two examples show how to use these directives in two different contexts: decision making at a given value and iteration under variables to create similar statements. In the first example, the value of a variable is inspected and a set of actions is executed by its value. Note that if the condition in the *@if* directive evaluates to false, the set of expressions that follow the *@else* directive is executed.

Listing 21 Sass code.

```
$boolean: true !default;
@mixin foo() {
  @debug "$boolean is #{$boolean}";
  @if $boolean {
    display: block;
  }
  @else {
    display: none;
  }
}
.some-selector {
  @include foo();
}
```

Listing 22 CSS code.

```
.some-selector {
  display: block;
}
```

The following example uses the *@for* directive to define multiple style declarations named by interpolation:

Listing 23 Sass code.

```
$squareC: 3;

@for $i from 1 through $squareC {
  #square-#{$i} {
    background-color: red;
    width: 50px * $i;
    height: 120px / $i;
  }
}
```

Listing 24 CSS code.

```
#square-1 {
  background-color: red;
  width: 50px;
  height: 120px;
}
#square-2 {
  background-color: red;
  width: 100px;
  height: 60px;
}
#square-3 {
  background-color: red;
  width: 150px;
  height: 40px;
}
```

3 CSS preprocessors survey

Currently, there are several CSS preprocessors offering similar features with a different syntax. In order to select one, users typically start by reading some sites where a subset of them is described and compared based on a adhoc list of advantages/disadvantages. This approach could be dangerous since you could invest time in one preprocessor and discover later that is not well document or do not have an active developers community, or do not support a specific language binding or feature, or even, has poor performance while running in a production environment. This work aims to offer a consistent study to base a sustained choice of a CSS preprocessor. In order to accomplish this challenge, we start by selecting a specific set of preprocessores based on a very popular 2016 frontend tooling survey [4] where 4,715 developers (mostly people with 5 to 10 years of frontend technologies experience) answered questions about Web tools and standards. Analysing deeply the survey, we can find the responses to the question “What is your CSS Processing tool of choice?”. The results were conclusive as shown in Table 1.

Table 1 CSS Processing tool.

Preprocessor	# Votes	Percentage	% Diff (to 2015)
Sass	2,989	63.39%	-0.56%
Less	478	10.14%	-5.05%
Stylus	137	2.91%	-0.84%
PostCSS	392	8.31%	N/A
No Preprocessor	643	13.64%	-1.4%
Other	73	1.55%	-0.52%

Analysing the results, Sass is still the CSS processing tool of choice for the majority of respondents with 63.39%. When compared to last years results, Less usage has dropped from 15.19% to 10.14% (down 5.05%).

Also we can state that the percentage of respondents using CSS processing tools grows to 86.36% (more 1.4% from 2015). This reenforces the importance of CSS processing tools in the Web development workflows.

We have included tools that are not pure preprocessors, such as PostCSS that is coined as a CSS postprocessor tool. These type of tools are becoming essential for frontend developers since they can apply specific actions after the CSS has been generated. Among the supported actions, the most popular are applying vendor prefixes automatically, creating pixel and IE8 media query fallbacks. The most notable example of this kind of tools is PostCSS. In this survey one can notice that PostCSS is used by 8.31% of respondents. Actually, it's usage is likely to be slightly higher, as this survey does not count for those respondents who use it in combination with another processing tool.

Based on this survey, we select Sass, Less and Stylus for the CSS Processors survey. In the next subsections the three preprocessors are described and compared based on three facets: maturity, coverage and performance.

3.1 Maturity

It is difficult to determine which preprocessor is most widely used or have more impact. In this study we made an effort to measure the popularity/impact of the CSS preprocessors in the Web. Popularity, is usually influenced by an active community. When choosing a specific language, library or framework for your project, consider first looking at the community behind and check if it is actively developed for bug fixes and new features. A large and active community will also make it easier to get support and will provide more learning advices and resources. Beyond that, verify how detailed, well-written and organized the documentation is.

Various methods of measuring tools popularity have been proposed, each subject to a different bias over what is measured. In this context, we will focus on two: counting the number of times the language name is mentioned in web searches (using Google Trends) and comparing the activity in GitHub where all the selected preprocessors are stored.

Firstly, we search in Google Trends for the three CSS preprocessors. The results are shown in Figure 1.

It appears there is much more activity with Sass and Less, rather than Stylus. From 2015, Sass is the CSS preprocessor most typed in Google searches.

The second method chosen to measure the impact/popularity of the selected preprocessors was the analysis of their Github activity. In fact, the three preprocessors chosen are pretty active on Github, as you can see in Table 2.

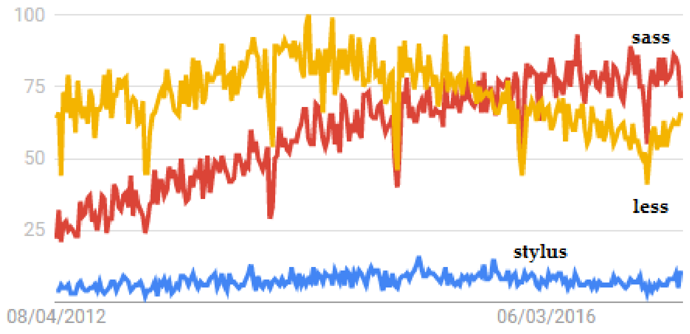

Figure 1 Interest over time on Sass, Less and Stylus preprocessors – Google Trends.

Table 2 CSS Processors Maturity.

GitHub data	Sass	Less	Stylus
First release date	0.1.0 (Oct/2006)	1.0 (Apr/2010)	0.02 (Jan/2011)
Last release date	3.4.23 (Dec/2016)	2.7.2 (Jan/2017)	0.54.5 (Apr/2016)
# Open issues	182	241	159
# Pull requests	7	28	17
# Commits	6241	2663	3881
# Releases	181	49	161
# Contributors	180	208	147
# Stars	9557	14590	7973
# Forks	1763	3353	949

Although relatively recent, these initiatives have been growing with the evolution of the HTML and CSS Web standards. Sass is the oldest and the one with greater update frequency of commits and releases. However, Less, despite being younger, is the most popular, with a larger number of stars and forks. The number of forks is relevant. A fork is a copy of a repository. Forking a repository allows you to freely experiment a repo (with changes) without affecting the original project. Thus, this means that most people are using Less base code to start their own projects. Regarding Stylus, it presents the lower values of the three in almost all the indicators.

3.2 Coverage

In the coverage criteria we will make a comparison of the three CSS preprocessors regarding the support for the most popular features, such as, variables, mixins, conditionals and loops.

In Table 3 we present the comparison on CSS preprocessors variable features.

All the preprocessores have the basic ability to declare variables and use them later. However, Less do not support the feature of default variables, that is, variables that are overwritten by regular ones, no matter when they are declared. Nevertheless, variable hosting (lazy) is only supported by Less, where variables can be declared after being used. The Stylus preprocessor is the only one that allows you to use the value of a previously declared property elsewhere (variables lookup). Finally, all the preprocessors support interpolation. This means that we can use variables as parts of selectors, properties, values, as parameters of the calc function – a CSS function that performs calculations to determine CSS property values – and even, place a set of selectors into a variable and reuse it.

Table 3 Variables features comparison.

Features	Sass	Less	Stylus
Basic	Yes	Yes	Yes
Default	Yes	No	Yes
Lazy	No	Yes	No
Lookup	No	No	Yes
Interpolation	Yes	Yes	Yes

Table 4 Mixins features comparison.

Features	Sass	Less	Stylus
Basic	Yes	Yes	Yes
Params	Yes	Yes	Yes
Params-named	Yes	Yes	Yes
Arguments	Yes	Yes	Yes

Listing 25 Less loop example.

```
.generate-column(@i: 1) when (@i =< 4) {
  .col-@{i} {
    width: @i * (100% / 4);
  }
  .generate-column(@i + 1);
}
```

Mixins is another powerful feature of CSS preprocessors. The CSS preprocessors mixins support is shown in Table 4.

All the CSS preprocessors support the mixins main features, namely, the inclusion of mixins (basic), mixins that can receive parameters passed to them (params), mixins that have named placeholders for each parameter passed to them (params-named) and, finally, mixins with an unknown number of arguments passed to them (arguments).

Other important feature of CSS preprocessors is the support for conditionals. Conditionals are useful when we want that a part of our code to be executed only certain condition is evaluated as true (or false). Although Less uses a slightly unconventional syntax (*), all the preprocessors support if statements within a declaration. Ternary operators allow for a single-line if/else test in the format of $x > 0?true : false$. Less do not support this feature and Sass uses an unusual syntax. Regarding, the capacity to interpolate “if statements” inside property names, only Sass and Stylus support it. Table 5 summarises all the conditional supported features.

Regarding loops (Table 6), we only compared loops, that increment values (basic), and loops that iterate though items in a list (intermediate). All preprocessors support both, although, Less needs to call the function as shown in Listing 25. Thus, we can conclude that Sass and Stylus are the preprocessores with greater support for the most popular features. Less has support for conditional and loops, but relies mostly on unusual syntax to achieve it.

Table 5 Conditional features comparison.

Features	Sass	Less	Stylus
Basic	Yes	Yes *	Yes
Ternary	Yes *	No	Yes
Property	Yes	No	Yes

Table 6 Loop features comparison.

Features	Sass	Less	Stylus
Basic	Yes	Yes *	Yes
Intermediate	Yes	Yes *	Yes

Listing 26 Benchmark script excerpt.

```
var path = require('path');
var fs   = require('fs');
var n = 999;

// Sass
var libsass = require('node-sass');

var scss = '$size: 100px;';
scss += '@mixin icon { width: 16px; height: 16px; }';
for ( i = 0; i < n; i++ ) {
    scss += 'body { h1 { a { color: black; } } }';
    scss += 'h2 { width: $size; }';
    scss += 'h1 { width: 2 * $size; }';
    scss += '.search { fill: black; @include icon; }';
}

var result =  libsass.renderSync({ data: scss });
console.log("Sass: ", result.stats.duration + "\t ms");

// Repeat for the other two preprocessors
...
```

3.3 Performance

In this subsection the three preprocessors are compared in terms of performance. This performance benchmark will be achived by measuring the compilation time of the preprocessors for different sizes of files.

For the experiment, we started by installing Node.js v6.10.2 (includes npm 3.10.10) in a machine running Windows 10 (64 bits), Intel Core i7-6700K – 4.00GHz, 16 GB RAM and SSD.

The next step was the creation of a JavaScript file with the test code. The aim of this test is to compare CSS preprocessors for parsings, nested rules, mixins, variables and math. An excerpt of the test is presented in Listing 26.

The test starts by the generation of the preprocessor code of three sizes (10/100/1000 KB). Then, it uses the internal function of each library for the compilation of the preprocessor code to the CSS format. The compilation process for each library is measured and the processing time are showned in the console. The results are presented in Table 7.

Table 7 Performance benchmark.

Preprocessors	10KB	100KB	1000KB
Sass	9ms	91ms	943ms
Less	63ms	287ms	2067ms
Stylus	133ms	581ms	3969ms

Based on these results, the main conclusion is that Sass is the fastest of the three. Please note, that the official implementation of Sass is open-source and coded in Ruby. In these tests we didn't use it. Instead, we used libSass a high-performance implementation in C.

4 Conclusions

CSS preprocessors are very powerful and they can help streamline your Web development process, especially if you are coming from a programming background. In this paper we surveys a set of CSS preprocessors regarding a set of predefined criteria such as: maturity, coverage and performance.

While it appears that Sass is more widely used and have better performance, Stylus demonstrates that covers very well the main and typical features of a CSS preprocessor. In fact, there isn't really a preprocessor that is better than the other. It usually comes down to the developer and what they are comfortable with using. The most important thing is to use one in order to help you make your CSS more maintainable and extendable.

As future work, the main idea is to extend this survey to other CSS preprocessors and include also some postprocessors that are being used nowadays (such as PostCSS and Rework). Other important upgrade is to enrich the tests not only based on the size of the file but in the diversity of block types used in code. This will create more realistic tests since many of the CSS preprocessors features are often used in production [2].

References

1 Chris Coyier. *Popularity of CSS Preprocessors*, 2012. CSS-Tricks. `http://css-tricks.com/poll-results-popularity-of-css-preprocessors`.

2 Davood Mazinanian and Nikolaos Tsantalis. An empirical study on the use of CSS preprocessors. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 168–178, 2016.

3 Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *22nd International Symposium on Foundations of Software Engineering*, pages 496–506, 2014.

4 Ashley Nolan. The State of Front-End Tooling 2016 – Results, 2016. Personal blog. `https://ashleynolan.co.uk/blog/frontend-tooling-survey-2016-results`.